# **RESTful CRUD API using Spring Data JPA**

Rupasri Guruprasad

## [**Rupasri Guruprasad**](https://www.linkedin.com/in/rupasrigu/)

Software Developer | Java Developer | Application Support Analyst | Backend Engineer| Specialise in Java, Spring Framework, Hibernate

October 26, 2024

In today's journey, we’ll set up a simple Employee Management service to handle **CRUD** (Create, Read, Update, Delete) operations with Spring Boot, Spring Data JPA, and a RESTful API. The service manages Employee entities, allowing us to add, retrieve, update, and delete employee records. Through this example, we'll see how each application layer—model, repository, service, and controller—works together to build a clean, maintainable architecture.

### **Project Structure and Layers**

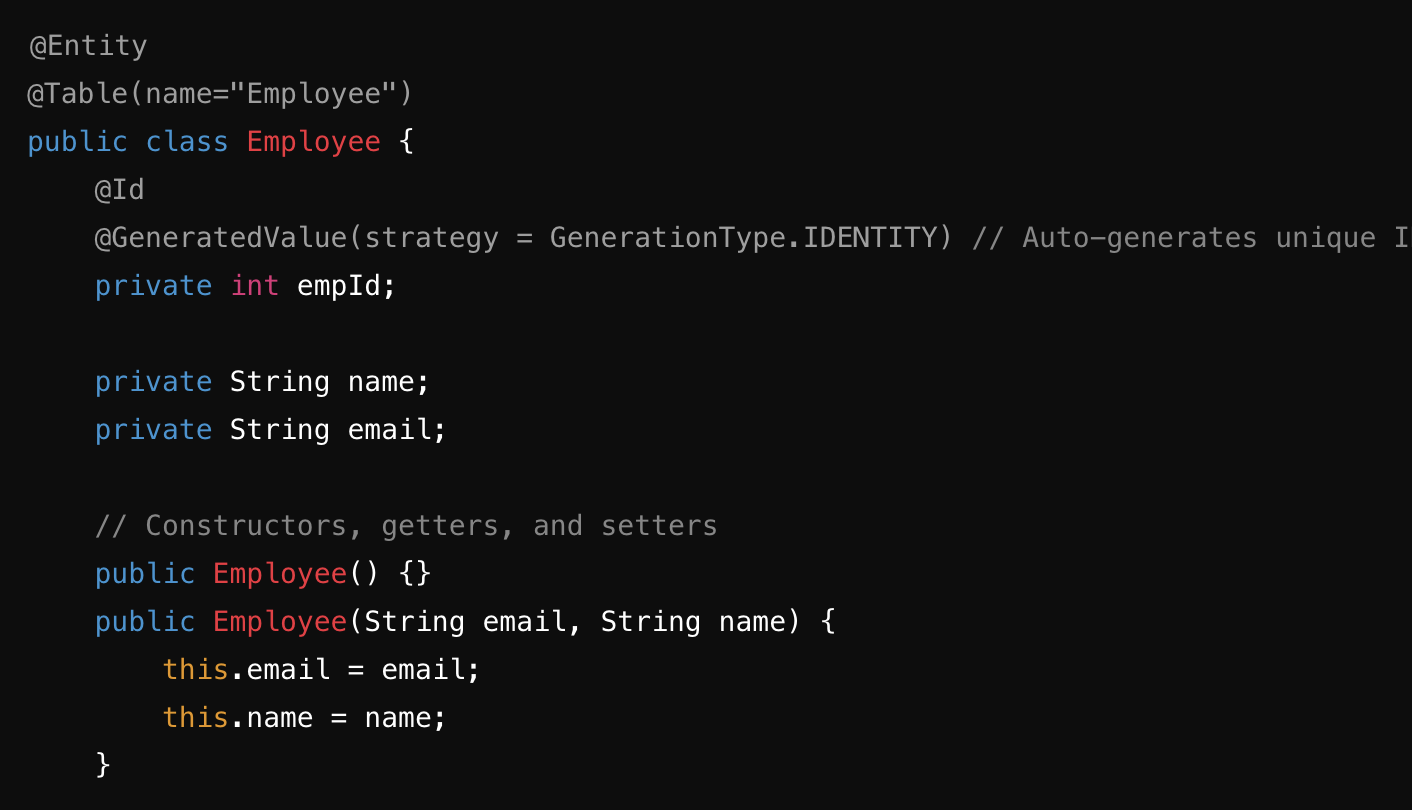
A typical Spring Boot REST application follows a layered structure:

1. **Model Layer**: Defines the data entities.
2. **Repository Layer**: Interacts with the database.
3. **Service Layer**: Contains business logic.
4. **Controller Layer**: Exposes REST endpoints.

With these layers in mind, let’s explore our Employee CRUD setup.

### **Model Layer**

The **Model Layer** defines the Employee entity, mapping it to a database table and specifying attributes like name, email, and empId with unique ID generation.

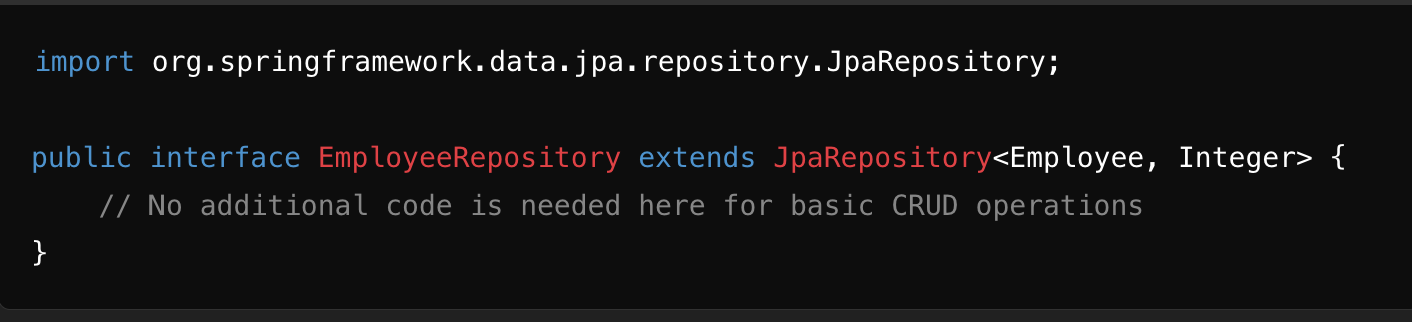


Model Class - Employee

* @Entity: Marks this class as a JPA entity, mapping it to a database table.
* @Table(name="Employee"): Specifies the table name.
* @Id: Indicates the primary key.
* @GeneratedValue(strategy = GenerationType.IDENTITY): Auto-generates unique values for the empId field, using the database’s identity column.

### **Repository Layer**

Our EmployeeRepository interface extends JpaRepository to simplify database interactions.



Repository Class

JpaRepository<Employee, Integer>: Provides built-in CRUD methods, where Employee is the entity type and Integer is the primary key’s data type.

### **Service Layer**

The EmployeeService class contains our business logic. We call methods from the repository layer here.



Service Layer - EmployeeService

Annotations explained:

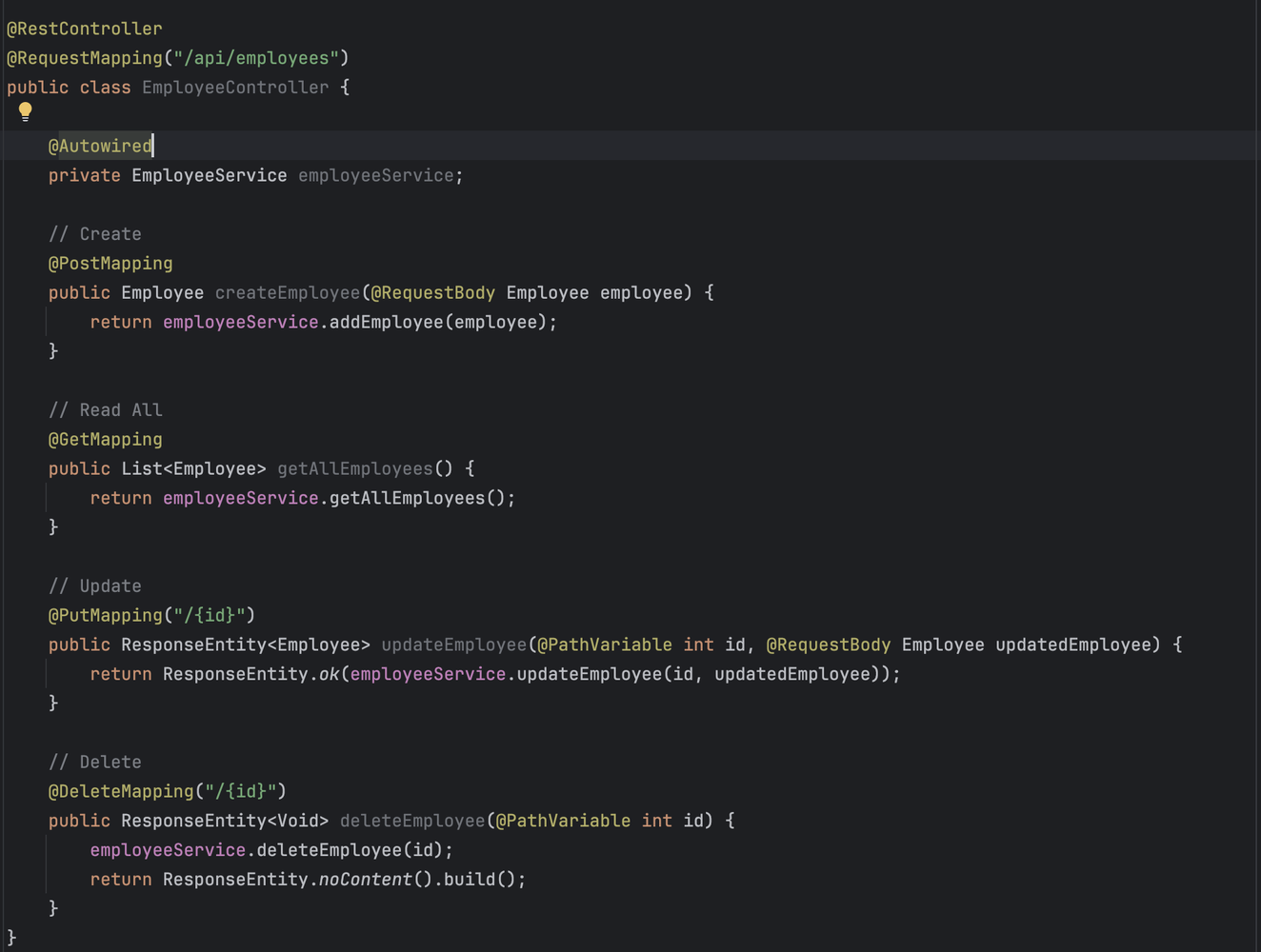
* @Service: Marks this class as a Spring service, making it injectable throughout the application.
* @Autowired: Injects EmployeeRepository, allowing us to perform database operations within the service.

Each method in EmployeeService:

* **addEmployee**: Adds a new employee.
* **getAllEmployees**: Retrieves all employees.
* **updateEmployee**: Finds an existing employee by ID, updates their name and email, and saves the changes.
* **deleteEmployee**: Deletes an employee by their ID.

### **Controller Layer: Exposing the API in EmployeeController**

Our controller layer exposes RESTful endpoints for the CRUD operations in EmployeeService, mapping each operation to a specific HTTP method.



Controller Layer

Annotations explained:

* @RestController and @RequestMapping: Define this class as a REST controller and set the base URL for endpoints as /api/employees.
* @PostMapping, @GetMapping, @PutMapping, @DeleteMapping: Specify HTTP methods and paths for creating, retrieving all, updating, and deleting employees.
* @RequestBody and @PathVariable: Map the request body to Employee and extract the {id} value from the URL.

The **ResponseEntity** classin Spring Boot is a powerful feature that helps control HTTP responses. It allows us to set the response status, headers, and body all in one place, giving flexibility over what our API returns to clients.In our EmployeeController, we use ResponseEntity to handle various scenarios, such as:

* Returning ResponseEntity.ok() to indicate successful responses, along with the employee data.
* Using ResponseEntity.noContent() for empty responses, such as after a successful deletion.

### **Wrapping Up**

By structuring our Employee service with clear layers, we’ve created a maintainable and scalable RESTful API. This CRUD setup, leveraging Spring Data JPA and Spring Boot, illustrates how easy it is to interact with data in a microservices-based architecture. Each layer—from model to controller—plays a role in ensuring clean separation of concerns, making it efficient to develop, test, and expand functionality.

Spring Boot makes backend data management a breeze, and with a setup like this, your application can handle much more than just employees.

Explore the complete code on GitHub for this project: [Microservices GitHub Repository](https://github.com/Rupa0807/Microservices/)

#techLearning #Java #SpringBoot #Microservices #LearnTogether #Day5