System Requirements Specification

Index

For

Department-Employee

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

Department-Employee-Microservice App
System Requirements Specification

1 Project Abstract

The **Department-Employee** Application is designed to efficiently manage employee records and departmental details within an organization. Utilizing a microservices architecture built with Spring Boot, it separates concerns into dedicated services for managing employee and department data. Each microservice independently maintains its own database and communicates seamlessly through RESTful APIs and service discovery via a Eureka Naming Server, ensuring a modular, scalable, and maintainable system.

Following is the requirement specifications:

	App
Microservices	
1	Employee Microservice
2	Department Microservice
Employee Microservice	
1	Create Employee
2	Get Employee by ID
3	Update Employee
Department Microservice	
1	Create Department
2	Get Department with Employee details by ID
3	Update Department

2 CONSTRAINTS

2.1 Department Constraints

- When fetching a department with employee details by ID, if the department ID does not exist, the service method should throw a `NotFoundException` with the message:
  "Department with id {id} not found"

- When updating a department, if the department ID does not exist, the service method should throw a `NotFoundException` with the message:
  "Department with id {id} not found"

- When fetching the associated employee from the Employee microservice, if the employee ID is not found or the request fails, the service method should throw a `NotFoundException` with the message:
  "Employee with id {employeeId} not found"

2.2 Department Constraints

- When updating an employee, if the employee ID does not exist, the service method should throw a `NotFoundException` with the message:
  "Employee with id {id} not found"

2.3 Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exception if data is invalid
- All the business validations must be implemented in dto classes only.
- All the database operations must be implemented on entity object only
- Do not change, add, remove any existing methods in service layer
- In Repository interfaces, custom methods can be added as per requirements.

3 DATABASE OPERATIONS

1. Department

- Class must be treated as an entity.
- Id must be of type id and generated by IDENTITY technique.
- `name` should not be blank and must be between 3 and 255 characters.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Department name is required"`
    - ➢ If size is invalid: `"Department name must be between 3 and 255 characters"`
- `employeeId` must not be null.
  - ➔ **Message if invalid:** `"Employee ID must not be null"`

2. Employee

- Class must be treated as an entity.
- Id must be of type id and generated by IDENTITY technique.
- `name` should not be blank and must be between 3 and 255 characters.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Name is required"`
    - ➢ If size is invalid: `"Name must be between 3 and 255 characters"`
- `email` should not be blank and must follow a valid email format.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Email is required"`
    - ➢ If invalid format: `"Email should be valid"`
- `position` should not be blank and must be between 2 and 100 characters.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Position is required"`
    - ➢ If size is invalid: `"Position must be between 2 and 100 characters"`

4 SYSTEM REQUIREMENTS

4.1 EUREKA-NAMING-SERVER

This is a discovery server for all the registered microservices. Following implementations are expected to be done:

a. Configure the Eureka server to run on port: 8761.
b. Configure the Eureka server to deregister itself as Eureka client.
c. Add appropriate annotation to Enable this module to run as Eureka Server.

You can launch the admin panel of Eureka server in the browser preview option.

4.2 EMPLOYEE-MICROSERVICE

The employee microservice manages employee-related operations. In this microservice, you have to write the logic for EmployeeService.java and EmployeeController.java classes. Following implementations are expected to be done:

a. Configure this service to run on port: 8081.

4.3 DEPARTMENT-MICROSERVICE

The department microservice manages department-related operations and communicates with Employee microservice via RESTTemplate. In this microservice, you have to write the logic for DepartmentService.java and DepartmentController.java. Following implementations are expected to be done:

a. Configure this service to run on port: 8082.
b. Configure a RESTTemplate to fetch Employee details from Employee microservice by Employee ID.

5 TEMPLATE CODE STRUCTURE

5.1 DEPARTMENT

1 PACKAGE: COM.DEPARTMENT

Resources

DepartmentServiceApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

2 PACKAGE: COM.DEPARTMENT.REPO

Resources

Class/Interface	Description	Status
DepartmentRepository (interface)	● Repository interface exposing CRUD functionality for Department entity. ● You can go ahead and add any custom methods as per requirements.	Already Implemented

3 PACKAGE: COM.DEPARTMENT.SERVICE

Class/Interface	Description	Status
DepartmentService (class)	● Contains template method implementation. ● Need to provide implementation for managing departments related functionalities. ● Do not modify, add or delete any method signature.	To be implemented.

4 PACKAGE: COM.DEPARTMENT.CONTROLLER

Resources

Class/Interface	Description	Status
DepartmentController (Class)	● Controller class to expose all rest-endpoints for department related activities. ● May also contain local exception handler methods.	To be implemented

5 PACKAGE: COM.DEPARTMENT.DTO

Resources

Class/Interface	Description	Status
DepartmentResponse (Class)	Used to wrap department details along with associated employee details. Acts as a response DTO for combined data.	Already implemented.
EmployeeDTO (Class)	DTO representing an employee with fields (ID, name, position). Used in	Already implemented.

	DepartmentResponse.	

6 PACKAGE: COM.DEPARTMENT.ENTITY

Resources

Class/Interface	**Description**	**Status**
Department (Class)	● This class is partially implemented. ● Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. ● Generate the **id** using the IDENTITY strategy	Partially implemented.

7 PACKAGE: COM.DEPARTMENT.EXCEPTION

Resources

Class/Interface	**Description**	**Status**
NotFoundException (Class)	● Custom Exception to be thrown when trying to fetch, update or delete the department info which does not exist. ● Need to create Exception Handler for same wherever needed (local or global)	Already implemented.
ErrorResponse (Class)	● RestControllerAdvice Class for defining global exception handlers.	Already implemented.

	• Contains Exception Handler for **InvalidDataException** class. • Use this as a reference for creating exception handler for other custom exception classes	
RestExceptionHandler (Class)	• RestControllerAdvice Class for defining rest exception handlers. • Contains Exception Handler for **NotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.

5.2 EMPLOYEE

1 PACKAGE: COM.EMPLOYEE

Resources

EmployeeServiceApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

2 PACKAGE: COM.EMPLOYEE.REPO

Resources

Class/Interface	**Description**	**Status**
EmployeeRepository (interface)	● Repository interface exposing CRUD functionality for Employee entity. ● You can go ahead and add any custom methods as per requirements.	Already Implemented

3 PACKAGE: COM.EMPLOYEE.SERVICE

Class/Interface	**Description**	**Status**
EmployeeService (class)	● Contains template method implementation. ● Need to provide implementation for managing employee related functionalities. ● Do not modify, add or delete any method signature.	To be implemented.

4 PACKAGE: COM.EMPLOYEE.CONTROLLER

Resources

Class/Interface	**Description**	**Status**
EmployeeController (Class)	● Controller class to expose all rest-endpoints for employee related activities. ● May also contain local exception handler methods.	To be implemented

5 PACKAGE: COM.EMPLOYEE.ENTITY

Resources

Class/Interface	**Description**	**Status**
Employee (Class)	● This class is partially implemented. ● Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. ● Generate the **id** using the IDENTITY strategy	Partially implemented.

6 PACKAGE: COM.EMPLOYEE.EXCEPTION

Resources

Class/Interface	**Description**	**Status**
NotFoundException (Class)	● Custom Exception to be thrown when trying to fetch, update or delete the employee info which does not exist. ● Need to create Exception Handler for same wherever needed (local or global)	Already implemented.
ErrorResponse (Class)	● RestControllerAdvice Class for defining global exception handlers.	Already implemented.

	• Contains Exception Handler for **InvalidDataException** class. • Use this as a reference for creating exception handler for other custom exception classes	
RestExceptionHandler (Class)	• RestControllerAdvice Class for defining rest exception handlers. • Contains Exception Handler for **NotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.

6 METHOD DESCRIPTIONS

1. Service Class - Method Descriptions

A. DepartmentService – Method Descriptions

- Declare dependencies for `DepartmentRepository` and `RestTemplate` using `@Autowired`.

Method	**Task**	**Implementation Details**

`@Autowired`
 	
`private DepartmentRepository departmentRepository`
	Inject repository dependency	- Annotated with `@Autowired`
		
		- Provides access to department DB operations

`@Autowired` `private RestTemplate restTemplate`	Inject RestTemplate dependency	- Used to call Employee microservice using REST

Method	Task	Implementation Details
`save()`	To implement logic for saving a new department	- Calls `departmentRepository.save(dept)` - Returns the saved department
`get()`	To implement logic for retrieving a department by ID	- Calls `departmentRepository.findById(id)` - Returns Optional<Department>
`update()`	To update department details by ID	- Checks existence using `departmentRepository.existsById(id)` - If not found, throws `NotFoundException` with message: "Department with id " + id + " not found" - Sets ID and updates using `save()`
`getDepartmentWithEmployee()`	Get department and its associated employee	- Calls `departmentRepository.findById(id)` - If not found, throws `NotFoundException` with message: "Department with id " + id + " not found" - Uses `restTemplate.getForObject()` to fetch employee - If employee not found (via REST), throws `NotFoundException` with message: "Employee with id " + employeeId+ " not found” - Constructs and returns `DepartmentResponse`

B. EmployeeService – Method Descriptions

- Declare dependencies for `EmployeeRepository` using `@Autowired`.

Method	Task	Implementation Details
`@Autowired` **`private EmployeeRepository employeeRepository`**	Inject repository dependency	- Annotated with `@Autowired` - Injects the `EmployeeRepository` for database operations

Method	Task	Implementation Details
`createEmployee()`	Save a new employee	- Calls `employeeRepository.save(employee)` - Persists the new employee record - Returns the saved `Employee` object
`getEmployee()`	Get an employee by ID	- Calls `employeeRepository.findById(id)` - Returns `Optional<Employee>`
`updateEmployee()`	Update employee details by ID	- Calls `employeeRepository.findById(id)` - If not found, throws `NotFoundException` with message: "Employee with id " + id + " not found" - Updates name, email, and position - Saves the updated employee using `save()` - Returns updated `Employee`

2. Controller Class - Method Descriptions

A. DepartmentController – Method Descriptions

- Declare a private variable with name `departmentService` of type `DepartmentService` and inject it using `@Autowired`.

Method	Task	Implementation Details
`@Autowired private DepartmentService departmentService`	Field-based dependency injection	- Annotated with `@Autowired` - Injects the `DepartmentService` instance for controller use

Method	Task	Implementation Details
`create()`	To create a new department	- Request type: **POST**, URL: `/api/departments` - Accepts `Department` entity from request body - Calls `departmentService.save(dept)` - Returns the created department
`get()`	To fetch department and its employee by ID	- Request type: **GET**, URL: `/api/departments/{id}` - Uses `@PathVariable` to get `id` - Calls `departmentService.getDepartmentWithEmployee(id)` - Returns `DepartmentResponse` object
`updateDepartment()`	To update an existing department by ID	- Request type: **PUT**, URL: `/api/departments/{id}` - Accepts `Department` from request body - Calls `departmentService.update(id, dept)` - Returns updated department wrapped in `ResponseEntity` with **ResponseEntity.ok()**

B. EmployeeController – Method Descriptions

- Declare a private variable named `employeeService` of type `EmployeeService` and inject it using `@Autowired`.

Method	Task	Implementation Details

`@Autowired`
 	
`private EmployeeService employeeService`
	Field-based dependency injection	- Annotated with `@Autowired`
		
		- Injects the `EmployeeService` instance for controller use

Method	Task	Implementation Details
`create()`	To create a new employee	- Request type: **POST**, URL: `/api/employees` - Accepts `Employee` entity using `@RequestBody` - Calls `employeeService.createEmployee(employee)` - Returns created employee with `ResponseEntity.ok()`
`get()`	To retrieve employee details by ID	- Request type: **GET**, URL: `/api/employees/{id}` - Extracts `id` using `@PathVariable` - Calls `employeeService.getEmployee(id)` - If found, returns employee with **`ResponseEntity.ok()`** - If not found, returns **`ResponseEntity.notFound().build()`**
`updateDepartment()`	To update an employee by ID	- Request type: **PUT**, URL: `/api/employees/{id}` - Accepts `Employee` data using `@RequestBody` - Calls `employeeService.updateEmployee(id, employee)` - Returns updated employee with **`ResponseEntity.ok()`**

7 Microservices Communication

Communication among the microservices needs to be achieved by using **RestTemplate**. A RestTemplate bean is configured within the application, but you are required to implement the communication logic in the `DepartmentService.java` class to interact with the Employee microservice.

- You are specifically required to configure the RestTemplate to fetch **Employee Details by Employee ID** from the **Employee-Microservice**.

8 Rest Endpoints

Rest Endpoints to be exposed in the controller along with method details for the same to be created

a. EmployeeController

URL Exposed		Purpose
1. /api/employees		Creates a new Employee
Http Method	POST **The employee data to be created must be received in the controller using @RequestBody.**	
Parameter 1	`Employee` Entity	
Return	`Employee` Entity	
2. /api/employees/{id}		Retrieves Employee details by ID
Http Method	GET	
Path Variable	Long id	
Return	`Employee` Entity	
3. /api/employees/{id}		Updates Employee details by ID
Http Method	PUT **The updated employee data must be received using @RequestBody.**	

Path Variable	Long id	
Parameter	Employee Entity	
Return	Employee Entity	

b. DepartmentController

URL Exposed		Purpose
1. /api/departments		Creates a new Department
Http Method	POST **The department data must be received using @RequestBody.**	
Parameter 1	Department Entity	
Return	Department Entity	
2. /api/departments/{id}		Retrieves Department and associated Employee by Department ID
Http Method	GET	
Path Variable	Long id	
Return	DepartmentResponse DTO (includes Department + Employee)	
3. /api/departments/{id}		Updates Department details by ID
Http Method	PUT **The updated department data must be received using @RequestBody.**	
Path Variable	Long id	
Parameter	Department Entity	
Return	Department Entity	

9 SEQUENCE TO EXECUTE

The sequence has to be followed for step 10 for every microservice are given below:

- eureka-naming-server
- department-micro-service
- employee-micro-service

**Strictly follow the above sequence to follow.

10 EXECUTION STEPS TO FOLLOW

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.**
3. **cd into your backend project folder**
4. **To build your project use command:**

   mvn clean package -Dmaven.test.skip
5. **To launch your application, move into the target folder (cd target). Run the following command to run the application:**

   java -jar <your application jar file name>
6. **This editor Auto Saves the code.**
7. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.**
8. **To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN.**
9. **To test any UI based application the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.**

10. **Default credentials for MySQL:**

   a. **Username: root**

   b. **Password: pass@word1**

11. **To login to mysql instance: Open new terminal and use following command:**

   a. **sudo systemctl enable mysql**

   b. **sudo systemctl start mysql**

   NOTE: After typing any of the above commands you might encounter any warnings.

   >> Please note that this warning is expected and can be disregarded. Proceed to the next step.

   c. **mysql -u root -p**

   The last command will ask for password which is 'pass@word1'

12. **Mandatory: Before final submission run the following command:**

   mvn test