Spring Boot REST Application: Job Portal

Spring Boot REST Application: Job Portal

1. Application Overview

- Name: Spring Boot Job Portal REST API

- Purpose: Manage job postings, including operations like retrieving, addi

- Technology Stack:

- Spring Boot (Spring Web, Spring Data)

- Java Collections for in-memory data storage

- Jackson for JSON/XML serialization

- Tools: Postman for API testing

2. Project Structure

1. Controller Layer (JobRestController):

- Handles HTTP requests and maps them to service methods.

- Supports JSON and XML data formats using content negotiation.

2. Service Layer (JobService):

- Contains business logic and acts as an intermediary between the contr

3. Repository Layer (JobRepo):

- Simulates a database using an in-memory list (List<JobPost>).

- Stores job postings and provides CRUD operations.

4. Model Layer (JobPost):

- Represents a job posting with attributes like postId, postProfile, postDe

5. Main Class (SpringBootRestApplication):
- Entry point for the Spring Boot application.

3. Key Components and Flow

1. Endpoints in JobRestController:
- GET /jobposts: Retrieves all job postings.
- GET /jobposts/{postId}: Retrieves a specific job post by its ID.
- POST /jobposts: Adds a new job post. Only accepts XML input (configu
- PUT /jobposts: Updates an existing job post.
- DELETE /jobposts/{postId}: Deletes a job post by its ID.

2. Data Flow:
- Controller: Receives HTTP requests, delegates processing to the servic
- Service: Contains business logic and interacts with the repository.
- Repository: Performs CRUD operations on the in-memory list.

4. Features
1. Content Negotiation:
- Supports JSON and XML formats for requests and responses.
- Example:
  - Add Accept: application/json or Accept: application/xml in the request

2. In-Memory Storage:
- Uses an ArrayList in JobRepo to simulate database operations.

3. Data Initialization:

- Pre-populates JobRepo with 20 predefined job posts.

4. Dynamic Filtering:

- Deletes job posts efficiently using List.removeIf() to avoid ConcurrentM

5. Example Endpoints

1. Retrieve All Job Posts:

Request:

GET /jobposts

Accept: application/json

Response:

```
[
  {
    "postId": 1,
    "postProfile": "Java Developer",
    "postDesc": "Must have good experience in core Java and advanced
    "reqExperience": 2,
    "postTechStack": ["Core Java", "J2EE", "Spring Boot", "Hibernate"]
  }
]
```

2. Add a New Job Post (XML Only):

Request:

POST /jobposts

Content-Type: application/xml

<JobPost>
<postId>21</postId>
<postProfile>AI Specialist</postProfile>
<postDesc>Develop AI models for various industries</postDesc>
<reqExperience>5</reqExperience>
<postTechStack>
<postTechStack>Python</postTechStack>
<postTechStack>AI</postTechStack>
</postTechStack>
</JobPost>

Response: 201 Created

3. Update a Job Post:

Request:

PUT /jobposts

Content-Type: application/json

```
{
  "postId": 1,
  "postProfile": "Senior Java Developer",
  "postDesc": "Expert in core and advanced Java",
  "reqExperience": 5,
  "postTechStack": ["Core Java", "Spring Boot"]
}
```

Response: Updated job post details.

4. Delete a Job Post:

Request:

DELETE /jobposts/1

Response: Data has been deleted successfully

6. Notes for Improvement

- Validation: Add input validation using javax.validation annotations (e.g., @

- Database Integration: Replace in-memory storage with an actual database

- Error Handling: Implement global exception handling using @ControllerA

- Logging: Add proper logging using SLF4J or Logback.