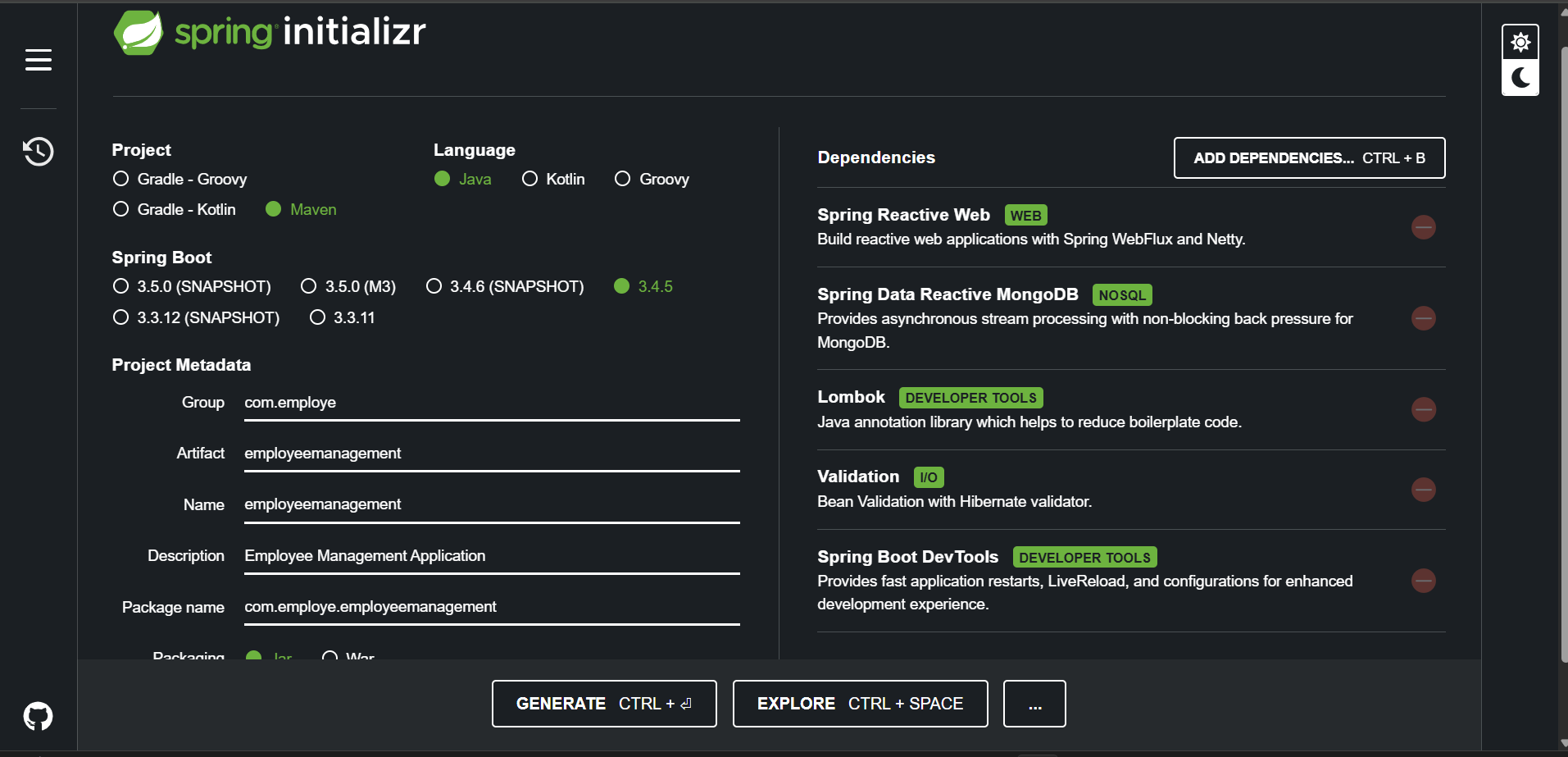
**Technologies & Tools**

|  |  |  |
| --- | --- | --- |
| 1 | FRONTEND | **HTML,CSS AND JS** |
| 2 | DESIGN FRAMEWORK | **BOOTSTRAP** |
| 3 | BACKEND | **SPRING BOOT (with spring webflux)** |
| 4 | DATABASE | **MONGODB** |
| 5 | API TESTING | **SWAGGER** |
| 6 | TESTING | **JUnit , Mockito** |
| 8 | Documentation | **PDF with ScreenShots** |

**Features to Include (Minimum Requirements):**

1. View All Employees
2. ➕ Add New Employee
3. ✏ Update Employee Details
4. ❌ Delete Employee
5. 🔎 Search or Filter Employees
6. 📋 Validation (name, email, etc.)
7. ✅ Unit Testing
8. API Testing with Swagger
9. Secure with Spring Security
10. Final PDF Documentation with Screenshots

****

**Project Setup Plan (Employee Management System – using Fetch + MongoDB)**

**Step 1: Spring Boot Initializer Setup**

1. Go to : <https://start.spring.io/>
2. **Project**: Maven
3. **Language**: Java
4. **Spring Boot Version**: Use latest stable
5. **Project Metadata:**
   1. Group: com.employee
   2. Artifact: employeemanagement
   3. Name: employeemanagement
   4. Package Name: com.employe.employeemanagement
6. **Dependencies:**
   1. **Spring Reactive Web**
   2. **Spring Data Reactive MongoDB**
   3. **Lombok**
   4. **Validation**
   5. **Spring Boot DevTools**
   6. **Spring Security**
   7. **Spring Boot TestSpringdoc OpenAPI (Swagger UI)-** **springdoc-openapi-starter-webflux-ui**
7. Then click **Generate**, unzip, and open in your IDE (IntelliJ, VS Code, etc.)

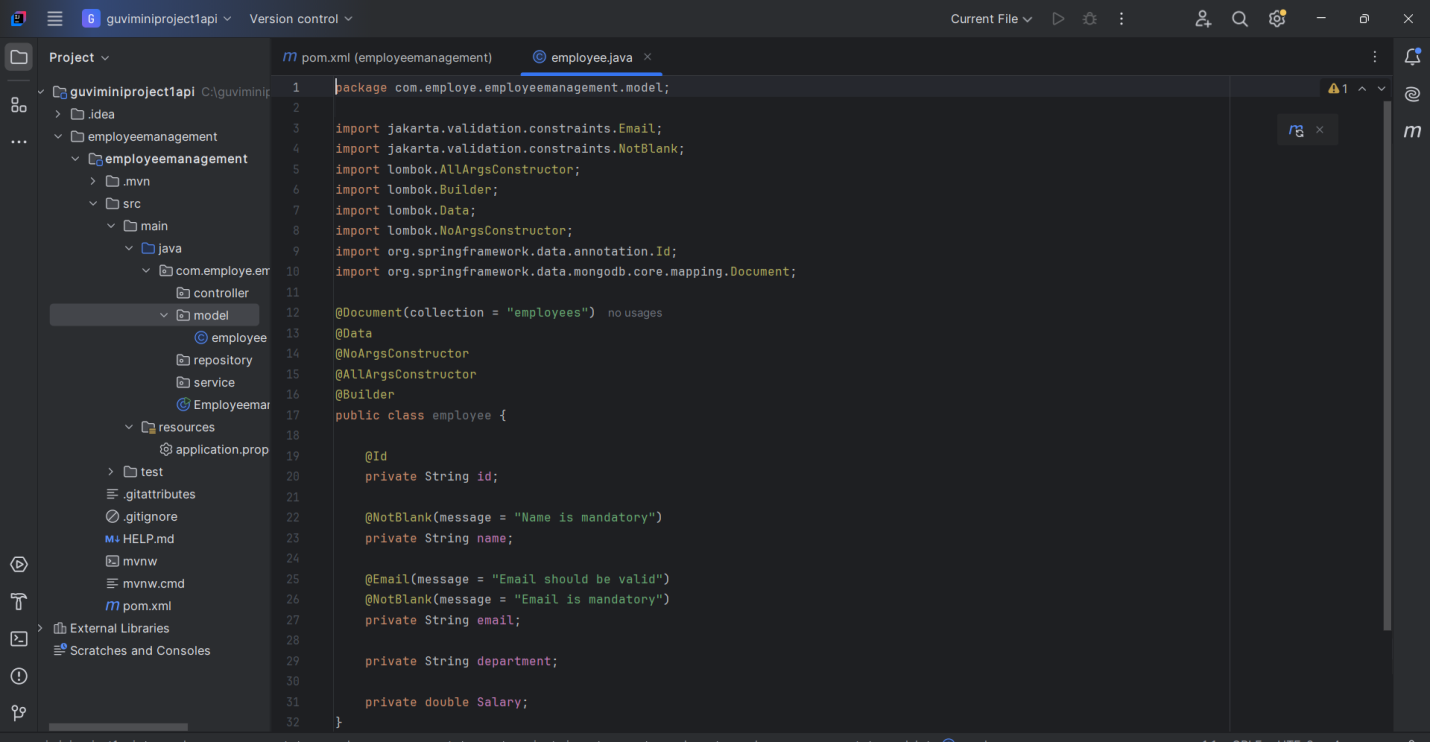
|  |
| --- |
|  |

**Step 2: Create packages**

1. Model
2. Repository
3. Service
4. Controller

**MODEL PACKAGE :**

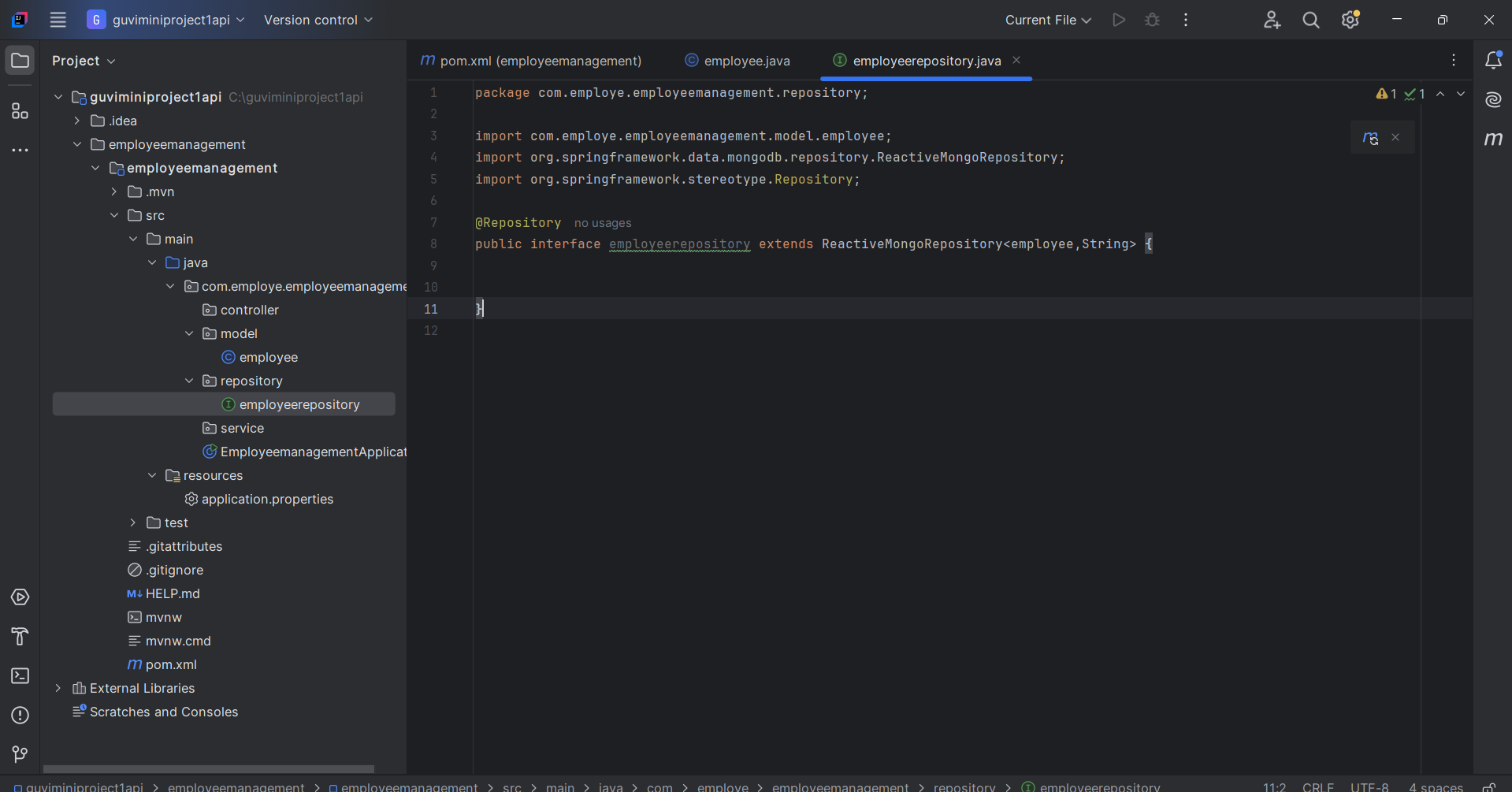
1. **CLASS EMPLOYEE**

****

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | | Annotation | Purpose | | --- | --- | | @Document(collection = "employees") | Tells Spring that this class maps to a MongoDB collection named employees. | | @Data | Lombok annotation that creates all the boilerplate (getters, setters, toString(), equals(), etc.) automatically. | | @NoArgsConstructor | Lombok creates a no-argument (default) constructor. | | @AllArgsConstructor | Lombok creates a constructor with all fields as parameters. | | @Builder | Helps you create objects in a clean, step-by-step way using the Builder pattern. | |  | |  | | | @NotBlank | | Validates that the field is not null and not empty (it must contain at least one non-whitespace character). Mostly used for String fields. | | | @Email | | Validates that the string provided is in a valid email format (like abc@example.com). | | | |

**REPOSITORY PACKAGE :**

1. **INTERFACE EMPLOYEEREPOSITORY**

****

**Explanation**

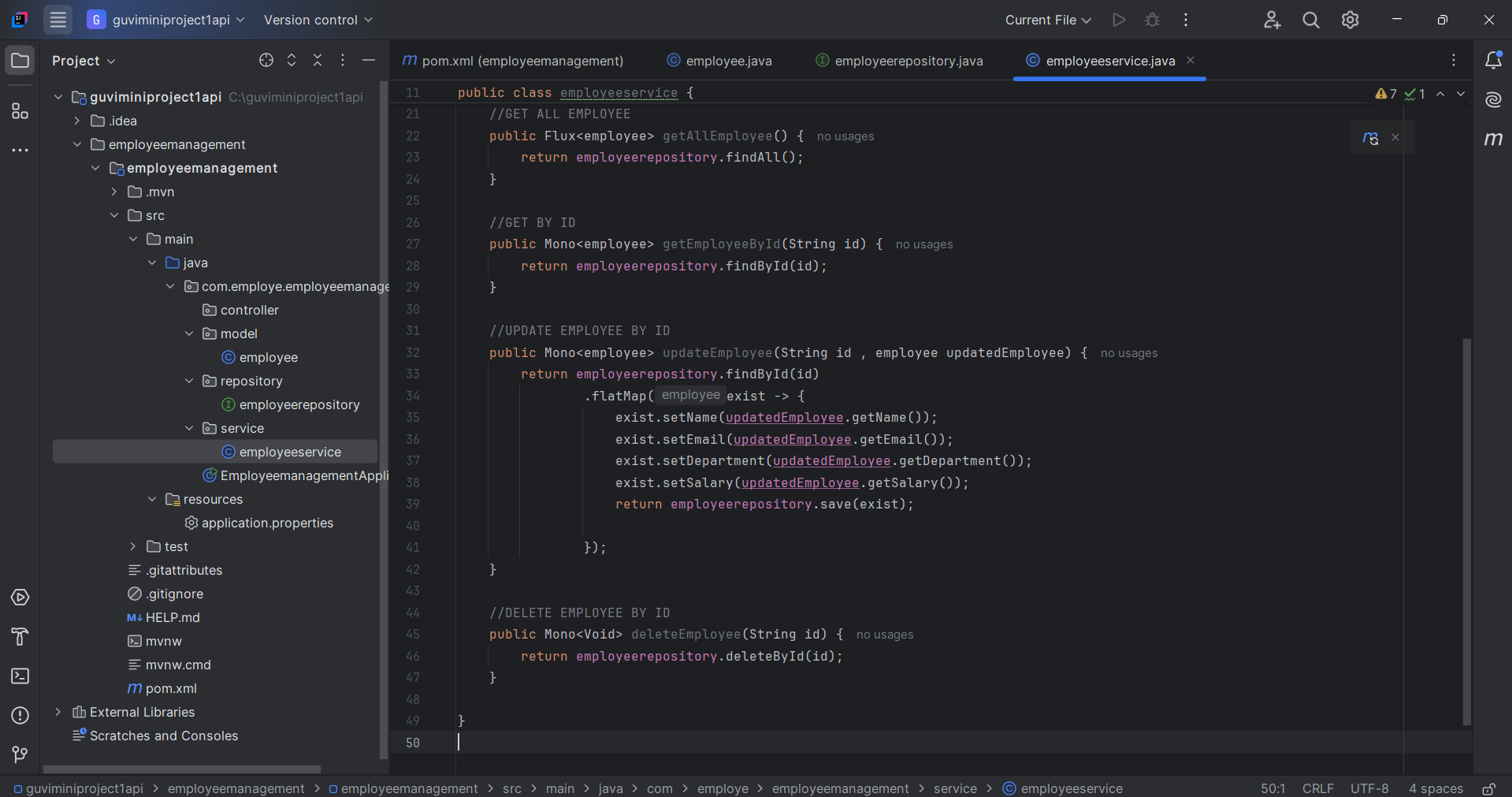
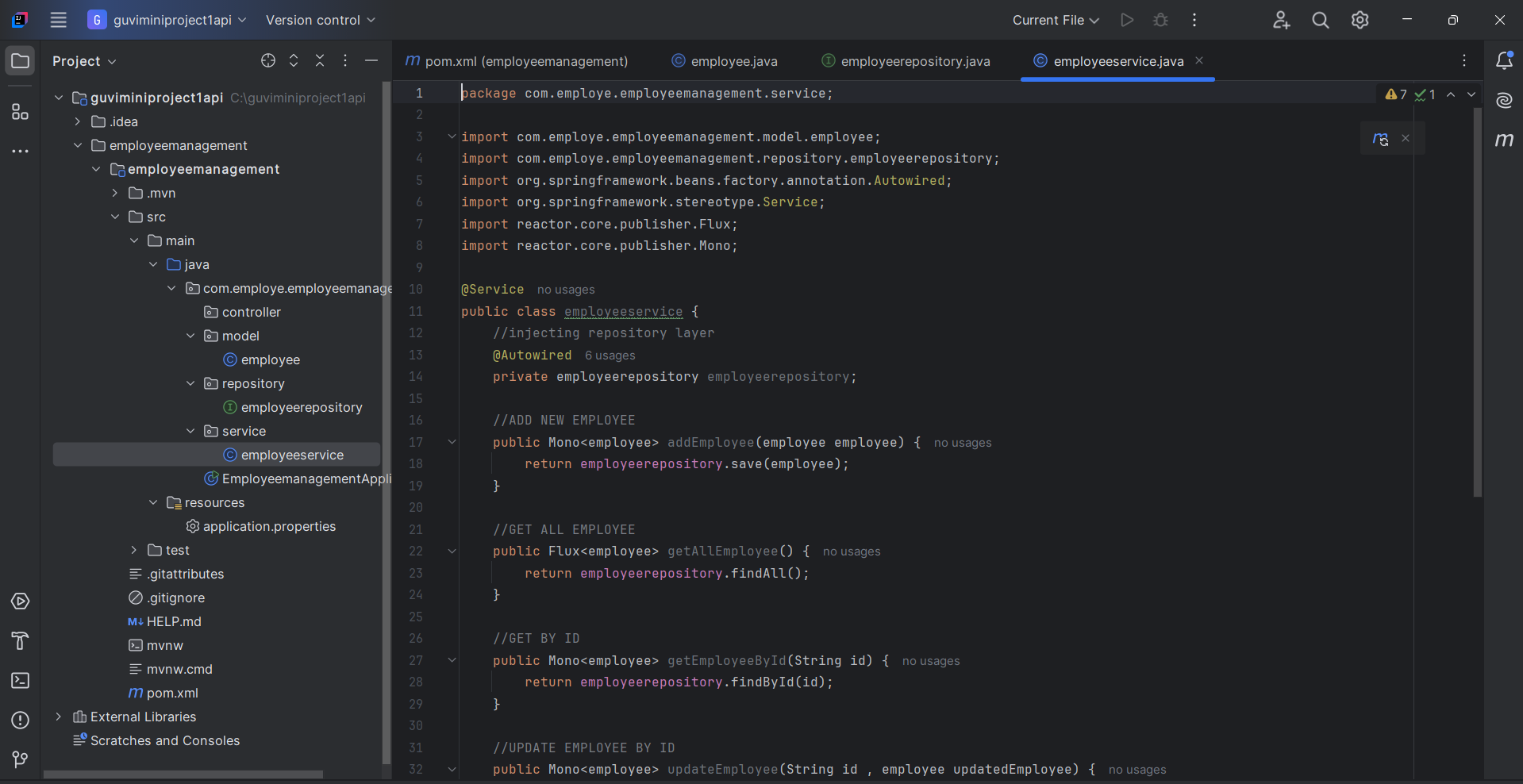
* We extend **ReactiveMongoRepository**<employee, String>
  + employee → the model NAME
  + String → the Id type
* Spring Data automatically creates MongoDB queries.

**SERVICE PACKAGE :**

1. **CLASS SERVICE**

We’ll use **Mono** and **Flux** from **Project Reactor**, since we’re using **WebFlux**.

This layer contains the **business logic**:  
 → Add employee  
 → Get all employees  
 → Update employee  
 → Delete employee  
 → Get employee by ID

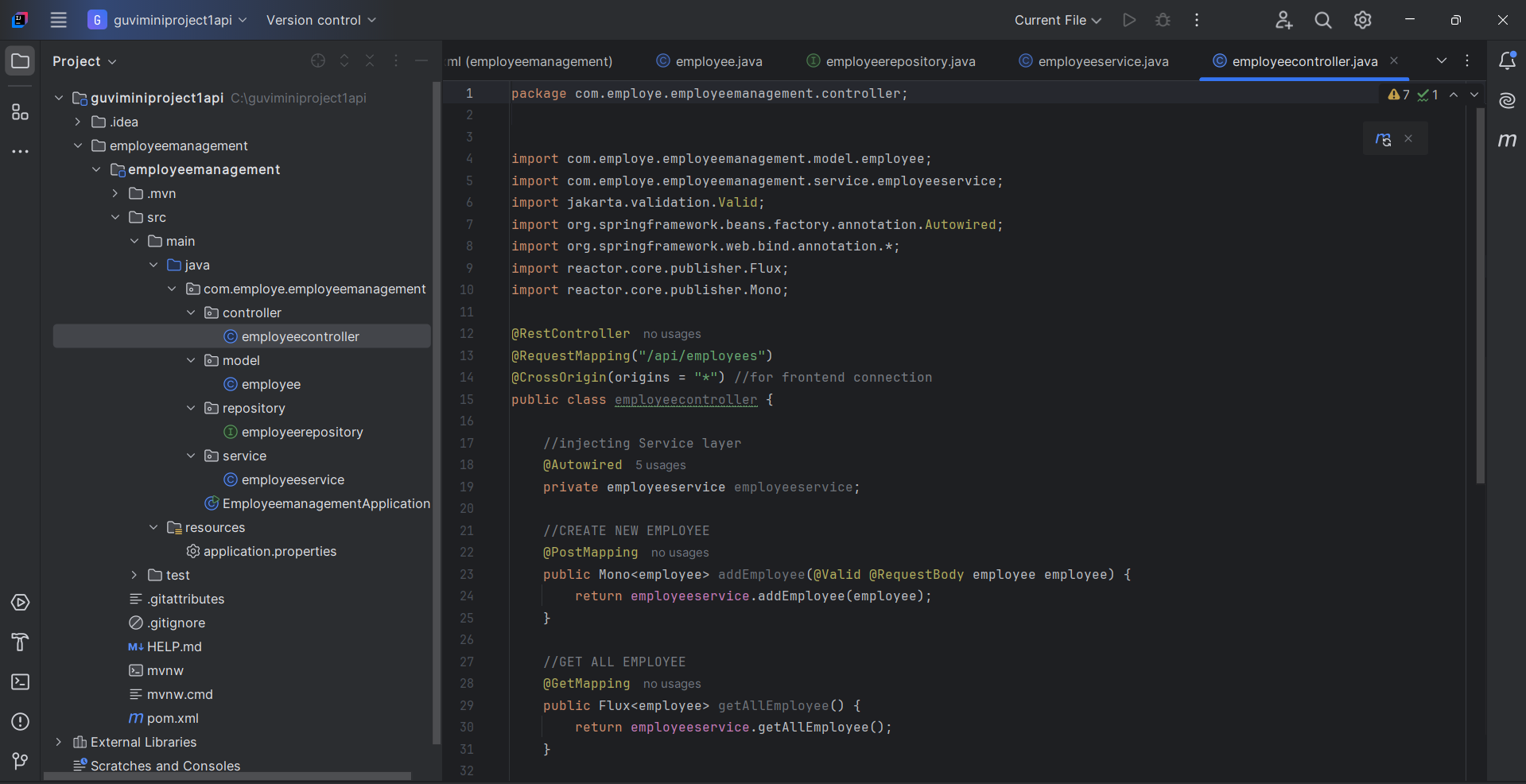


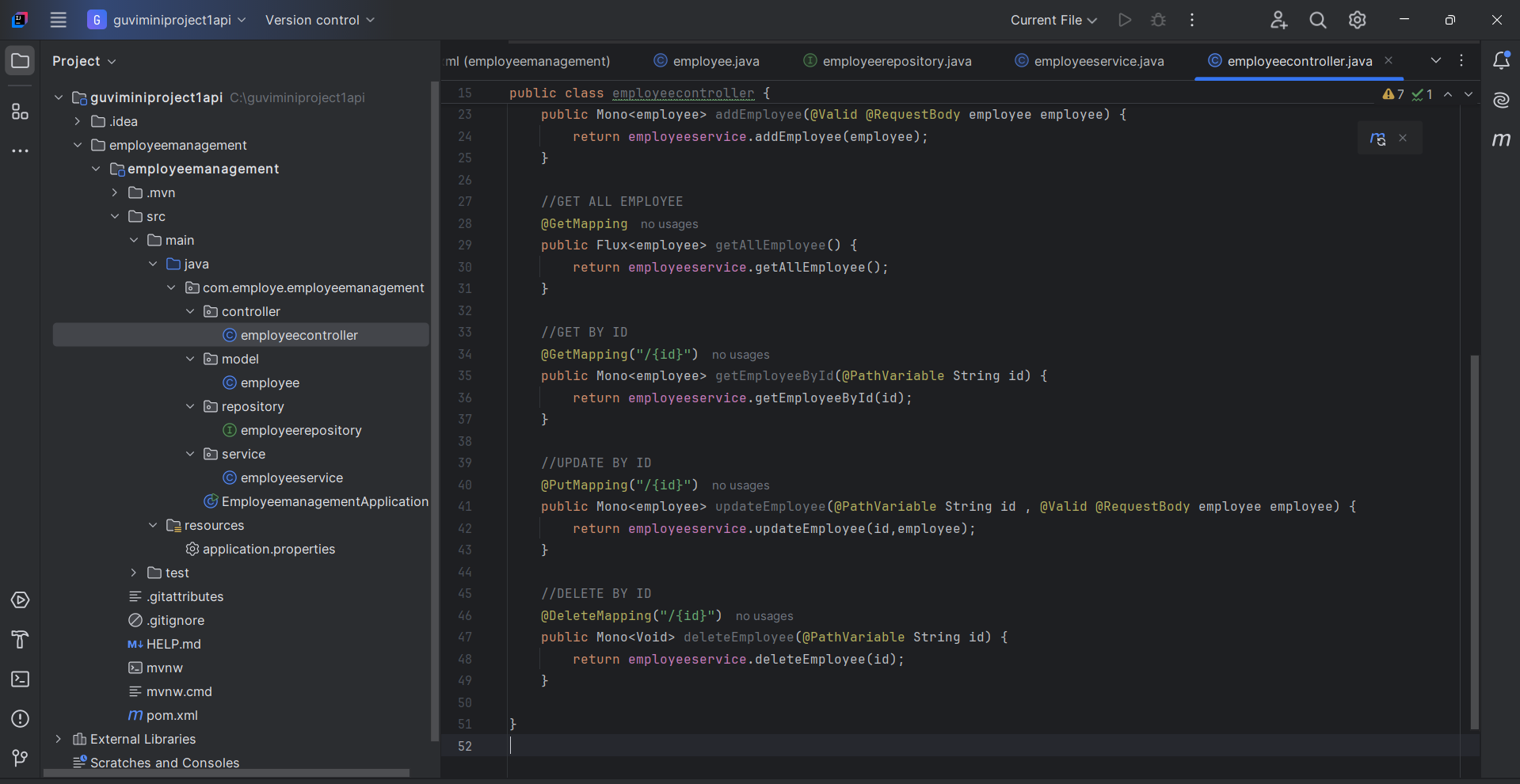
**Explanation**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Method** | **Purpose** | | --- | --- | | Mono<Employee> | Mono means single value (used for single employee) | | Flux<Employee> | Flux means stream of multiple values (used for list of employees) | | save() | Save to MongoDB | | findById() | Find one By Id | | findAll() | Get all | | deleteById() | Delete by Id | |  |  | |

**CONTROLLER PACKAGE :**

1. CLASS EMPLOYEECONTROLLER





**API ENDPOINTS SUMMARY**

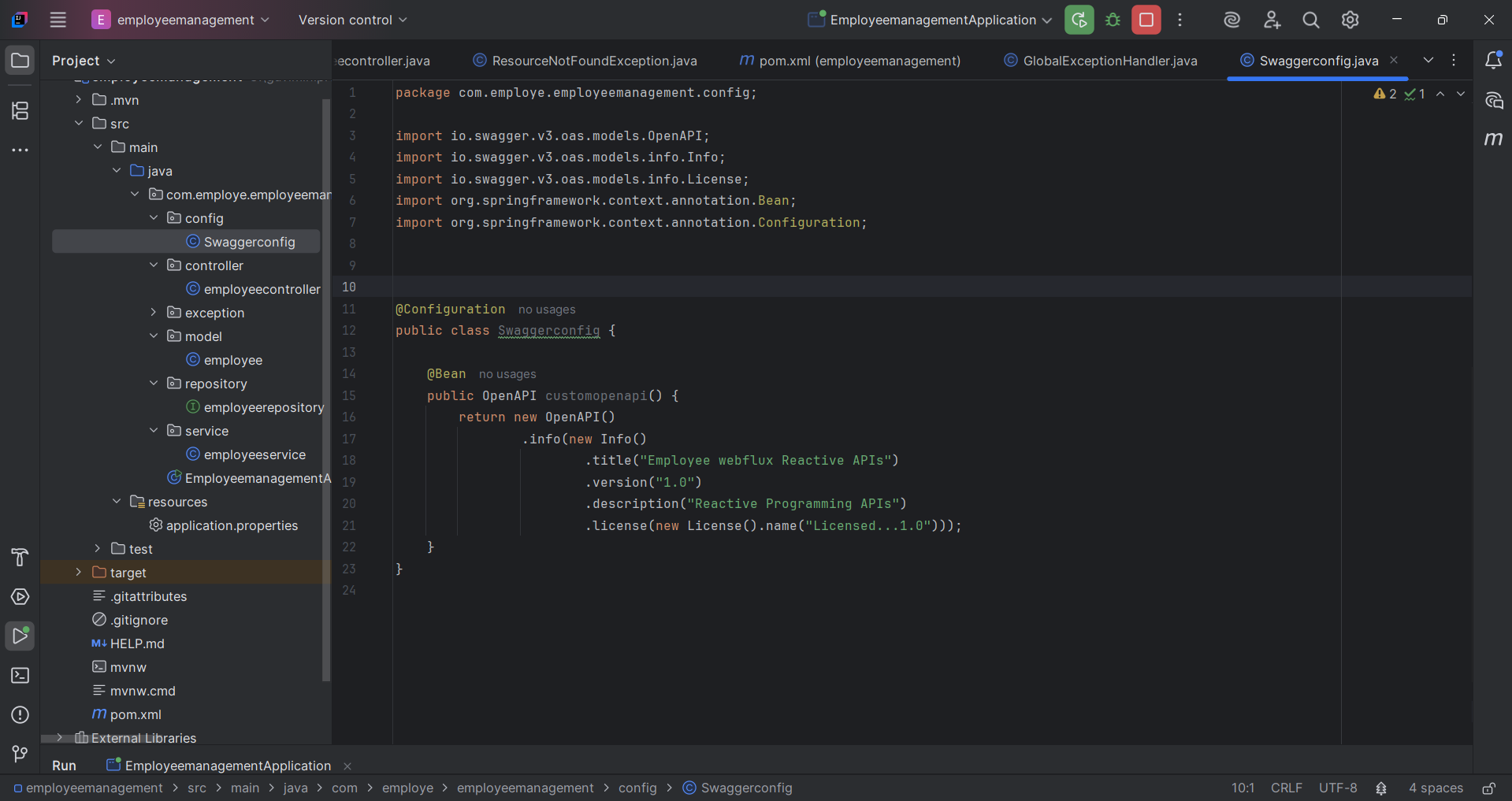
|  |  |  |  |
| --- | --- | --- | --- |
| Operation | HTTP Method | API Path | What it Does |
| Add Employee | POST | /api/employees | Adds new employee |
| Get All | GET | /api/employees | Gets all employees |
| Get by ID | GET | /api/employees/{id} | Get one employee |
| Update | PUT | /api/employees/{id} | Update by ID |
| Delete | DELETE | /api/employees/{id} | Delete by ID |

**Annotations Used:**

|  |  |
| --- | --- |
| Annotation | Meaning (English) |
| @RestController | Marks this class as a REST API controller. |
| @RequestMapping("/api/employees") | All API URLs will start with /api/employees. |
| @CrossOrigin(origins = "\*") | Allows requests from any frontend (CORS). |
| @Autowired | Spring will inject (auto-wire) the service here. |
| @PostMapping, @GetMapping, etc. | Define API methods for POST, GET, PUT, DELETE. |

**CONFIG PACKAGE**

**SwaggerConfiguration class:**

****

## Purpose of this File

This file configures **Swagger/OpenAPI** for your Spring Boot application.  
Swagger helps to **automatically generate documentation** and a **UI page** to test your REST APIs easily.

@Configuration Annotation

1. @Configuration

* This tells Spring Boot that this class provides **Spring configuration**.
* Spring will scan and load this as part of the application context.

2.@Bean

public OpenAPI customopenapi() {

* This method returns a **custom OpenAPI configuration**
* @Bean means Spring will manage this object in its container.

3. return new OpenAPI()

.info(new Info()

.title("Employee webflux Reactive APIs")

.version("1.0")

.description("Reactive Programming APIs")

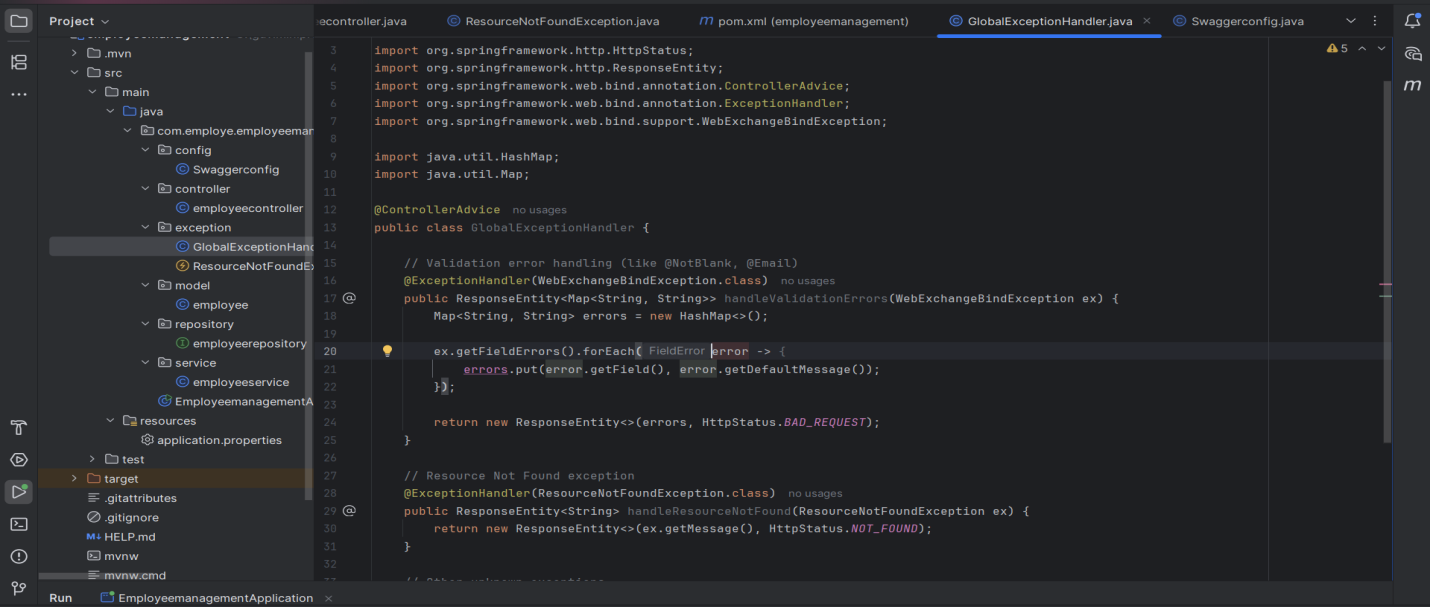
.license(new License().name("Licensed...1.0")));

|  |  |
| --- | --- |
| Method | Meaning (English) |
| title(...) | Title of the Swagger UI page |
| version(...) | Version number |
| description(...) | Small explanation of your API |
| license(...) | Any license info you want to show |

So the output will be a **Swagger UI page** like:

|  |
| --- |
| Title: Employee webflux Reactive APIs |
| Version: 1.0 |
| Description: Reactive Programming APIs |
| License: Licensed...1.0 |

EXCEPTION PACKAGE

GlobalExceptionHandling class

This code handles all types of errors (validation, not found, unknown errors) in one central place using @ControllerAdvice.

@ControllerAdvice

public class GlobalExceptionHandler {

@ControllerAdvice: Special Spring annotation to handle exceptions **globally** across the whole application.

* 1. Validation Error Handling

@ExceptionHandler(WebExchangeBindException.class)

public ResponseEntity<Map<String, String>> handleValidationErrors(WebExchangeBindException ex) {

Map<String, String> errors = new HashMap<>();

ex.getFieldErrors().forEach(error -> {

errors.put(error.getField(), error.getDefaultMessage());

});

return new ResponseEntity<>(errors, HttpStatus.BAD\_REQUEST);

}

* This method handles **backend validation errors** (like @NotBlank, @Email, @Min).
* Catches the validation errors from WebFlux (that’s why it uses WebExchangeBindException).
* For each invalid field, it creates a map:

#### Return:

* HTTP Status: 400 BAD REQUEST
* Body: Field name → Error message

**2.Resource Not Found Handling**

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<String> handleResourceNotFound(ResourceNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

* If employee ID not found in DB, we throw ResourceNotFoundException.
* This method catches it and returns a 404 error with a message

### 3.General Exception (Fallback for all other errors)

### @ExceptionHandler(Exception.class)

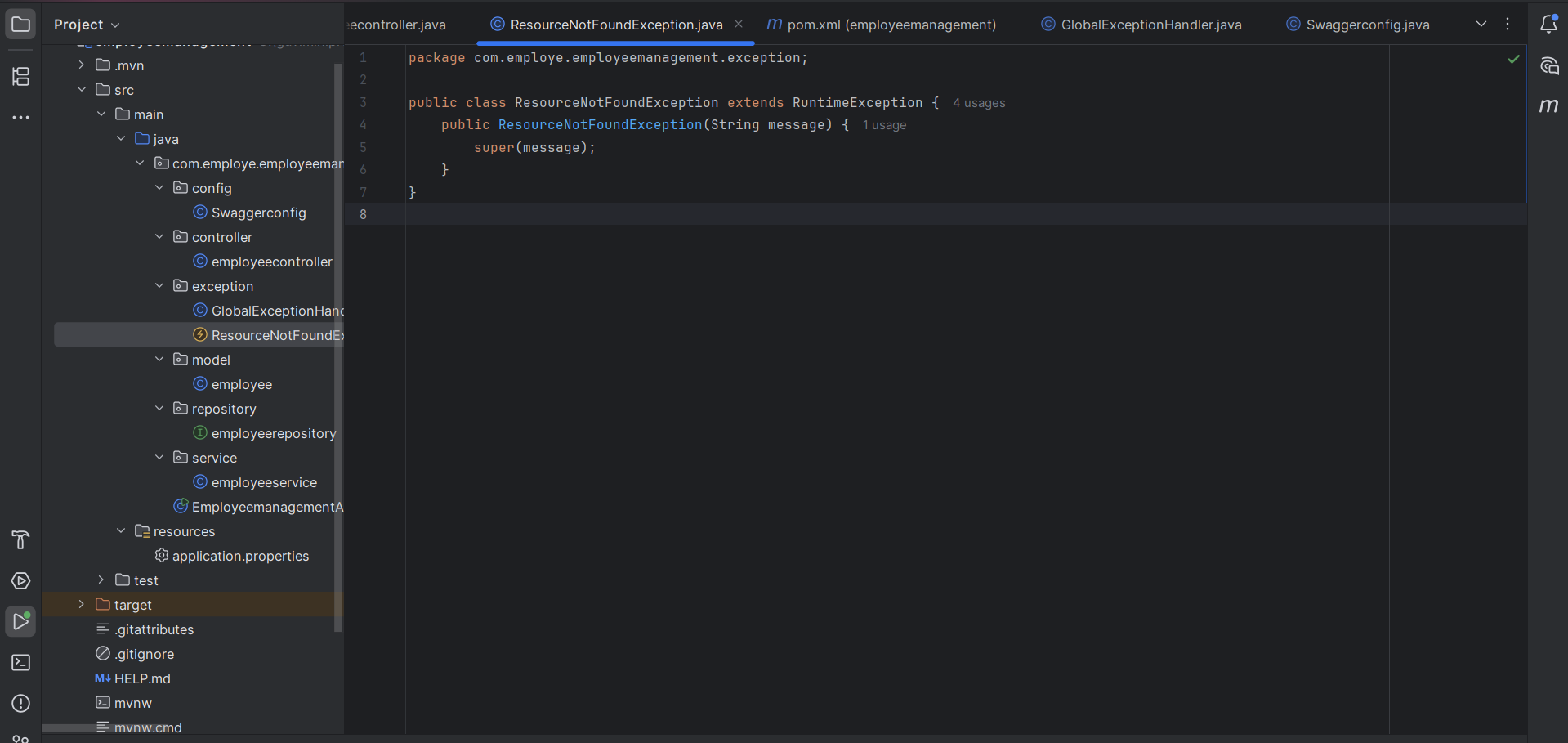
### public ResponseEntity<String> handleGeneral(Exception ex) {

### return new ResponseEntity<>("Internal Server Error: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

### }

* This is the fallback for **any unhandled exceptions**.
* Useful for debugging and preventing server crashes.
* Returns a 500 Internal Server Error.

**FILE 2 :ResourceNotFoundException.java**

****

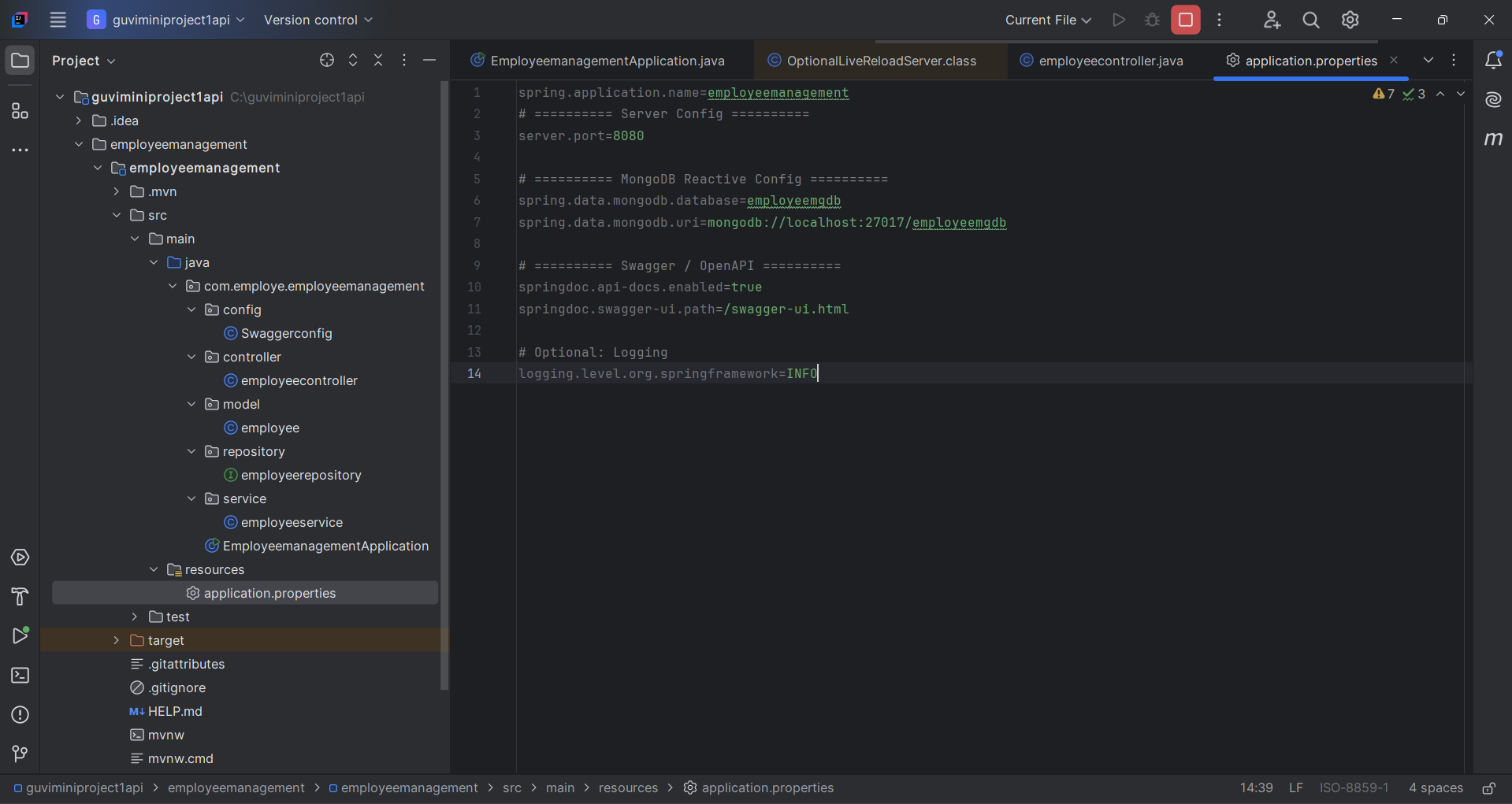
|  |
| --- |
| public class ResourceNotFoundException extends RuntimeException { |
| public ResourceNotFoundException(String message) { |
| super(message); |
| } |
| } |

* This is a **custom exception**.
* We use it to throw errors when a specific employee is not found.

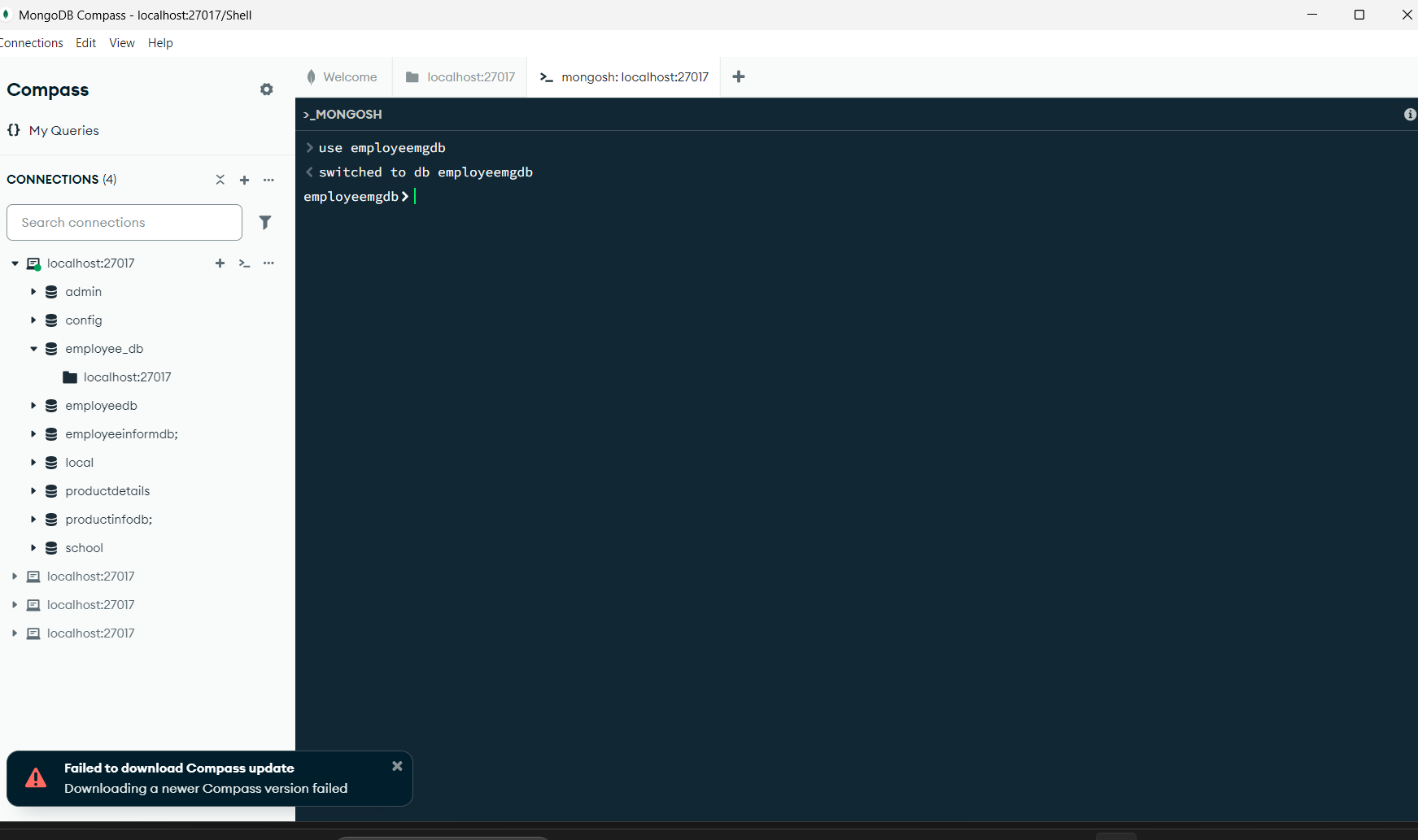
|  |  |  |  |
| --- | --- | --- | --- |
| Exception Type | Caught By | HTTP Status | Description |
| Validation errors | WebExchangeBindException | 400 (Bad Request) | Form validation miss |
| Resource not found | ResourceNotFoundException | 404 (Not Found) | IF EMPLOYEE ID NOT PRESENT |
| General/Other exceptions | Exception | 500 (Internal Server Error) | Unknown issue |

**APPLICATION PROPERTIES :**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Property** | **Meaning** | | --- | --- | | server.port=8080 | Starts your backend server on port 8080 | | spring.data.mongodb.uri | Tells Spring where your MongoDB is running | | springdoc.swagger-ui.path | Enables Swagger UI at <http://localhost:8080/swagger-ui.html> | |

****

**Now start MongoDB**



INSERT DATABASE “**employeemgdb**”

Then run your Spring Boot app and go to:  
 <http://localhost:8080/swagger-ui.html>  
You’ll see all your APIs displayed neatly.

## API Testing using Swagger (OpenAPI)

If you followed Step 5, Swagger should already be enabled.

### 🔗 Test URL:

After running the backend, open your browser and visit:

👉 [**http://localhost:8080/swagger-ui.html**](http://localhost:8080/swagger-ui.html)

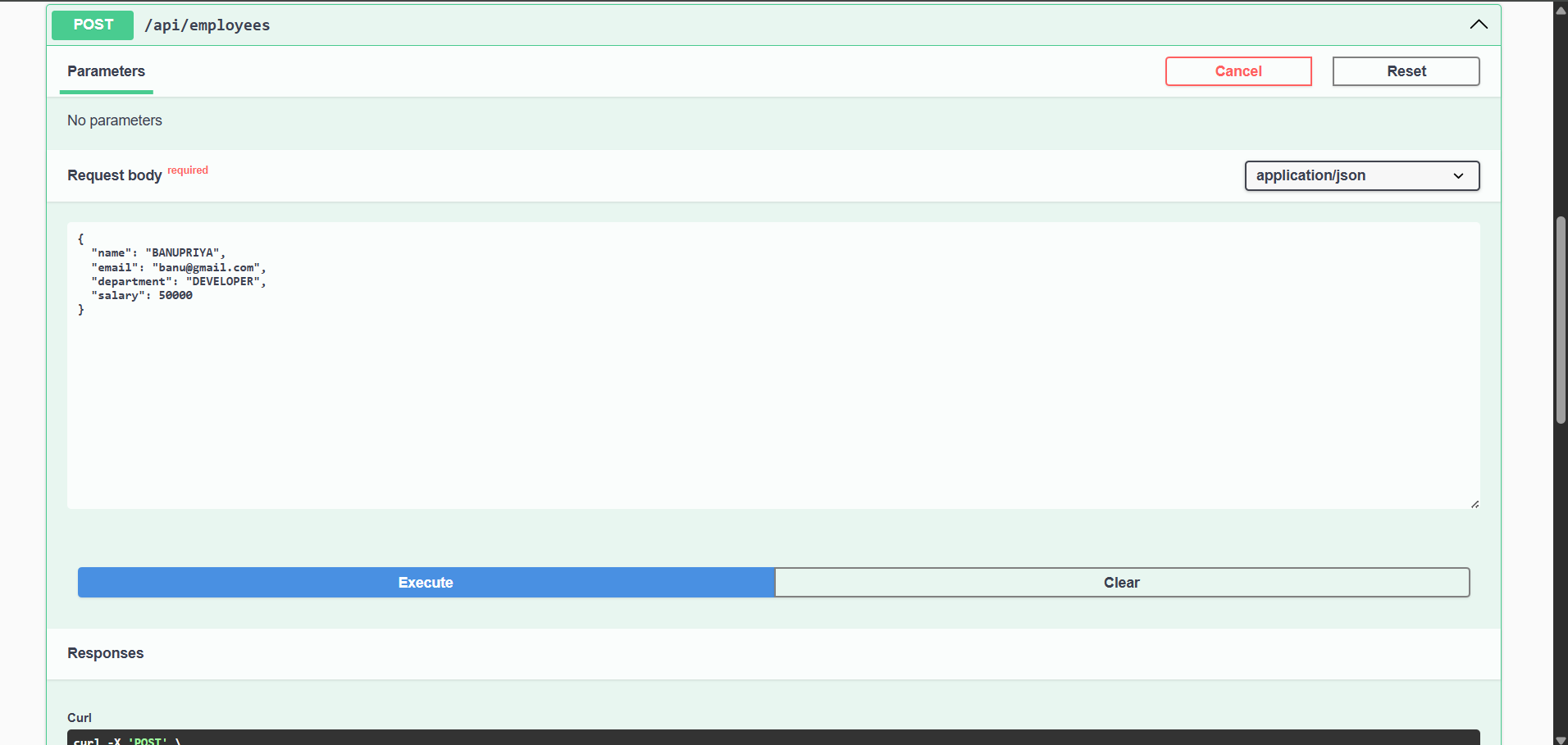
You will see:

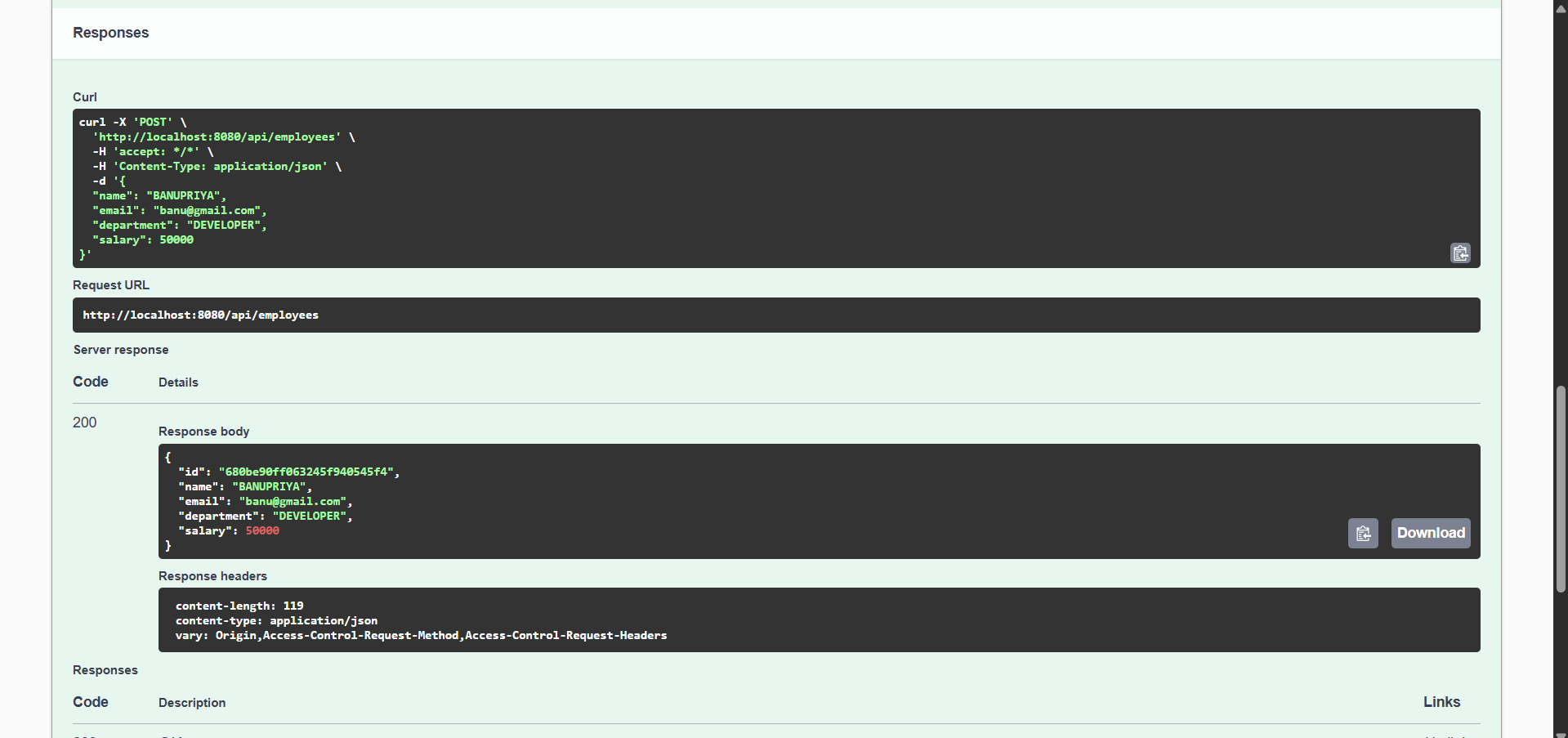
* POST /api/employees → Add Employee
* GET /api/employees → Get All
* PUT /api/employees/{id} → Update
* DELETE /api/employees/{id} → Delete
* GET /api/employees/{id} → Get One

You can click “**Try it out**”, enter request data, and see live responses ✅

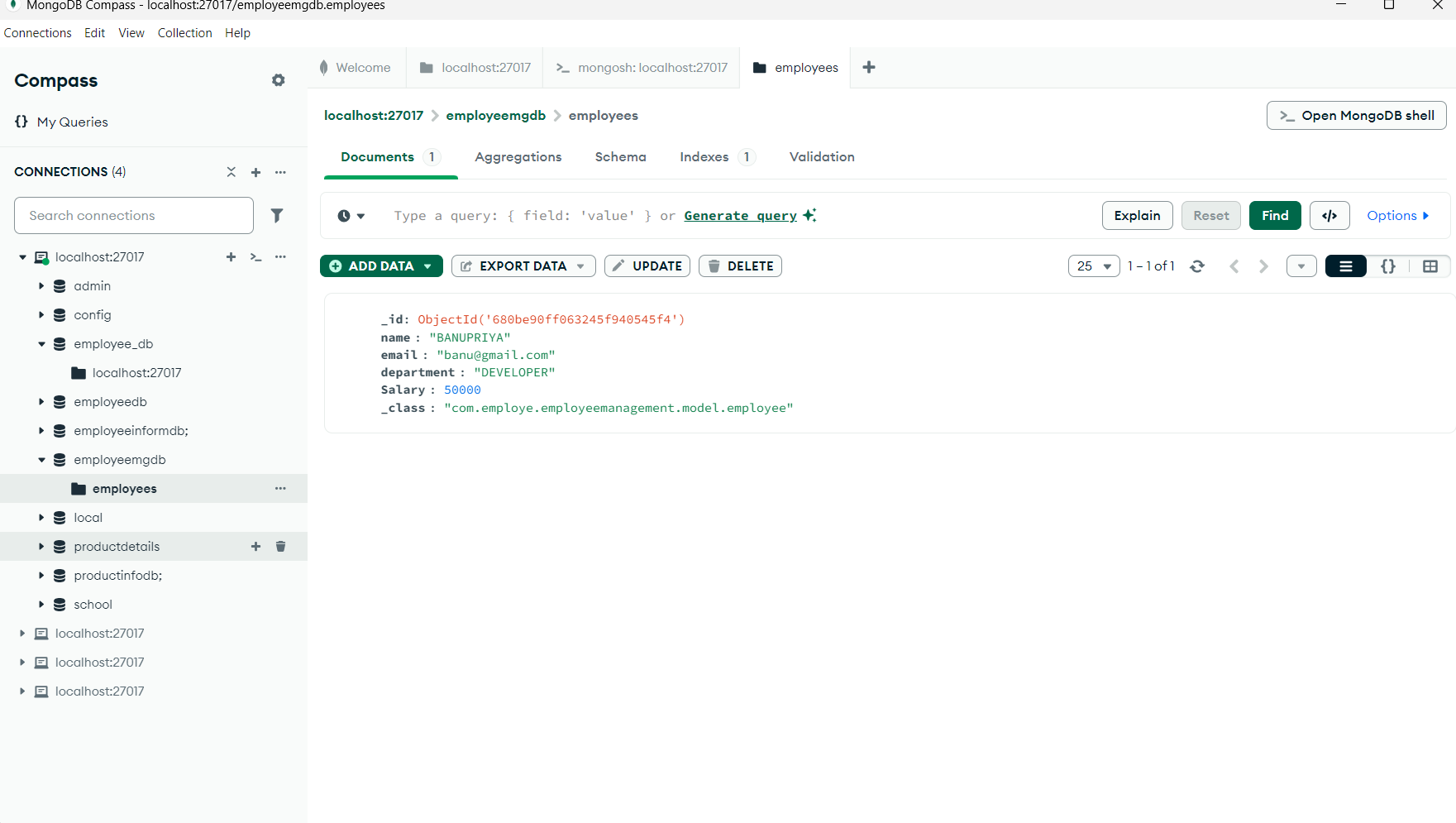


**POST**

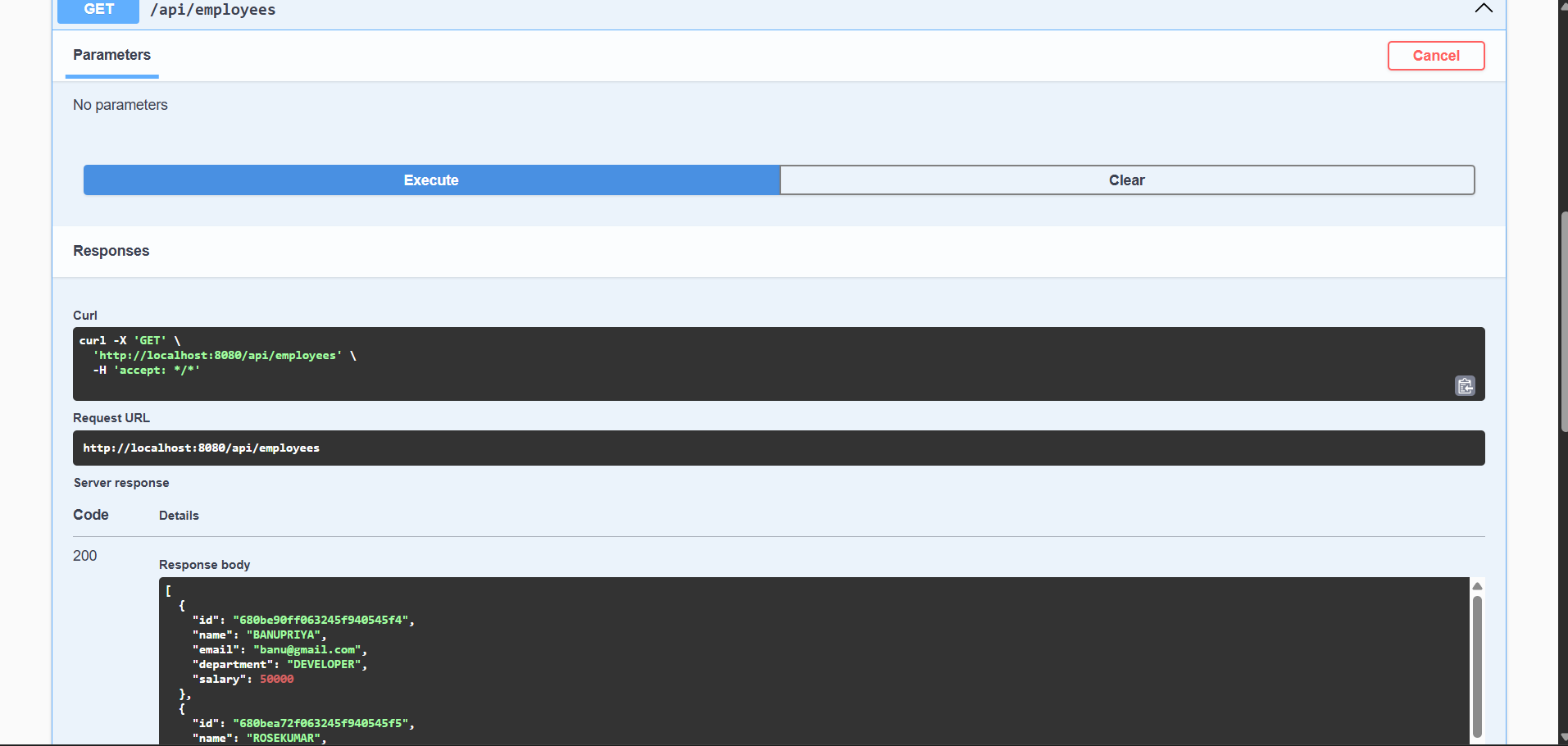
****

**RESPONSE**

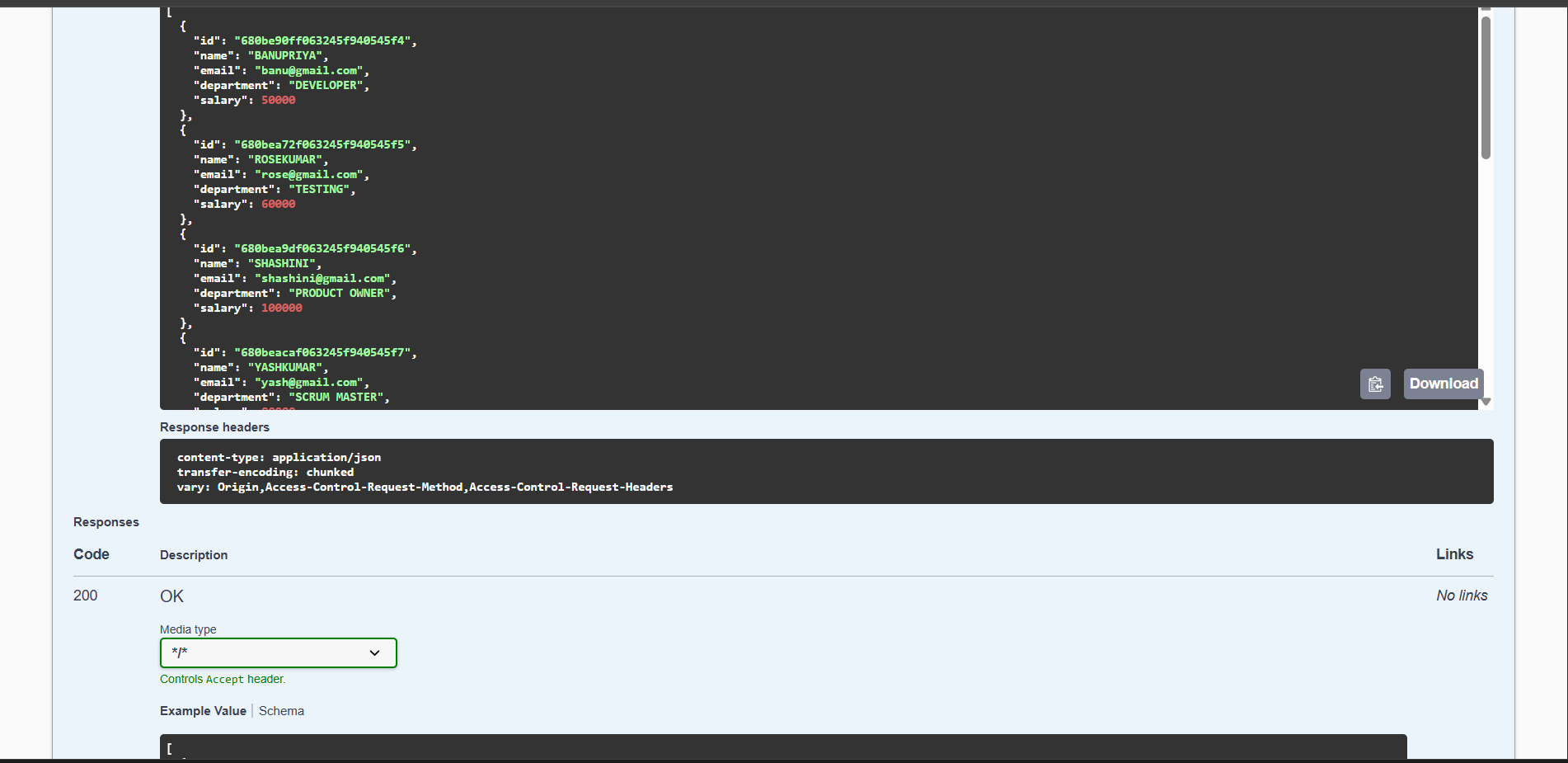
**DATABASE CONNECTION**

****

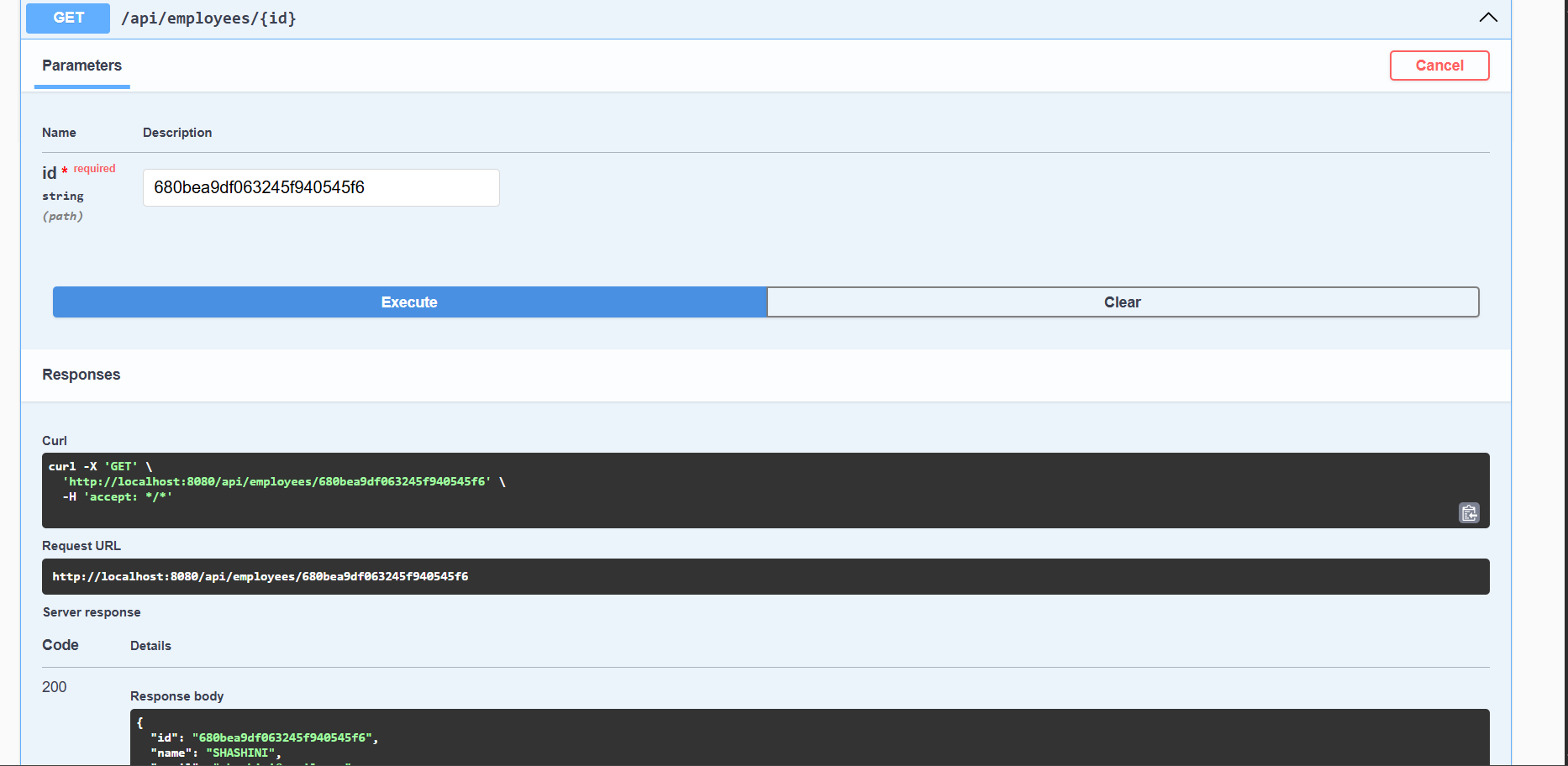
**GET**

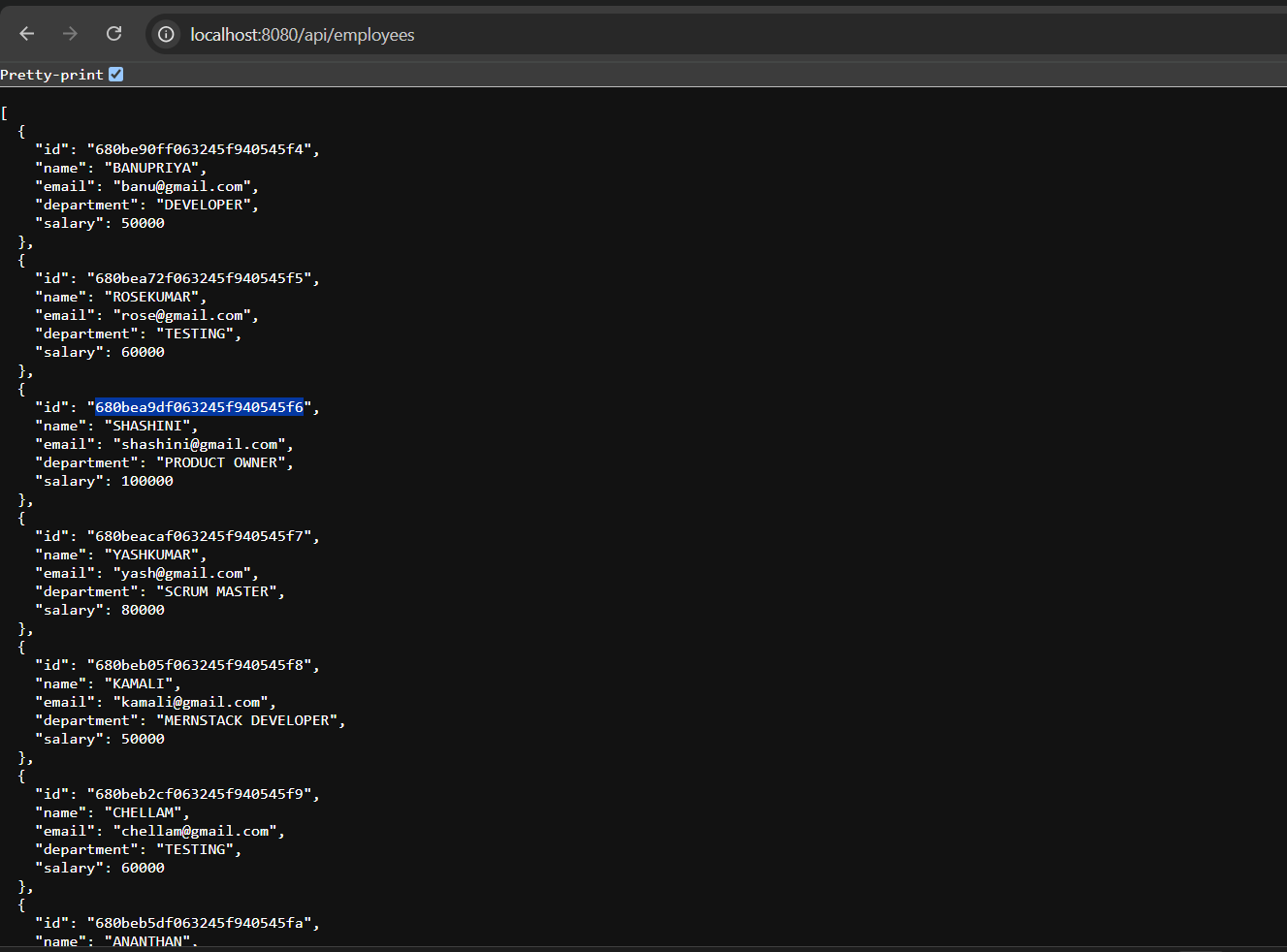
****

**RESPONSE**

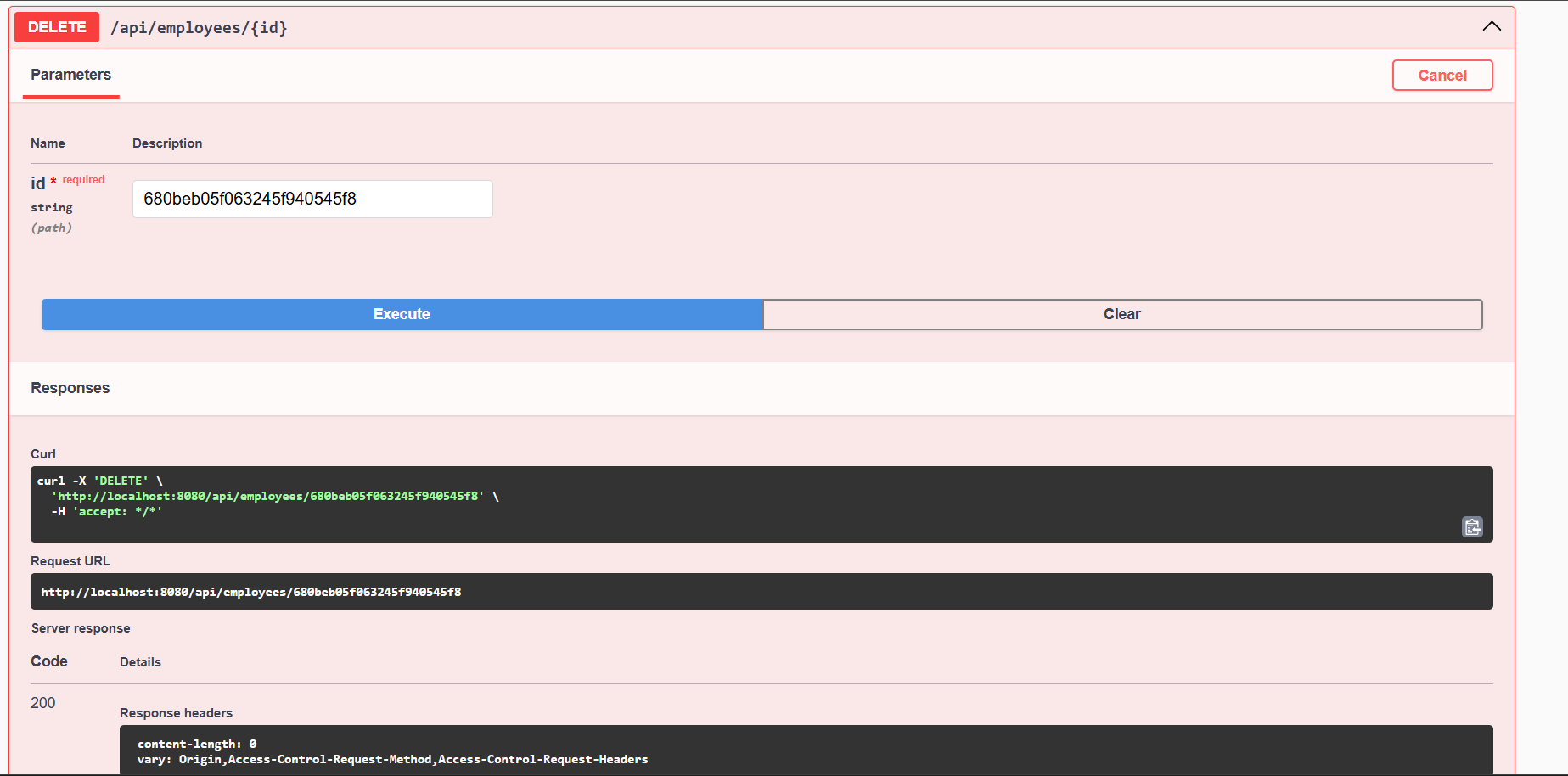
****

**GET BY ID**

****

**TOTAL EMPLOYEE LIST**

**DELETE**

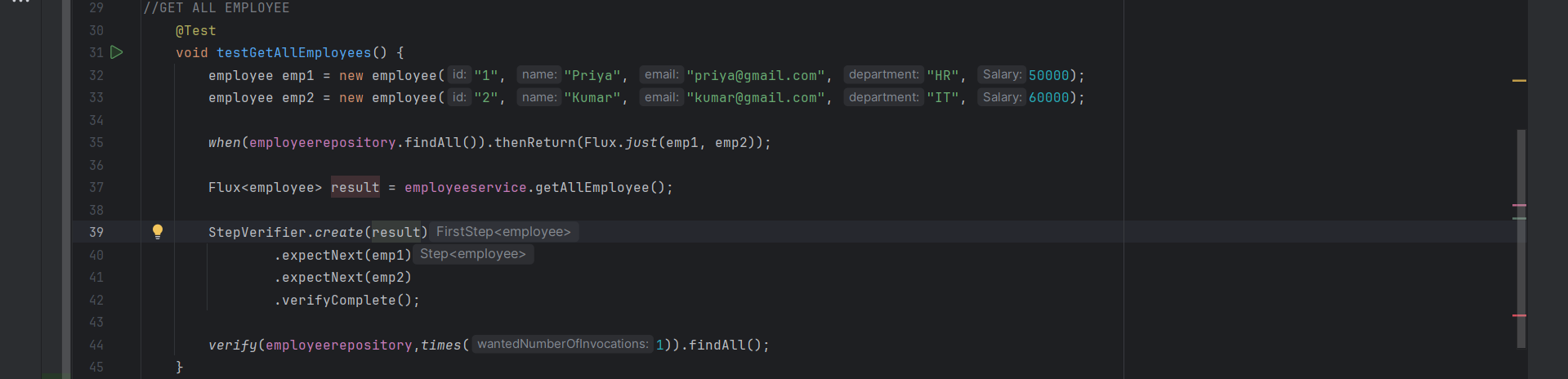
****

# First — Overall Structure of Unit Test:

**In any Unit Test, we always follow 4 steps:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Step** | **What We Do** | **Why?** | | --- | --- | --- | | 1. Mock | Fake/mock the backend/database behavior | So test will not touch real database | | 2. Action | Call the service method you want to test | Main method we are checking | | 3. Verify Data | Check the result using StepVerifier | Confirm if output is correct | | 4. Verify Behavior | Check if repository method was called properly | Confirm method flow | |

**Test 1: Get All Employees — testGetAllEmployees()**

****

**EXPLANATION :TEST 1**

|  |  |  |
| --- | --- | --- |
| Step | Code | Meaning |
| 1. Mock | when(employeerepository.findAll()).thenReturn(Flux.just(emp1, emp2)); | If someone calls findAll, return two employees (Priya and Rose) |
| 2. Action | Flux<employee> result = employeeservice.getAllEmployees(); | Call the service method to get all employees |
| 3. Verify Data | StepVerifier.create(result).expectNext(emp1).expectNext(emp2).verifyComplete(); | Check that first Priya, then Rose are returned |
| 4. Verify Behavior | verify(employeerepository, times(1)).findAll(); | Confirm that findAll method was called exactly once |

**Test 2: CREATE NEW EMPLOYEE — testCreateEmployee();**

****

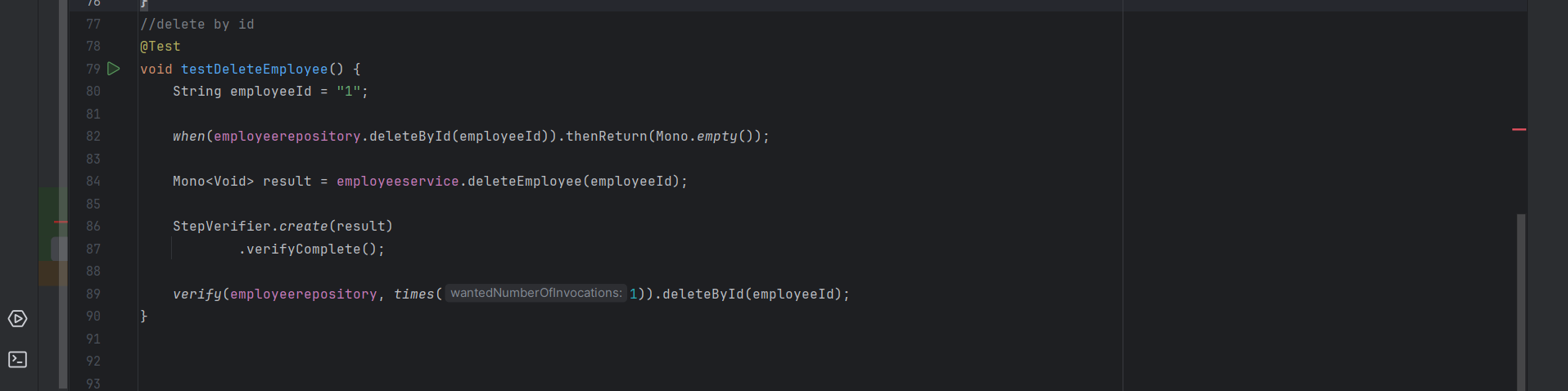
|  |  |  |
| --- | --- | --- |
| Step | Code | Meaning |
| 1. Mock | when(employeerepository.save(employee)).thenReturn(Mono.just(employee)); | If someone saves an employee, return the same employee (fake) |
| 2. Action | Mono<employee> result = employeeservice.createEmployee(employee); | Call service method to save an employee |
| 3. Verify Data | StepVerifier.create(result).expectNext(employee).verifyComplete(); | Check if the result gives the same employee correctly |
| 4. Verify Behavior | verify(employeerepository, times(1)).save(employee); | Confirm that the save method was called exactly once |

**Test 3: GET THE ELEMENT BY ID — testGetEmployeeById()**

****

|  |  |  |
| --- | --- | --- |
| Step | Code | Meaning |
| 1. Mock | when(employeerepository.findById(employeeId)).thenReturn(Mono.just(employee)); | If someone searches for ID 1 give employee priya |
| 2. Action | Mono<employee> result = employeeservice.getEmployeeById(employeeId); | Call service method to get employee by ID |
| 3. Verify Data | StepVerifier.create(result).expectNext(employee).verifyComplete(); | Check if the correct employee is returned |
| 4. Verify Behavior | verify(employeerepository, times(1)).findById(employeeId); | Confirm that the findById method was called exactly once |

**Test 3: DELETE EMPLOYEE BY ID — testDeleteEmployee()**

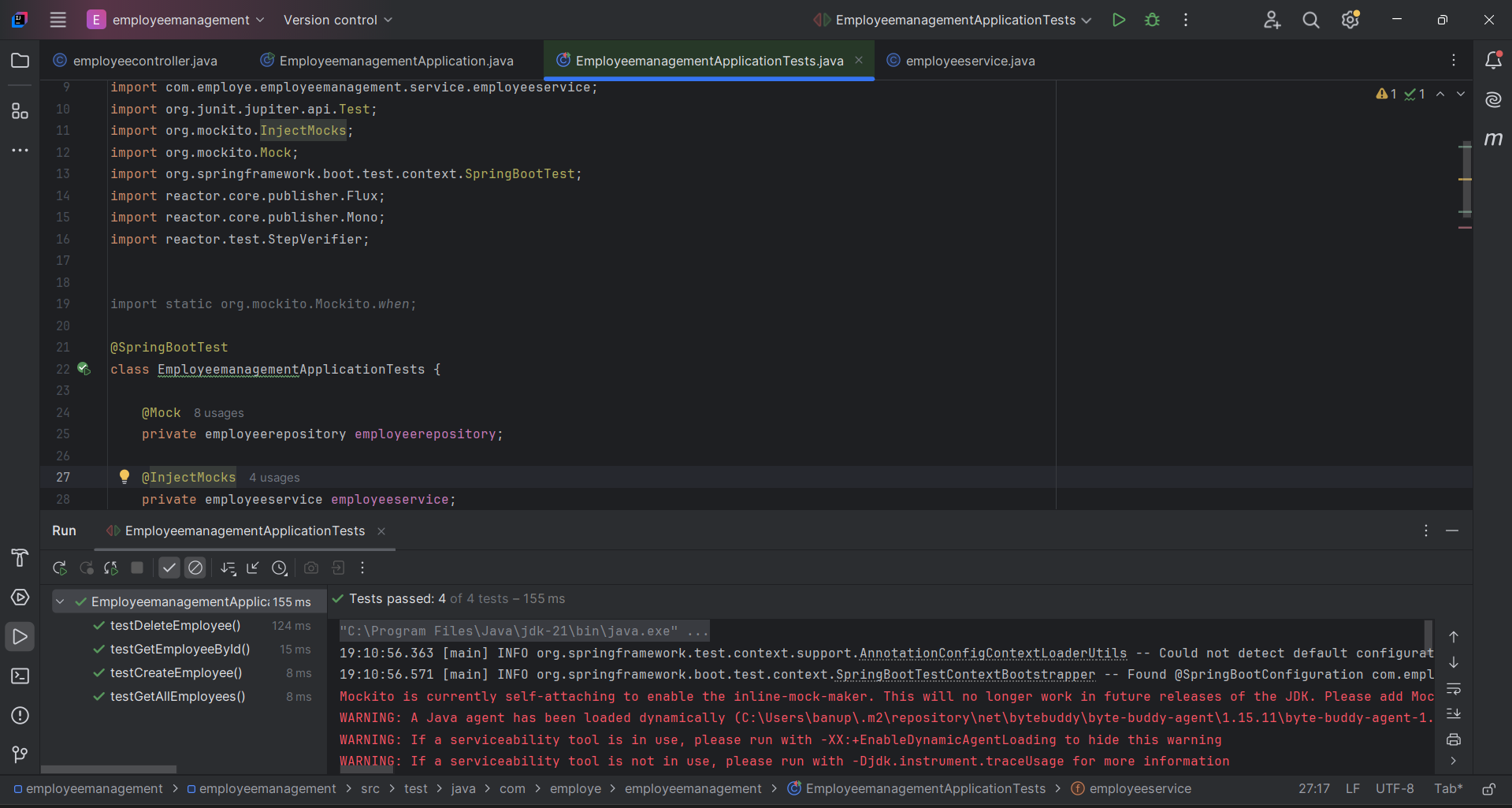
****

|  |  |  |
| --- | --- | --- |
| Step | Code | Meaning |
| 1. Mock | when(employeerepository.deleteById(employeeId)).thenReturn(Mono.just(employee)); | If someone delete for ID 1 ,just return empty(nothing) |
| 2. Action | Mono<void> result = employeeservice.deleteEmployee(employeeId); | Call service method to delete employee by ID |
| 3. Verify Data | StepVerifier.create(result).verifyComplete(); | No data should return –just complete. |
| 4. Verify Behavior | verify(employeerepository, times(1)).deleteById(employeeId); | Confirm that the deleteById method was called exactly once |

**Important Concepts**

|  |  |
| --- | --- |
| Concept | Simple Meaning |
| Mono | Single value async (like promise) |
| Flux | Multiple values async (like list of promises) |
| Mockito when() | Create fake behaviors |
| Mockito verify() | Confirm correct methods were called |
| StepVerifier | Testing reactive flows |

**TEST RESULTS**

****

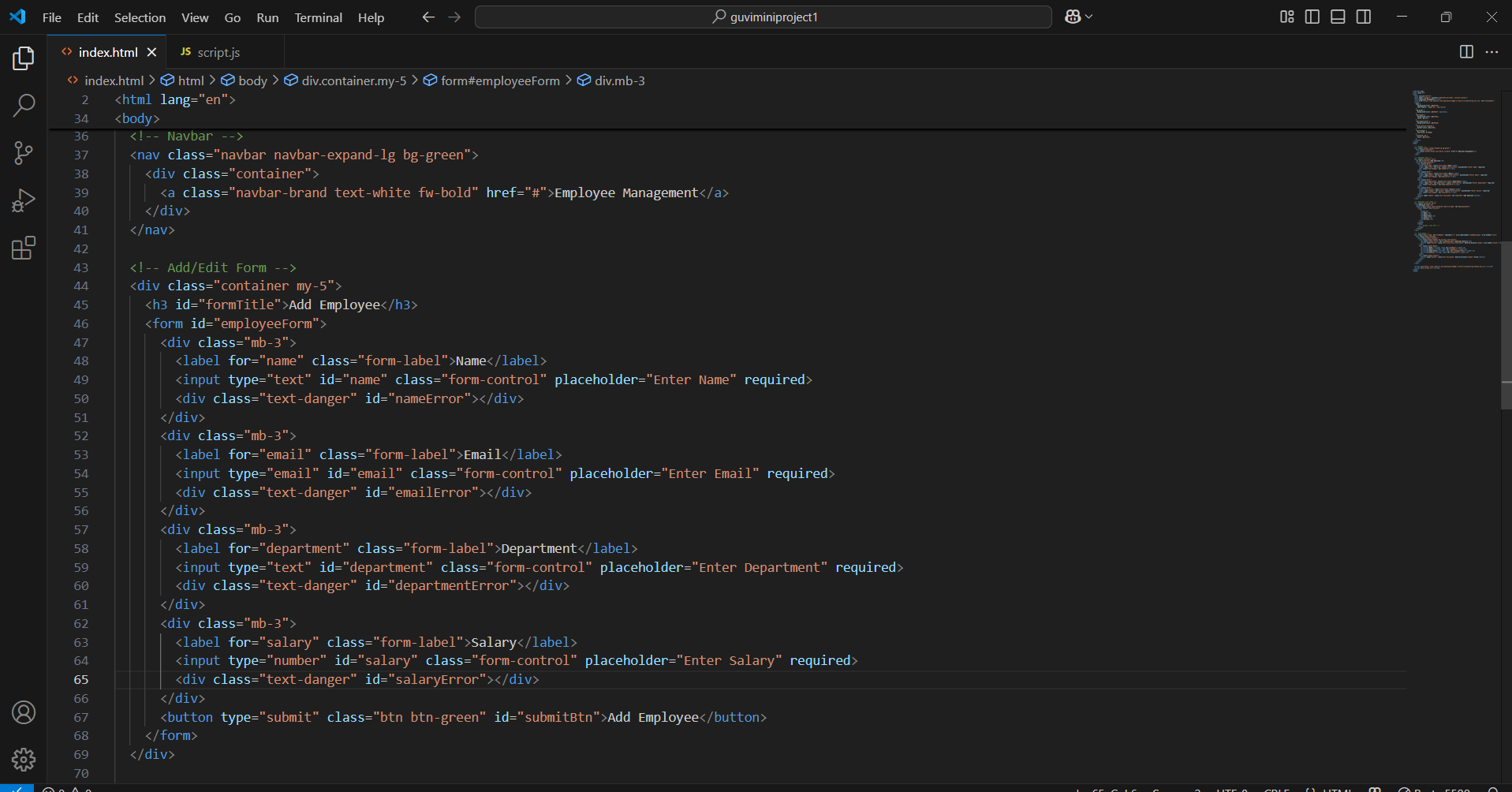
**All the test cases are passed**

**FRONTEND DEVELOPMENT**

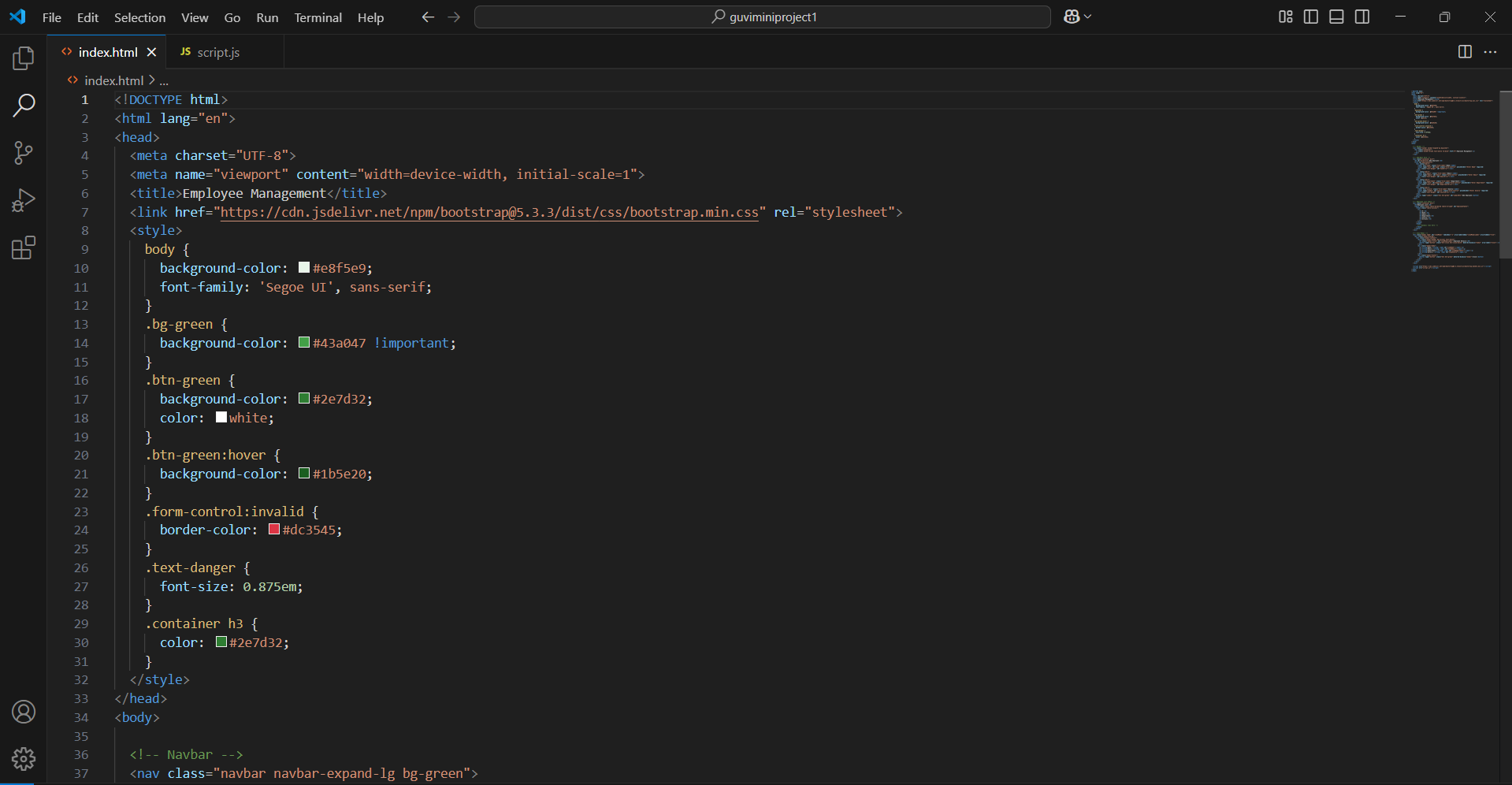
**HTML,CSS,JAVASCRIPT ALSO USING BOOTSTRAP**

In it we using fetch method to fetch the data from the database

HTML FILE

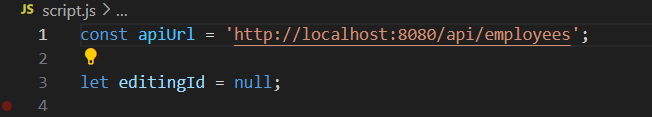


IN LINE STYLE WITH BOOTSTRAP



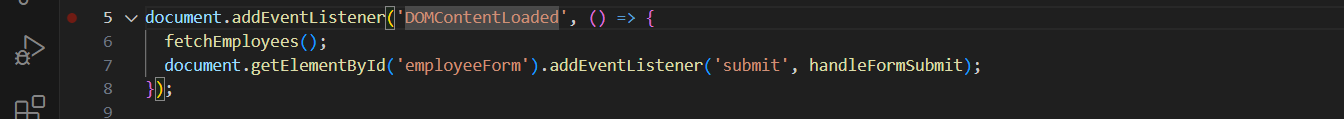
JAVASCRIPT FILE

**1.Global Declaration**



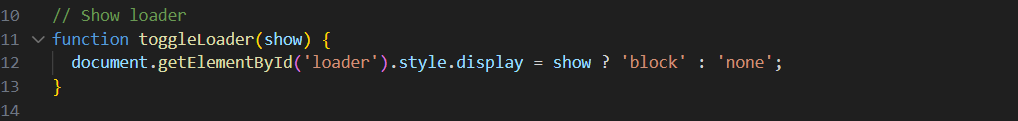
* apiUrl: The base URL to call your Spring Boot backend API.
* editingId: Tracks the current employee being edited. It's null if you're adding a new employee.

**2.DOMContentLoaded Event**

****

* This runs after the HTML page is fully loaded.
* Calls fetchEmployees() to load employee data from the backend.
* Adds a submit event listener to the form to handle add/update.

**3.Loader Toggle**



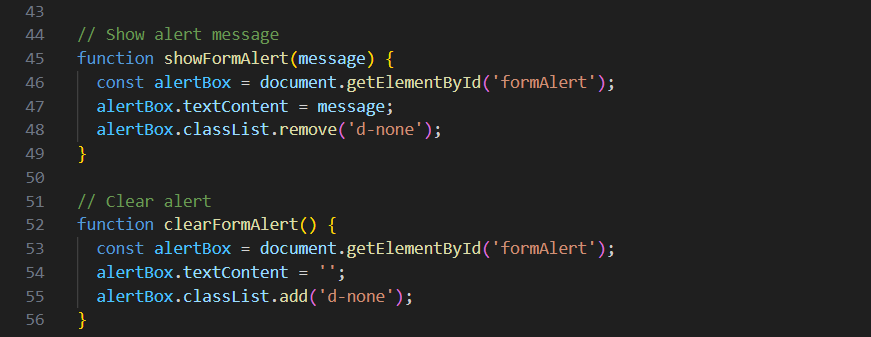
* Shows or hides a loading spinner based on the show flag (true or false).

4.Fetch All Employees

* Fetches all employees from the backend API.
* Builds and displays the employee table dynamically.
* Shows a loader during fetch and handles errors gracefully.



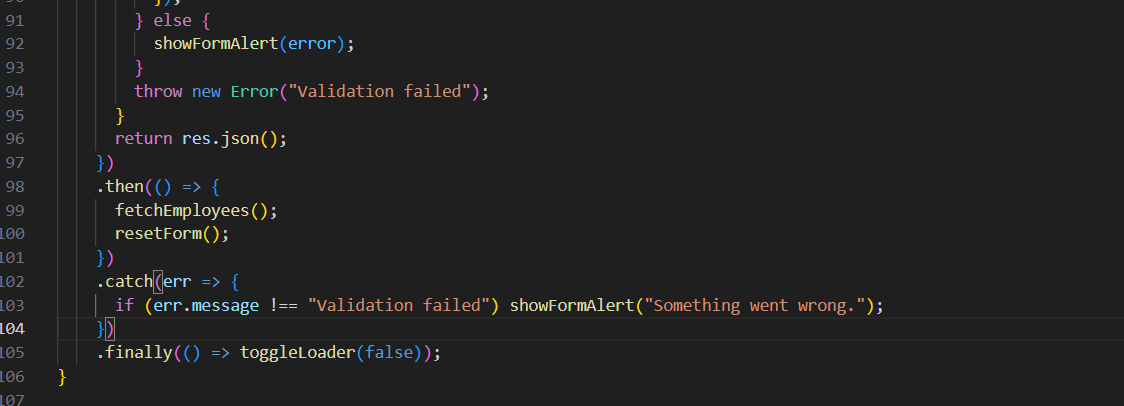
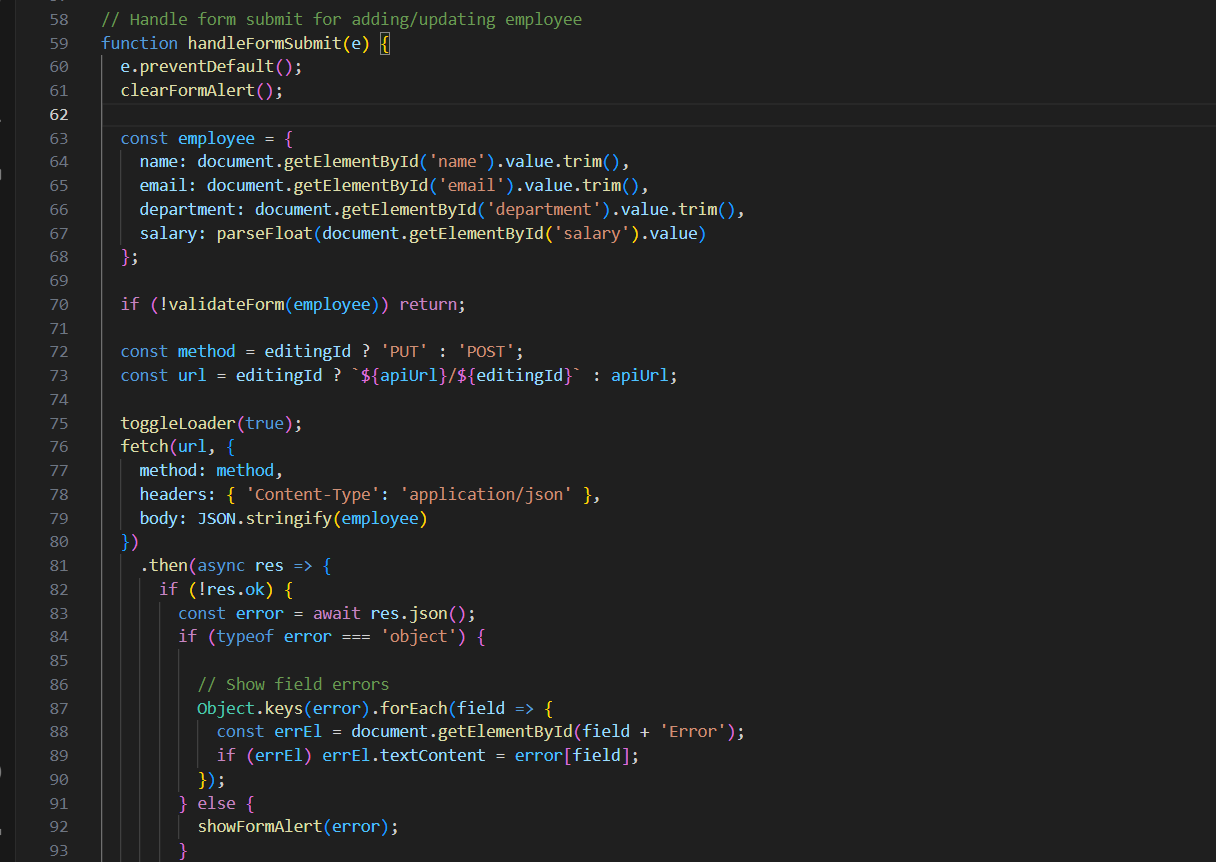
5.Form Alert Display



* showFormAlert(msg): Shows error messages in an alert box.
* clearFormAlert(): Hides the alert box and clears text.

6. Handle Add/Update Form Submit

* Prevents the page from reloading (e.preventDefault()).
* Builds the employee object from form inputs.
* Validates form (frontend).
* Sends POST (new) or PUT (update) request to backend.
* Shows validation errors from backend if any.
* On success, reloads the employee table and resets the form.

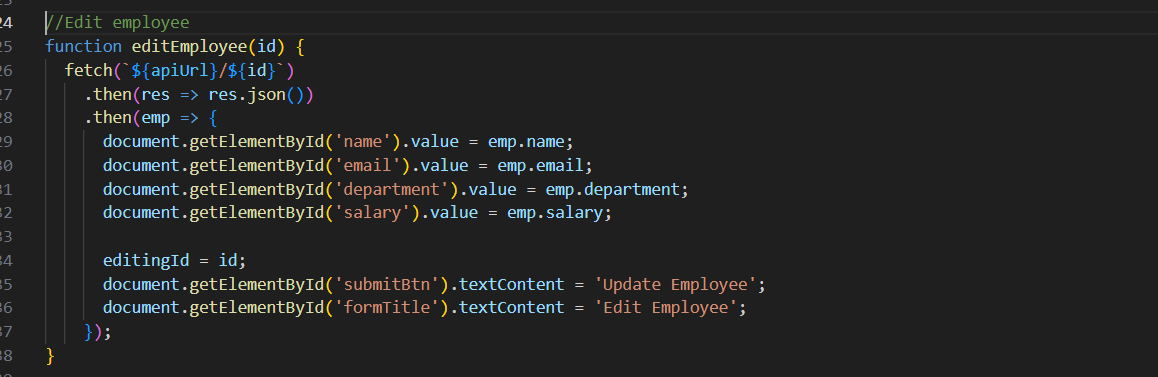


**7.Frontend Form Validation**

* Checks if all fields are filled properly.
* Checks if email format is valid.
* Checks if salary is a positive number.
* Displays error messages near respective form fields.

****

8. Edit Employee



* Fills the form with the employee's data for editing.
* Changes the form title and button to “Update Employee”.

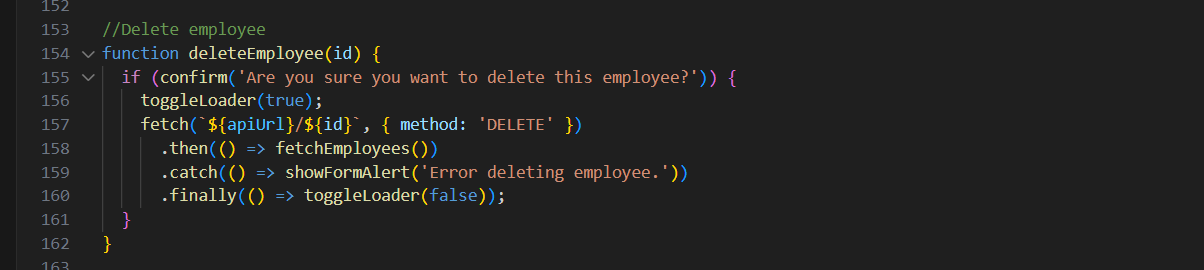
**8.Reset Form**



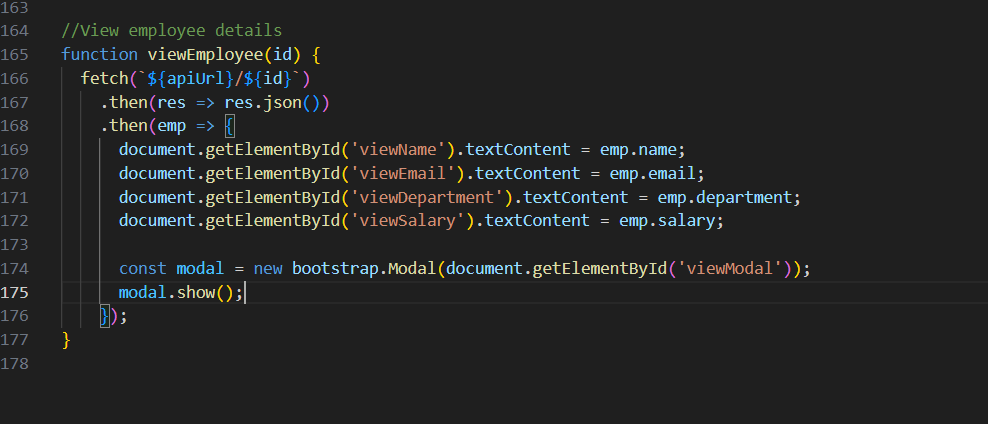
* Clears the form after adding or updating an employee.
* Resets button and title to "Add Employee".
* Clears validation messages and alert box.

10. Delete Employee

* Asks confirmation before deleting
* Sends DELETE request to backend.
* Refreshes the employee list.

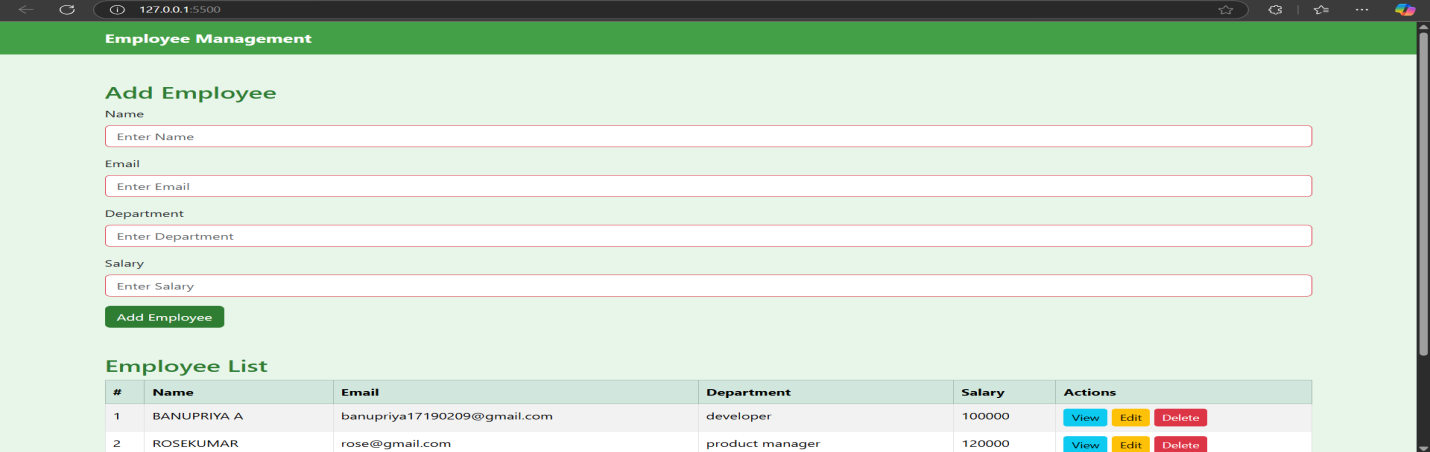


11. View Employee Details (Modal)



* Gets full employee details from backend by ID.
* Fills and opens a Bootstrap modal to display them.

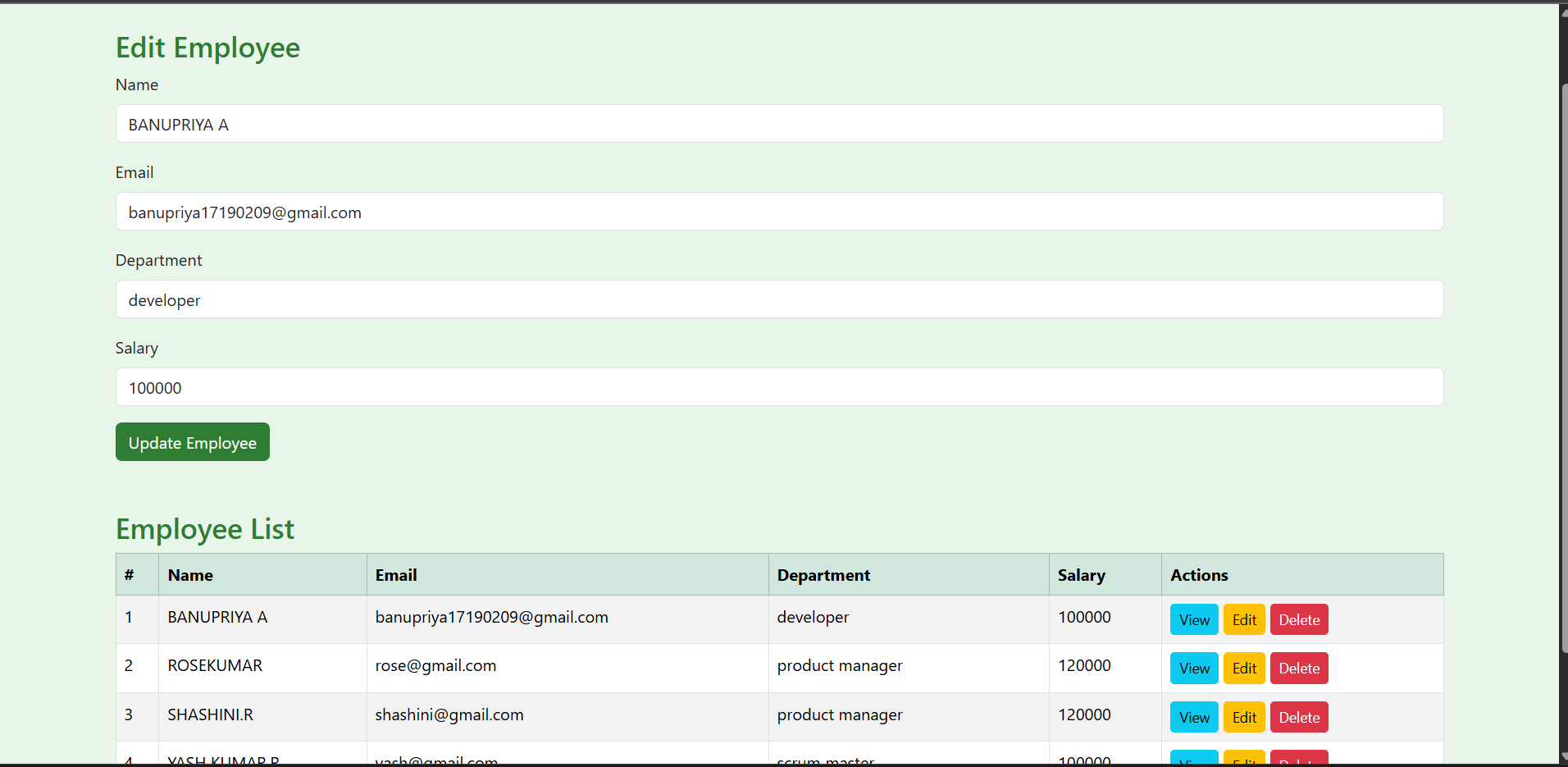
MY FRONTEND VIEW



VIEW EMPLOYEE



EDIT EMPLOYEE



DELETE EMPLOYEE

