Salsa - Backend Engineer Code Test

The company **LinkedOut** is building a web application to post job positions and employees looking for a job to search and apply to the positions they are interested in. As part of their MVP, they would like to build a backend service that exposes an API (REST or GraphQL) for managing those job postings and searching them.

We are interested in assessing that both the functional and non-functional requirements are implemented.

You may use any backend framework or language of your choice (Node.js with Express/NestJS, Python with FastAPI/Django, Ruby on Rails, Java with Spring Boot/Micronaut, Go, etc.), but ensure the solution follows best practices in software engineering.

You can submit the solution as a ZIP file or host it in a private Git repository (GitHub, Bitbucket, GitLab, etc.).

Functional Requirements

There are 2 type of users:

- **Employers** who post jobs and want to manage the jobs they have published
- **Employees** looking for jobs that want to search for jobs that are interesting to them. In a future iteration (not part of this MVP), they will be able to apply to a job post

1. CRUD Operations for Jobs

Your API should support CRUD operations for employers’ job postings with the following fields:

- **Job Title**
- **Company Name**
- **Location**
- **Job Description**
- **Job Type** (e.g., Full-time, Part-time, Contract)
- **Salary Range (**min amount, max amount)
- **Benefits and Perks (**gym memberships, cafeteria, snacks, etc)
- **Extras** (whatever else the employer wish to add)

2. Search & Filtering

The ‘secret sauce’ of this application is the powerful ‘search results ranking’ algorithm that they want to start developing as part of this MVP.
Your API should support users searching for jobs and also be able to filter the results.

The search results should be based on a series of ‘rules’ that need to be applied in order (the higher the rule, the more prominence the rule has in the order of the search results):

1. Posts created in the last 7 days rank first above older posts
2. Posts with higher salaries rank first above posts with lower salaries
3. Posts made by companies with more open job posts rank first than those for companies with less posts

As with every MVP, it should be very easy to ‘move’ the rules up or down depending on what we learn from the customers, as well as being able to code new rules very easily (e.g. “Posts with full-time positions rank first above job posts with part-time”).

Also, your API should accept users to filter the search results by at least:

1. Job Title
2. Location
3. Salary Ranges

Similarly, it would be desirable that the architecture allows for extension to support more filter criteria (e.g. filter by job type; by whether it offers benefits or not; etc)

Non-Functional Requirements

Documentation

- Provide a READMe instructions on how to run the application
- Document any architecture and design decisions, trade-offs, and assumptions you made in your decision.

Scalability Considerations

- **Disk persistence is NOT required** (use an in-memory database like SQLite, Redis, or simply data structures in memory).
- Design the API as if it were to be deployed at scale. The system should assume handling **millions of job records and millions of searches**, so please document how you would think about achieving said scalability (**no need to implement it)**.

Code Quality & Testing

- Ensure that your code is clean, modular, and well-structured.
- Apply proper **error handling** and **validation**.
- Include **unit and/or integration tests** to validate core functionalities.
- Use a testing framework suitable for your tech stack (e.g., Jest, PyTest, Mocha, JUnit, etc.).

Bonus Features (Optional)

To go the extra mile, consider implementing one or more of the following:

- **Rate limiting** for API endpoints.
- **Authentication & Authorization** (JWT, API Keys, OAuth).
- **Pagination** for job listings to optimize performance.
- **Background workers** for caching search results (e.g., using a queue like RabbitMQ, Kafka, or Redis queues).
- **GraphQL API** instead of REST for flexibility.