Exercise A: Data Ingestion and Enrichment

Objective:
The goal of Exercise A is to ingest and preprocess salary survey data, and enrich it with additional inform ation like geolocation.

Steps:

1. Data Ingestion:
   - Load the raw salary survey data from a CSV file.
   - Perform initial data cleaning to handle missing values, duplicates, and format inconsistencies.

2. Salary Data Cleaning:
   - Clean the "What is your annual salary?" field to remove non-numeric characters and ensure a consiste nt format.
   - Convert the cleaned salary data to a numeric format for analysis.

3. Geolocation Enrichment:
   - Utilize a geocoding service (e.g., Nominatim) to extract city information from the provided location data .
   - Update the "Where are you located? (City/state/country)" field with the extracted city information.

4. Data Export:
   - Export the cleaned and enriched data to a new CSV file for further analysis or storage.

5. Elasticsearch Indexing:
   - Establish a connection to an Elasticsearch instance.
   - Define an index mapping that reflects the structure of the data.
   - Bulk index the cleaned and enriched data into Elasticsearch for easy querying.

6. Data Analysis:
   - Perform various analyses on the data using Elasticsearch queries, e.g., aggregations by city, average salary, etc.

Shortcomings:

1. Incomplete or inaccurate location data may lead to geocoding errors.
2. Ambiguous job titles or industry categories might require additional cleaning.
3. Data validation and cleaning procedures may vary based on the dataset.

Improvements:

1. Implement additional data validation checks and regex-based cleaning for specific fields.
2. Consider using multiple geocoding services for robust location extraction.
3. Automate the data cleaning process with custom functions for repetitive tasks.

Exercise B: Exposing a Query API

Objective:
The goal of Exercise B is to design and implement a read-only API for querying compensation data.

Steps:

1. Flask API Setup:

- Create a Flask application to serve as the API server.

2. Elasticsearch Connection:
   - Connect to the Elasticsearch instance where the compensation data is stored.

3. API Routes:
   - Define API routes for listing compensation data, fetching a single record, and implementing bonus features (e.g., sparse fieldset).

4. List Compensation Data:
   - Implement an API endpoint to list compensation data.
   - Allow users to filter and sort results based on specific fields/attributes.

5. Fetch a Single Record:
   - Create an API endpoint to fetch a single compensation record by ID.

6. Bonus Feature: Sparse Fieldset:
   - Implement a bonus feature to return a sparse fieldset of a compensation record.

7. Error Handling:
   - Include proper error handling to provide informative responses in case of invalid requests or missing data.

8. Run the API:
   - Launch the Flask application to start the API server.

9. Testing:
   - Test the API routes using tools like Postman or through direct HTTP requests.

Shortcomings:

1. Handling complex queries with multiple filters and sorting parameters may require advanced Elasticsearch queries.

Improvements:

1. Implement authentication and authorization mechanisms for secure access.
2. Enable caching mechanisms to optimize response times for frequently queried data.