1. This is the entity class which represents a persistent data structure in a database. This is mapped to the database table, and each instance of the class corresponds to a row in the table.
2. To mark it as an entity class , I use the @Entity annotation.
3. This annotation tells the JPA provider (e.g., Hibernate in most Spring Boot applications) that the class should be treated as an entity and mapped to a database table.
4. the @Id annotation is used to mark this field as the primary key. Also I use @GeneratedValue to specify that the primary key values are auto-increment
5. I used Lombok also so that with help of it I can generate getter setter and constructor automatically
6. To interact with entity class I have created a repository with name EmployeeRepository which is extending JPA repository.
7. JPA repository provides various methods for common database operations (e.g., save, delete, find by ID, etc.) related to the Employee entity. The <Employee, Integer> part specifies that this repository deals with entities of type Employee, and the primary key is of type Integer.
8. Then we have our controller class named as EmployeeController. To make this as controller we have used @RestController annotation. This annotation is used to define a class as a Spring REST controller. It is specifically designed for building RESTful web services that primarily produce JSON or XML responses

[@Controller is used to define a class as a Spring MVC controller. It is typically used for creating web applications that serve HTML views.

Controllers annotated with @Controller are responsible for handling incoming HTTP requests, processing the requests, invoking business logic, and returning a view (HTML template) as the response. They are commonly used in traditional server-side web applications.

@RestController is used to define a class as a Spring REST controller. It is specifically designed for building RESTful web services that primarily produce JSON or XML responses, rather than HTML views.

Controllers annotated with @RestController are responsible for handling HTTP requests, processing them, invoking business logic, and returning data objects as JSON or XML responses.

]

1. Also by using @ @Autowired I have injected the dependency of EmployeeService In controller.
2. Then I have created 2 api’s using get and post mapping. @getmapping and @postmapping annotations are used to map HTTP GET and POST requests to specific methods in the controller. They specify the URL paths at which these methods should respond to requests.
3. @RequestBody: This annotation is used to indicate that the employeee parameter of the addEmployeedetails method should be populated with data from the HTTP request body.
4. public ResponseEntity<List<Employee>> getallEmployee(): This method retrieves a list of employees from the employeeService and returns it as a ResponseEntity with an HTTP status of HttpStatus.OK. The list of employees is serialized into JSON and sent as the response body.
5. public ResponseEntity<Employee> addEmployeedetails(@RequestBody Employee employeee): This method accepts an Employee object as the request body, likely for adding a new employee record. It delegates this task to the employeeService and returns the newly added employee as a ResponseEntity with an HTTP status of HttpStatus.OK.
6. while integrating I faced the CORS issue so I created this class which help in integration of front end and backend
7. this code configures CORS settings in your Spring Boot application to allow requests from a specific origin (http://localhost:4200) and specifies which HTTP methods, headers, and credentials are allowed in cross-origin requests. This is important for security when your frontend and backend are hosted on different domains or ports.
8. For testing It I have used junit and Mockito.
9. @InjectMocks is injecting dependencies into the EmployeeController.
10. @Mock: This annotation is used to create a mock of the EmployeeService dependency.
11. @BeforeEach: This annotation is used to indicate that the method annotated with it (setup in this case and init in other) should be executed before each test method

20.a. testGetAllEmployee: This method tests the behavior of the getallEmployee method of the EmployeeController. It does the following:

**Creates a list of mock employee objects**.

Uses Mockito to mock the behavior of the employeeService.getallEmployee() method to return the mock employee list.

**Calls the employeeController.getallEmployee() method**.

Asserts that the HTTP response status code is HttpStatus.OK (indicating success) and that the response body contains the mock employee list.

testAddEmployeeDetails: This method tests the behavior of the addEmployeedetails method of the EmployeeController. It does the following:

**Creates a mock employee object**.

Uses Mockito to mock the behavior of the employeeService.addEmployeedetails method to return the mock employee.

**Calls the employeeController.addEmployeedetails method with a request employee object.**

Asserts that the HTTP response status code is HttpStatus.OK and that the response body contains the mock employee.

20.b. testGetAllEmployee Method:

**This test method checks the behavior of the getallEmployee method in the EmployeeService.**

It sets up the scenario by creating a list of mock employees and using Mockito to mock the behavior of the employeeRepository.findAll() method to return these mock employees.

**It then calls the getallEmployee method of the employeeService.**

Finally, it asserts that the result contains the expected number of employees and specific employee details.

testAddEmployeeDetails Method:

**This test method checks the behavior of the addEmployeedetails method in the EmployeeService.**

It prepares the scenario by creating a new employee (newEmployee) and another employee (savedEmployee) that is expected to be returned when employeeRepository.save(newEmployee) is called. It uses Mockito to mock this behavior.

**It then calls the addEmployeedetails method of the employeeService with newEmployee.**

Finally, it asserts that the result employee matches the expected details, which are set in savedEmployee.

1. In this code I designed a navigation bar with links to different sections of a web application, including dropdown menus for services. It uses Bootstrap for styling and Angular's routing mechanism (routerLink) to navigate between different views or components within the application.
2. Then I have created a service class in an Angular application responsible for making HTTP requests to interact with a backend server.
3. @Injectable Decorator: This decorator tells Angular that this service should be provided at the root level, making it a singleton service available throughout the application.
4. Class Declaration: class EmployeeServiceService: This is the declaration of the EmployeeServiceService class.
5. The constructor injects the HttpClient service into this service, allowing it to make HTTP requests.
6. baseURL Property:

baseURL = "http://localhost:8080";: in This we stores the base URL of the backend server

AddEmployee Method:

AddEmployee(employee: Employee): This method is used to send an HTTP POST request to add an employee to the backend server.

It uses the httpClient.post() method to send the employee data to the server and returns the resulting Observable.

getEmployeesList Method:

getEmployeesList(): Observable<Employee[]>: This method is used to send an HTTP GET request to retrieve a list of employees from the backend server.

It uses the httpClient.get() method to fetch the employee data from the server and returns an Observable containing an array of Employee objectsTop of Form

**Explaining ts file of employee component**

**I have imported some necessary files**

selector: 'app-employee-details': Specifies the component's selector, which is used to embed this component in the HTML template.

templateUrl: './employee-details.component.html': Specifies the HTML template file for this component.

styleUrls: ['./employee-details.component.css']: Specifies the CSS stylesheet(s) associated with this component.

employees!: Employee[]: it will be used to store the list of employees fetched from the server.

ngOnInit(): This method is part of the Angular lifecycle and is called when the component is initialized.

It calls the getEmployees method to fetch the list of employees from the servic