**ADDITIONAL EXERCISES HANDSON** **SOLUTIONS-WEEK-04**

***-AAISHA SULTANA GUDURU 6384619***

**3. SPRING-REST-HANDSON**

**Problem Statement - Display Employee List and Edit Employee form using RESTful Web Service**

In the previous angular module, we developed a screen that lists employees and it was populated with hard coded values. Now this angular application has be changed to get the data from RESTful Web Service developed in Spring. The following are the high level activities that needs to be done to accomplish this:

· Create static employee list data using spring xml configuration

· Create a REST Service that reads data from xml configuration and returns it

· Make changes in angular component to consume the created REST Service

Once above activities are completed, clicking on the Edit button against each employee should display Edit Employee form with values retrieved from RESTful Web Service.

This will also involve activities similar to the one specified above. NOTE: There is no specific activity as part of this hands on, refer the next hands ons that covers above three activities in detail.

Create static employee list data using spring xml configuration Follow steps below to accomplish this activity:

· Incorporate the following in employee.xml:

o Create one or two more departments

o Create four more instances of Employee. (use employee sample data from angular)

o Reuse existing skills instead of creating new ones

o Include all four employee instances in an ArrayList.

· In EmployeeDao, incorporate the following:

o Create static variable with name EMPLOYEE\_LIST of type ArrayList<Employee>

o Include constructor that reads employee list from xml config and set the EMPLOYEE\_LIST

o Create method getAllEmployees() that returns the EMPLOYEE\_LIST

Create REST service to gets all employees Follow steps below to accomplish this activity:

· In EmployeeService, incorporate the following:

o Change the annotation for this class from @Component to @Service

o Create method getAllEmployees() that invokes employeeDao.getAllEmployees() and return the employee list

o Define @Transactional annotation for this method.

· In EmployeeController, incorporate the following:

o Include a new get method with name getAllEmployees() that returns the employee list

o Mark this method as GetMapping annotation with the URL as '/employees'

o Within this method invoke employeeService.getAllEmployees() and return the same.

· Test the service using postman.

Create REST service for department

Create a new service to get all the departments.

Follow steps below to achieve this:

· Create a new REST Service, define below list of classes and respective methods:

o DepartmentController

§ getAllDepartments() with URL "/departments", this method will return array of departments

o DepartmentService

§ getAllDepartments()

o DepartmentDao

§ getAllDepartments() - Create a static variable DEPARTMENT\_LIST, this should be populated from spring xml configuration

· Test the service using postman.

· Also verify if department REST service is called by looking into the logs

**IMPLEMENTATION:**

The spring-rest hands-on has been implemented as a single Spring Boot service that exposes two read-only REST resources backed entirely by Spring XML.

The runtime stack is Java 17 and Spring Boot 3.5.4 with Maven; Jackson provides JSON serialization out of the box through the Web starter.

At startup the application imports src/main/resources/employee.xml via @ImportResource("classpath:employee.xml"), making the XML-defined graph available to the container before any web requests are served.

The XML configuration models a small HR domain and aligns precisely with the brief.Departments include Engineering, HR, and Finance; a reusable skill catalog defines Java, Angular, and SQL once and is referenced wherever needed.

The dataset contains six employees in total, satisfying the requirement to add four additional instances while reusing skills rather than duplicating them.

To make the collections consumable by the Java layer, the XML publishes employeeList and departmentList as ArrayList beans. Because these lists are created by Spring at boot, the data becomes immediately accessible to downstream layers with no database dependency.

The repository tier is implemented as lightweight DAOs that are intentionally simple and read-only.

EmployeeDao declares a static EMPLOYEE\_LIST field and a constructor that receives the XML’s employeeList via @Qualifier, assigning it to the static field for fast in-memory retrieval.

A getAllEmployees() method returns that list without mutation. DepartmentDao mirrors the pattern with DEPARTMENT\_LIST and getAllDepartments().

This satisfies the specific instructions while keeping the data source centralized in the XML file.

A thin service layer sits above the DAOs to preserve separation of concerns and enable future evolution. EmployeeService and DepartmentService are annotated with @Service and expose getAllEmployees() and getAllDepartments() respectively. Both methods are marked @Transactional(readOnly = true) to communicate intent even though no persistence transaction is required; the transaction annotation is harmless in this context and keeps the service signature consistent with typical Spring service design. Any later move to a real repository (JPA or JDBC) would be localized to the DAO layer without changing the web API.

The web layer provides precisely the two endpoints requested by the hands-on. EmployeeController exposes GET /api/employees, returning a JSON array of employees with nested department and skills objects generated by Jackson from the POJOs. DepartmentController exposes GET /api/departments, returning the JSON array of departments defined in the XML. Both controllers are stateless, side-effect free, and return 200 OK on success.

Operationally, the service runs on an available port; port 8083 was used during verification to avoid a local conflict with other processes. The console confirms embedded Tomcat initialization, XML context import, and registration of handler mappings for /api/employees and /api/departments. Invocation logs demonstrate request flow through controller and service to the DAO methods.

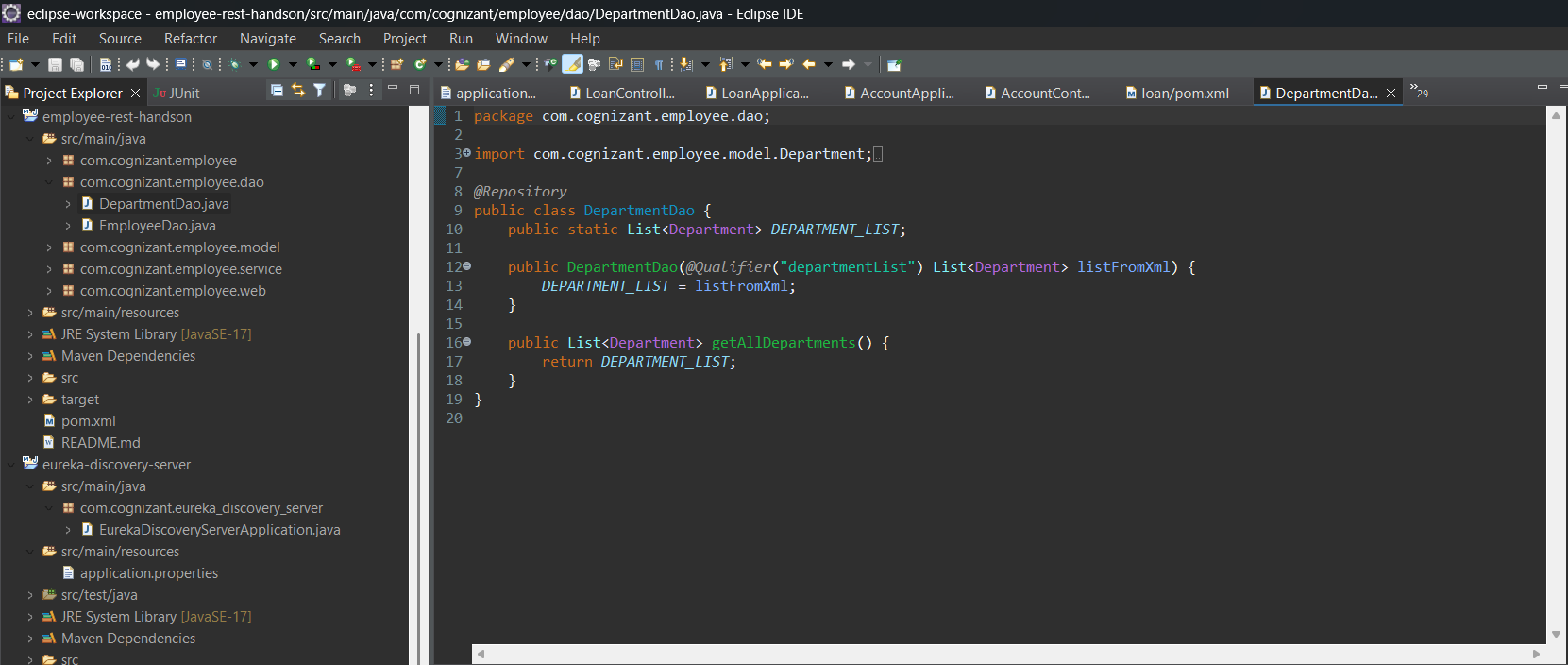
End-to-end validation was performed with both a browser and Postman. Requests to /api/employees consistently returned six records with the correct nested structures, and /api/departments returned the expected three departments.

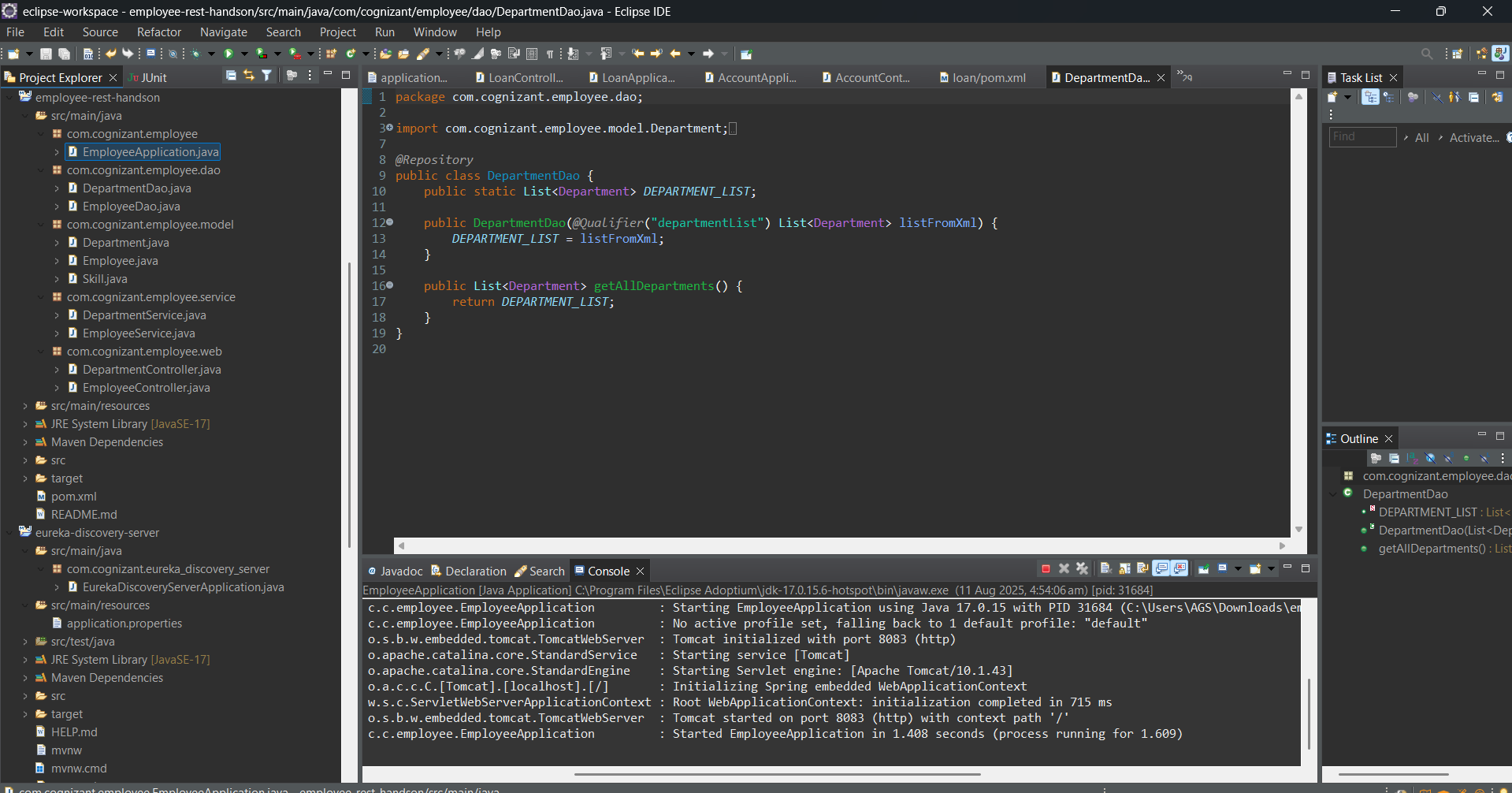
Responses serialize as compact JSON by default; when viewed in Postman with pretty-print enabled, the hierarchical structure is evident and matches the XML source.

From a build and packaging perspective, the Maven POM includes spring-boot-starter-web for the REST stack and spring-tx to compile @Transactional.

The project builds as a standard executable JAR and can be started via mvn spring-boot:run or java -jar target/…​.jar; the port can be changed by setting server.port in application.properties or by passing --server.port=XXXX at runtime. The codebase is structured for easy extension: adding GET /api/employees/{id} or introducing filtering/paging would involve only the controller and service; migrating from XML to a persistent repository would be localized to the DAO implementations and configuration.

In summary, the deliverable fulfills the handson exactly as specified: a Spring XML–backed dataset with additional departments and employees that reuse shared skills; collection beans exposed for consumption; DAO to Service to Controller layering implemented with read-only semantics and two REST endpoints that were exercised successfully in Postman.The successful executions of the project is shown as snippets below.





OUTPUT:

