**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.

### Problem Statement: Display Employee List and Edit Employee Form Using RESTful Web Service

This problem involves creating a RESTful Web Service in Spring to fetch and display an employee list, which is then consumed by an Angular front-end application. The goal is to transform a previously hardcoded list of employees into a dynamic list that is fetched from a backend service.

#### Key Activities to Accomplish:

1. **Create Static Employee List Data Using Spring XML Configuration**
2. **Create a REST Service in Spring to Return Employee Data**
3. **Modify Angular Component to Consume the REST Service and Display/Edit Employee Data**

### Step-by-Step Implementation:

#### 1. **Create Static Employee List Data Using Spring XML Configuration**

To create a static employee list in Spring using XML configuration, you would define the employee data in an XML file and then load it as a Spring bean.

**Step 1.1: Create Employee Class**

Define an Employee class that will be used to hold employee data.

javaCopyEditpublic class Employee {  
 private int id;  
 private String name;  
 private String designation;  
 private double salary;  
  
 // Constructor, getters, and setters  
  
 public Employee(int id, String name, String designation, double salary) {  
 this.id = id;  
 this.name = name;  
 this.designation = designation;  
 this.salary = salary;  
 }  
  
 // Getters and setters  
}

**Step 1.2: Define Employees in XML Configuration**

In applicationContext.xml or another configuration file, you can define a list of employees as beans.

xmlCopyEdit<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans   
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- Define individual employee beans -->  
 <bean id="employee1" class="com.example.Employee">  
 <constructor-arg value="1"/>  
 <constructor-arg value="John Doe"/>  
 <constructor-arg value="Software Engineer"/>  
 <constructor-arg value="70000"/>  
 </bean>  
  
 <bean id="employee2" class="com.example.Employee">  
 <constructor-arg value="2"/>  
 <constructor-arg value="Jane Smith"/>  
 <constructor-arg value="Product Manager"/>  
 <constructor-arg value="85000"/>  
 </bean>  
  
 <!-- Define a list of employees -->  
 <bean id="employeeList" class="java.util.ArrayList">  
 <constructor-arg>  
 <list>  
 <ref bean="employee1"/>  
 <ref bean="employee2"/>  
 </list>  
 </constructor-arg>  
 </bean>  
  
</beans>

**Step 1.3: Create a Service to Access Employee List**

You need a service that fetches the employee list from the XML configuration.

javaCopyEditimport java.util.List;  
  
public class EmployeeService {  
  
 private List<Employee> employeeList;  
  
 public void setEmployeeList(List<Employee> employeeList) {  
 this.employeeList = employeeList;  
 }  
  
 public List<Employee> getEmployeeList() {  
 return employeeList;  
 }  
}

In the Spring configuration file, you would inject the employeeList bean into this service.

xmlCopyEdit<bean id="employeeService" class="com.example.EmployeeService">  
 <property name="employeeList" ref="employeeList"/>  
</bean>

#### 2. **Create REST Service to Return Employee Data**

Now, we will expose a RESTful API to fetch the employee list from the backend using Spring's @RestController.

**Step 2.1: Create Employee REST Controller**

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class EmployeeController {  
  
 @Autowired  
 private EmployeeService employeeService;  
  
 // Endpoint to retrieve all employees  
 @GetMapping("/employees")  
 public List<Employee> getEmployees() {  
 return employeeService.getEmployeeList();  
 }  
  
 // Endpoint to retrieve a single employee by ID  
 @GetMapping("/employee/{id}")  
 public Employee getEmployeeById(@PathVariable int id) {  
 List<Employee> employees = employeeService.getEmployeeList();  
 for (Employee employee : employees) {  
 if (employee.getId() == id) {  
 return employee;  
 }  
 }  
 return null; // Return null if employee not found  
 }  
}

**Step 2.2: Configure Spring to Enable REST**

Make sure that your Spring configuration has proper setup for enabling REST services. If you are using Spring Boot, ensure you have @SpringBootApplication on the main class.

javaCopyEditimport org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class EmployeeApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(EmployeeApplication.class, args);  
 }  
}

#### 3. **Modify Angular Component to Consume the REST Service**

Now that the REST API is ready, modify your Angular front-end to consume the employee data and display it dynamically.

**Step 3.1: Create Angular Service to Fetch Employee Data**

Create an Angular service to interact with the backend and retrieve employee data.

typescriptCopyEditimport { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
import { Observable } from 'rxjs';  
  
@Injectable({  
 providedIn: 'root'  
})  
export class EmployeeService {  
  
 private apiUrl = 'http://localhost:8080/employees'; // Adjust the URL as per your backend  
  
 constructor(private http: HttpClient) {}  
  
 getEmployees(): Observable<Employee[]> {  
 return this.http.get<Employee[]>(this.apiUrl);  
 }  
  
 getEmployeeById(id: number): Observable<Employee> {  
 return this.http.get<Employee>(`${this.apiUrl}/${id}`);  
 }  
}

**Step 3.2: Create Angular Component to Display Employees**

Create an Angular component that will display the employee list and edit form.

typescriptCopyEditimport { Component, OnInit } from '@angular/core';  
import { EmployeeService } from './employee.service';  
import { Employee } from './employee.model';  
  
@Component({  
 selector: 'app-employee-list',  
 templateUrl: './employee-list.component.html',  
 styleUrls: ['./employee-list.component.css']  
})  
export class EmployeeListComponent implements OnInit {  
  
 employees: Employee[] = [];  
 selectedEmployee: Employee | null = null;  
  
 constructor(private employeeService: EmployeeService) {}  
  
 ngOnInit(): void {  
 this.employeeService.getEmployees().subscribe(data => {  
 this.employees = data;  
 });  
 }  
  
 onEdit(employee: Employee): void {  
 this.employeeService.getEmployeeById(employee.id).subscribe(data => {  
 this.selectedEmployee = data;  
 });  
 }  
}

**Step 3.3: Create Angular Template for Display and Edit**

htmlCopyEdit<div \*ngIf="selectedEmployee">  
 <h2>Edit Employee</h2>  
 <form>  
 <label for="name">Name:</label>  
 <input id="name" [(ngModel)]="selectedEmployee.name">  
 <label for="designation">Designation:</label>  
 <input id="designation" [(ngModel)]="selectedEmployee.designation">  
 <label for="salary">Salary:</label>  
 <input id="salary" [(ngModel)]="selectedEmployee.salary">  
 <button type="submit">Save</button>  
 </form>  
</div>  
  
<h2>Employee List</h2>  
<ul>  
 <li \*ngFor="let employee of employees">  
 {{ employee.name }} - {{ employee.designation }}   
 <button (click)="onEdit(employee)">Edit</button>  
 </li>  
</ul>

**Step 3.4: Add Angular Model for Employee**

typescriptCopyEditexport class Employee {  
 id: number;  
 name: string;  
 designation: string;  
 salary: number;  
}

### Final Structure and Testing:

* **Backend:**
  + Spring controller exposes RESTful API to fetch employees.
  + Spring XML configuration defines static employee list.
* **Frontend:**
  + Angular service makes HTTP requests to consume REST APIs.
  + Angular component displays the list and shows edit form.

### Conclusion:

This solution sets up a Spring-based REST API to expose employee data, which is then consumed by an Angular front-end. When an employee is selected for editing, their details are fetched via the REST API and displayed in an editable form.

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

* Incorporate the following in employee.xml:
  + Create one or two more departments
  + Create four more instances of Employee.  (use employee sample data from angular)
  + Reuse existing skills instead of creating new ones
  + Include all four employee instances in an ArrayList.

* In EmployeeDao, incorporate the following:
  + Create static variable with name EMPLOYEE\_LIST of type ArrayList<Employee>
  + Include constructor that reads employee list from xml config and set the EMPLOYEE\_LIST
  + Create method getAllEmployees() that returns the EMPLOYEE\_LIST

### Step-by-Step Implementation for Creating Static Employee List Using Spring XML Configuration

Here we will create static employee data with multiple departments and employee instances using Spring XML configuration. We will also modify the EmployeeDao to store and fetch the employee data.

### 1. **Create Static Employee List Data Using Spring XML Configuration**

#### 1.1: **Create Employee Class**

First, ensure you have an Employee class that will hold the employee data.

javaCopyEditpublic class Employee {  
 private int id;  
 private String name;  
 private String designation;  
 private double salary;  
 private String department;  
 private List<String> skills;  
  
 public Employee(int id, String name, String designation, double salary, String department, List<String> skills) {  
 this.id = id;  
 this.name = name;  
 this.designation = designation;  
 this.salary = salary;  
 this.department = department;  
 this.skills = skills;  
 }  
  
 // Getters and setters  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getDesignation() {  
 return designation;  
 }  
  
 public void setDesignation(String designation) {  
 this.designation = designation;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(double salary) {  
 this.salary = salary;  
 }  
  
 public String getDepartment() {  
 return department;  
 }  
  
 public void setDepartment(String department) {  
 this.department = department;  
 }  
  
 public List<String> getSkills() {  
 return skills;  
 }  
  
 public void setSkills(List<String> skills) {  
 this.skills = skills;  
 }  
}

#### 1.2: **Create XML Configuration for Employees (employee.xml)**

Now, let's define the employees in the employee.xml configuration file. We'll add new departments, more employee instances, and reuse existing skills.

xmlCopyEdit<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans   
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- Define skills -->  
 <bean id="skill1" class="java.lang.String">  
 <constructor-arg value="Java"/>  
 </bean>  
 <bean id="skill2" class="java.lang.String">  
 <constructor-arg value="Spring"/>  
 </bean>  
 <bean id="skill3" class="java.lang.String">  
 <constructor-arg value="Angular"/>  
 </bean>  
 <bean id="skill4" class="java.lang.String">  
 <constructor-arg value="Database"/>  
 </bean>  
  
 <!-- Define departments -->  
 <bean id="department1" class="java.lang.String">  
 <constructor-arg value="Software Development"/>  
 </bean>  
 <bean id="department2" class="java.lang.String">  
 <constructor-arg value="Marketing"/>  
 </bean>  
  
 <!-- Define employees -->  
 <bean id="employee1" class="com.example.Employee">  
 <constructor-arg value="1"/>  
 <constructor-arg value="John Doe"/>  
 <constructor-arg value="Software Engineer"/>  
 <constructor-arg value="70000"/>  
 <constructor-arg ref="department1"/>  
 <constructor-arg>  
 <list>  
 <ref bean="skill1"/>  
 <ref bean="skill2"/>  
 </list>  
 </constructor-arg>  
 </bean>  
  
 <bean id="employee2" class="com.example.Employee">  
 <constructor-arg value="2"/>  
 <constructor-arg value="Jane Smith"/>  
 <constructor-arg value="Product Manager"/>  
 <constructor-arg value="85000"/>  
 <constructor-arg ref="department1"/>  
 <constructor-arg>  
 <list>  
 <ref bean="skill2"/>  
 <ref bean="skill3"/>  
 </list>  
 </constructor-arg>  
 </bean>  
  
 <bean id="employee3" class="com.example.Employee">  
 <constructor-arg value="3"/>  
 <constructor-arg value="Robert Brown"/>  
 <constructor-arg value="Marketing Executive"/>  
 <constructor-arg value="60000"/>  
 <constructor-arg ref="department2"/>  
 <constructor-arg>  
 <list>  
 <ref bean="skill3"/>  
 <ref bean="skill4"/>  
 </list>  
 </constructor-arg>  
 </bean>  
  
 <bean id="employee4" class="com.example.Employee">  
 <constructor-arg value="4"/>  
 <constructor-arg value="Emily White"/>  
 <constructor-arg value="HR Specialist"/>  
 <constructor-arg value="55000"/>  
 <constructor-arg ref="department1"/>  
 <constructor-arg>  
 <list>  
 <ref bean="skill1"/>  
 <ref bean="skill4"/>  
 </list>  
 </constructor-arg>  
 </bean>  
  
 <!-- Define an ArrayList to hold the employees -->  
 <bean id="employeeList" class="java.util.ArrayList">  
 <constructor-arg>  
 <list>  
 <ref bean="employee1"/>  
 <ref bean="employee2"/>  
 <ref bean="employee3"/>  
 <ref bean="employee4"/>  
 </list>  
 </constructor-arg>  
 </bean>  
</beans>

In this configuration:

* We have created four employees (employee1, employee2, employee3, employee4), each with their respective department and skills.
* We reused the skills and departments that were previously created in the XML configuration.
* All four employee beans are added to an ArrayList, which will hold the employee data.

#### 1.3: **Create EmployeeDao to Access the Employee List**

Now we need to modify the EmployeeDao class to store the employee list and provide a method to fetch all employees.

javaCopyEditimport java.util.ArrayList;  
import java.util.List;  
  
public class EmployeeDao {  
  
 // Static variable to hold the list of employees  
 private static List<Employee> EMPLOYEE\_LIST;  
  
 // Constructor that reads employee list from XML configuration and sets the EMPLOYEE\_LIST  
 public EmployeeDao(List<Employee> employeeList) {  
 EMPLOYEE\_LIST = employeeList;  
 }  
  
 // Method to fetch all employees  
 public List<Employee> getAllEmployees() {  
 return EMPLOYEE\_LIST;  
 }  
}

In this class:

* The EMPLOYEE\_LIST is a static variable that will hold the list of employees.
* The constructor of EmployeeDao takes a List<Employee> as a parameter, which is injected from the Spring configuration (the employeeList bean).
* The getAllEmployees() method returns the EMPLOYEE\_LIST.

### 2. **Spring Configuration to Inject EmployeeDao**

Make sure that the Spring configuration file is set to inject the employeeList bean into the EmployeeDao class.

xmlCopyEdit<bean id="employeeDao" class="com.example.EmployeeDao">  
 <constructor-arg ref="employeeList"/>  
</bean>

Here, the EmployeeDao bean is created, and the employeeList (which is an ArrayList<Employee>) is injected into it.

### 3. **Conclusion**

This implementation creates a static list of employees using Spring XML configuration, which includes multiple departments, employee instances, and skills. The EmployeeDao class is set up to hold this list and provide a method to fetch all employees. The XML configuration file contains the employee list, skills, and departments, which are injected into the EmployeeDao class.

This setup is ready for further integration with the rest of your application, such as creating RESTful services or connecting with Angular for front-end consumption.

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

* In EmployeeService, incorporate the following:
  + Change the annotation for this class from @Component to @Service
  + Create method getAllEmployees() that invokes employeeDao.getAllEmployees() and return the employee list
  + Define @Transactional annotation for this method.

* In EmployeeController, incorporate the following:
  + Include a new get method with name getAllEmployees() that returns the employee list
  + Mark this method as GetMapping annotation with the URL as '/employees'
  + Within this method invoke employeeService.getAllEmployees() and return the same.

​​​​​​

* Test ​the service using postman.

### Step-by-Step Implementation for Creating a REST Service to Get All Employees

This task involves creating a REST service to return all employees by using Spring annotations and a service layer. Let's break this down into steps:

### 1. **Create EmployeeService to Fetch All Employees**

#### 1.1: **Change the Annotation in EmployeeService**

You need to change the annotation for the EmployeeService class from @Component to @Service, as it is a service layer class.

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
import java.util.List;  
  
@Service  
public class EmployeeService {  
  
 @Autowired  
 private EmployeeDao employeeDao;  
  
 // Method to get all employees  
 @Transactional  
 public List<Employee> getAllEmployees() {  
 return employeeDao.getAllEmployees();  
 }  
}

**Explanation:**

* @Service: This annotation marks the class as a Spring service, which is typically used for business logic.
* @Transactional: This annotation ensures that the method is executed within a transaction. Any exceptions or rollbacks will be automatically handled by Spring if required.

#### 1.2: **EmployeeDao Class**

The EmployeeDao class should already have the getAllEmployees method that retrieves the employee list from the XML configuration.

javaCopyEditimport java.util.List;  
  
public class EmployeeDao {  
  
 private static List<Employee> EMPLOYEE\_LIST;  
  
 public EmployeeDao(List<Employee> employeeList) {  
 EMPLOYEE\_LIST = employeeList;  
 }  
  
 public List<Employee> getAllEmployees() {  
 return EMPLOYEE\_LIST;  
 }  
}

Ensure that the EMPLOYEE\_LIST is populated from the Spring XML configuration as described in the previous step.

### 2. **Create EmployeeController to Expose the REST API**

Now, let's create the controller that will expose the getAllEmployees method as a RESTful API endpoint.

#### 2.1: **Create EmployeeController Class**

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
import java.util.List;  
  
@RestController  
public class EmployeeController {  
  
 @Autowired  
 private EmployeeService employeeService;  
  
 // Method to get all employees  
 @GetMapping("/employees")  
 public List<Employee> getAllEmployees() {  
 return employeeService.getAllEmployees();  
 }  
}

**Explanation:**

* @RestController: This annotation is used to create RESTful APIs. It combines @Controller and @ResponseBody to automatically serialize Java objects to JSON or XML format.
* @GetMapping("/employees"): This annotation maps HTTP GET requests to the /employees endpoint to the getAllEmployees() method in the controller.
* The getAllEmployees() method invokes the EmployeeService's getAllEmployees() method and returns the employee list.

### 3. **Test the Service Using Postman**

After implementing the above steps, you can now test the service using **Postman**.

#### 3.1: **Test the GET Request**

* **URL**: http://localhost:8080/employees (Ensure that your Spring Boot application is running on port 8080 or adjust the port as necessary)
* **Method**: GET
* **Response**: You should see a list of employees returned in JSON format.

**Example of the JSON response:**

jsonCopyEdit[  
 {  
 "id": 1,  
 "name": "John Doe",  
 "designation": "Software Engineer",  
 "salary": 70000,  
 "department": "Software Development",  
 "skills": ["Java", "Spring"]  
 },  
 {  
 "id": 2,  
 "name": "Jane Smith",  
 "designation": "Product Manager",  
 "salary": 85000,  
 "department": "Software Development",  
 "skills": ["Spring", "Angular"]  
 },  
 {  
 "id": 3,  
 "name": "Robert Brown",  
 "designation": "Marketing Executive",  
 "salary": 60000,  
 "department": "Marketing",  
 "skills": ["Angular", "Database"]  
 },  
 {  
 "id": 4,  
 "name": "Emily White",  
 "designation": "HR Specialist",  
 "salary": 55000,  
 "department": "Software Development",  
 "skills": ["Java", "Database"]  
 }  
]

### 4. **Summary of Changes**

* **EmployeeService**:
  + Changed the class annotation to @Service.
  + Added a method getAllEmployees() which fetches the employee list from EmployeeDao.
  + Applied @Transactional to ensure the method runs within a transaction.
* **EmployeeController**:
  + Created a REST endpoint /employees that uses @GetMapping.
  + The controller invokes the EmployeeService's getAllEmployees() method and returns the employee list.
* **Postman Testing**:
  + The /employees endpoint should return a JSON array with employee details.

With this, you have successfully created a RESTful web service in Spring to fetch all employees from the backend and exposed it as a REST endpoint. You can test the service using Postman as described above.

**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:
  + DepartmentController
    - getAllDepartments() with URL "/departments", this method will return array of departments
  + DepartmentService
    - getAllDepartments()
  + 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.

### Step-by-Step Implementation for Creating a REST Service to Get All Employees

The task involves exposing a REST service to get all employees from a backend service. We will go through the necessary steps to create this functionality using Spring.

### 1. **Modify EmployeeService to Fetch All Employees**

In the EmployeeService class, we'll perform the following changes:

1. Change the annotation from @Component to @Service to designate this class as a Spring service.
2. Create a method getAllEmployees() that calls the DAO method employeeDao.getAllEmployees() to fetch all employees.
3. Add @Transactional to ensure the method is executed within a transaction.

#### 1.1: **Update EmployeeService Class**

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
import java.util.List;  
  
@Service  
public class EmployeeService {  
  
 @Autowired  
 private EmployeeDao employeeDao;  
  
 // Method to get all employees  
 @Transactional  
 public List<Employee> getAllEmployees() {  
 return employeeDao.getAllEmployees();  
 }  
}

**Explanation:**

* **@Service**: Marks this class as a service component to be managed by Spring.
* **@Transactional**: Ensures that the method is executed in a transactional context, which means if there’s an exception, the transaction will roll back automatically.
* **getAllEmployees()**: Calls employeeDao.getAllEmployees() to retrieve the list of employees from the database.

### 2. **Create REST Controller to Expose the Endpoint**

In the EmployeeController class, we'll expose the getAllEmployees() method as a RESTful endpoint:

1. Use @GetMapping("/employees") to define the endpoint.
2. The method will invoke employeeService.getAllEmployees() to get the employee data and return it as a response.

#### 2.1: **Update EmployeeController Class**

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
import java.util.List;  
  
@RestController  
public class EmployeeController {  
  
 @Autowired  
 private EmployeeService employeeService;  
  
 // REST endpoint to fetch all employees  
 @GetMapping("/employees")  
 public List<Employee> getAllEmployees() {  
 return employeeService.getAllEmployees();  
 }  
}

**Explanation:**

* **@RestController**: Combines @Controller and @ResponseBody, meaning the controller will return JSON data directly in response to HTTP requests.
* **@GetMapping("/employees")**: Maps HTTP GET requests to the /employees endpoint to this method.
* **getAllEmployees()**: Calls the service's getAllEmployees() method and returns the list of employees.

### 3. **Test the Service Using Postman**

Now that the service is created, you can test it using **Postman**. Follow the steps below:

#### 3.1: **Postman Test**

* **URL**: http://localhost:8080/employees (Ensure that your Spring Boot application is running on port 8080 or adjust the port accordingly).
* **Method**: GET
* **Response**: A list of employee data in JSON format.

**Example of Expected JSON Response**:

jsonCopyEdit[  
 {  
 "id": 1,  
 "name": "John Doe",  
 "designation": "Software Engineer",  
 "salary": 70000,  
 "department": "Software Development",  
 "skills": ["Java", "Spring"]  
 },  
 {  
 "id": 2,  
 "name": "Jane Smith",  
 "designation": "Product Manager",  
 "salary": 85000,  
 "department": "Software Development",  
 "skills": ["Spring", "Angular"]  
 },  
 {  
 "id": 3,  
 "name": "Robert Brown",  
 "designation": "Marketing Executive",  
 "salary": 60000,  
 "department": "Marketing",  
 "skills": ["Angular", "Database"]  
 },  
 {  
 "id": 4,  
 "name": "Emily White",  
 "designation": "HR Specialist",  
 "salary": 55000,  
 "department": "Software Development",  
 "skills": ["Java", "Database"]  
 }  
]

#### 3.2: **Postman Configuration**

* **Method**: GET
* **URL**: http://localhost:8080/employees
* **Headers**: If needed, set the Content-Type to application/json.

### 4. **Conclusion**

This implementation creates a RESTful service that:

* Fetches a list of employees using a service (EmployeeService) that accesses the DAO (EmployeeDao).
* The EmployeeController exposes a REST endpoint (/employees) for clients to retrieve the list of employees.
* We tested the service using Postman to ensure the employee data is returned correctly in JSON format.

With this setup, your Spring application now serves employee data via a RESTful API that can be consumed by front-end applications, such as Angular or React, or any HTTP client like 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:
  + DepartmentController
    - getAllDepartments() with URL "/departments", this method will return array of departments
  + DepartmentService
    - getAllDepartments()
  + 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.

### Step-by-Step Implementation for Creating a REST Service for Departments

This task involves creating a REST service to fetch all departments. We will define the required classes (DepartmentController, DepartmentService, and DepartmentDao) and implement their respective methods. We will also configure the department data in the Spring XML file, and test the service using Postman.

### 1. **Create the Department REST Service**

We will define three main components:

1. DepartmentController: Exposes the /departments endpoint to return the list of departments.
2. DepartmentService: Contains business logic for fetching departments.
3. DepartmentDao: Holds the department list, populated from the Spring XML configuration.

### 2. **Create DepartmentDao to Manage Department List**

The DepartmentDao will be responsible for managing the static list of departments, which will be populated from the Spring XML configuration.

#### 2.1: **Create DepartmentDao Class**

javaCopyEditimport java.util.List;  
  
public class DepartmentDao {  
  
 private static List<String> DEPARTMENT\_LIST;  
  
 // Constructor to set the department list from Spring XML  
 public DepartmentDao(List<String> departmentList) {  
 DEPARTMENT\_LIST = departmentList;  
 }  
  
 // Method to get all departments  
 public List<String> getAllDepartments() {  
 return DEPARTMENT\_LIST;  
 }  
}

**Explanation:**

* **DEPARTMENT\_LIST**: A static list that holds the department names.
* The constructor accepts a list of departments injected from the Spring XML configuration.
* **getAllDepartments()**: Returns the list of all departments.

### 3. **Create DepartmentService to Access Department Data**

The DepartmentService will call the DepartmentDao to fetch the list of departments.

#### 3.1: **Create DepartmentService Class**

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class DepartmentService {  
  
 @Autowired  
 private DepartmentDao departmentDao;  
  
 // Method to get all departments  
 public List<String> getAllDepartments() {  
 return departmentDao.getAllDepartments();  
 }  
}

**Explanation:**

* **@Service**: Marks this class as a Spring service, making it eligible for dependency injection.
* **getAllDepartments()**: Calls the DepartmentDao's getAllDepartments() method to fetch the department list.

### 4. **Create DepartmentController to Expose the REST Endpoint**

The DepartmentController will expose the /departments endpoint, which will return the list of departments in JSON format.

#### 4.1: **Create DepartmentController Class**

javaCopyEditimport org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
import java.util.List;  
  
@RestController  
public class DepartmentController {  
  
 @Autowired  
 private DepartmentService departmentService;  
  
 // REST endpoint to get all departments  
 @GetMapping("/departments")  
 public List<String> getAllDepartments() {  
 return departmentService.getAllDepartments();  
 }  
}

**Explanation:**

* **@RestController**: This annotation indicates that the class is a Spring MVC controller and its methods return data directly as JSON.
* **@GetMapping("/departments")**: This annotation binds the getAllDepartments() method to HTTP GET requests at the /departments URL.
* **getAllDepartments()**: Calls departmentService.getAllDepartments() to fetch the department list and return it as a JSON response.

### 5. **Configure Department List in Spring XML**

We need to configure the department list in the Spring XML file. The list will be injected into DepartmentDao.

#### 5.1: **Update Spring XML Configuration (applicationContext.xml)**

xmlCopyEdit<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans   
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- Define department names -->  
 <bean id="department1" class="java.lang.String">  
 <constructor-arg value="Software Development"/>  
 </bean>  
 <bean id="department2" class="java.lang.String">  
 <constructor-arg value="Marketing"/>  
 </bean>  
 <bean id="department3" class="java.lang.String">  
 <constructor-arg value="Human Resources"/>  
 </bean>  
  
 <!-- Define an ArrayList to hold the departments -->  
 <bean id="departmentList" class="java.util.ArrayList">  
 <constructor-arg>  
 <list>  
 <ref bean="department1"/>  
 <ref bean="department2"/>  
 <ref bean="department3"/>  
 </list>  
 </constructor-arg>  
 </bean>  
  
 <!-- Inject the department list into DepartmentDao -->  
 <bean id="departmentDao" class="com.example.DepartmentDao">  
 <constructor-arg ref="departmentList"/>  
 </bean>  
  
 <!-- Define DepartmentService bean -->  
 <bean id="departmentService" class="com.example.DepartmentService"/>  
</beans>

**Explanation:**

* We define the department names as individual beans (department1, department2, department3).
* These beans are then added to an ArrayList, which is injected into the DepartmentDao bean.
* DepartmentService is also declared as a Spring bean.

### 6. **Test the Service Using Postman**

Now that the service is set up, you can test it using Postman.

#### 6.1: **Test the GET Request**

* **URL**: http://localhost:8080/departments (Ensure your Spring Boot application is running on the appropriate port).
* **Method**: GET
* **Response**: A JSON array containing the department names.

**Example of the JSON response:**

jsonCopyEdit[  
 "Software Development",  
 "Marketing",  
 "Human Resources"  
]

#### 6.2: **Verify Logs**

To verify that the department REST service is being called, check your application logs. You should see logs indicating that the DepartmentController and other relevant components are being invoked when the /departments endpoint is accessed.

Ensure you have enabled logging in your application.properties or application.yml for Spring Boot:

propertiesCopyEditlogging.level.org.springframework.web=DEBUG

This will log all incoming HTTP requests and the execution of controllers.

### 7. **Conclusion**

By following these steps, you have created a REST service to manage departments in your Spring application. The components include:

* **DepartmentDao**: Manages the static department list populated from the Spring XML configuration.
* **DepartmentService**: Handles the business logic for fetching departments.
* **DepartmentController**: Exposes the /departments endpoint to fetch all departments.

You can now test the service using Postman and verify if the department REST service is called by checking the logs.