TypeScript Onboarding Assignment

Resilient Data Processing with Strong Typing

Purpose

This assignment is part of the onboarding process and is intended to help you become productive in our TypeScript ecosystem.

The focus is on:

- Idiomatic TypeScript (strict typing, unions, generics, boundaries)
- Clean modular design and separation of concerns
- Correct handling of asynchronous workflows (bounded concurrency, early exit)
- Thoughtful error handling and retry behavior
- Proper use of Zod for runtime validation
- Proper usage of Axios for HTTP communication

There is no strict time limit.
Treat this as if you were implementing a small production component that will be read and maintained by other developers.

Problem Statement

Implement a small TypeScript library with a CLI that:

1. Reads a list of Pokémon names from a JSON file
2. Fetches data for each Pokémon from a public API using Axios
3. Filters entities based on business criteria
4. Processes items concurrently with bounded parallelism
5. Handles transient failures using retries and timeouts
6. Produces structured output and a run report

Files Provided to You

You will receive the following files:

1. pokemon.input.json

A JSON file containing Pokémon names:

```
{
  "names": ["pikachu", "eevee", "bulbasaur", ...]
}
```

---

2. contracts.ts

A TypeScript file defining the public contracts of the system:

- Domain DTOs (e.g. PokemonDto, Passport)
- Discriminated unions for:
    - Criteria evaluation results
    - Per-item processing results (success / failure)
- Run configuration type
- Run output and report types
- Public interfaces such as:
    - PokemonSource
    - InvestigationRunner

These contracts define:

- What data shapes exist
- What the library API looks like
- What the CLI should ultimately produce

Input

Pokémon Names File

- Provided via pokemon.input.json
- The CLI should accept a path to this file (e.g. --input ./pokemon.input.json)
- File contents should be treated as unknown and validated using Zod
- The structure should be:
    - An object
    - With a names field
    - names is a non-empty array of non-empty strings

Invalid input:

- Should fail fast
- Should not be retried

Data Source

PokeAPI:

https://pokeapi.co - visit the website to test their API

https://pokeapi.co/api/v2/pokemon/{name}

All HTTP calls should be implemented using **Axios**.

Matching Criteria

A Pokémon is considered a match if:

- It has at least one of the following types:
  - Electric
  - Fire
  - Psychic
- It has at least one of the following moves:
  - Thunder Shock
  - Quick Attack
  - Electro Ball
  - Thunder Wave

Processing should stop once the configured minimum number of matches is reached.

Output

The program should produce three JSON files:

1. **Run summary**
   - Processed count
   - Matched count
   - Failed count
   - Duration (ms)
2. **Passports**
   - One object per matched Pokémon (as defined in contracts.ts)
3. **Failures**

    - One entry per failed Pokémon, including:
        - Name
        - Number of attempts
        - Final error message
        - Root cause (if available)

All outputs should be structured and typed.

TypeScript & Design Requirements

1. Strict Typing

- tsconfig should use strict: true
- No any
- No unsafe type assertions to bypass validation
- Domain models should be immutable where appropriate (readonly)

---

2. Zod (Mandatory)

You should implement Zod schemas and actively use them at the following boundaries:

a. Input File Boundary

Validate the structure of pokemon.input.json.

b. HTTP Boundary

Treat axiosResponse.data as unknown and validate it with Zod before mapping it to the DTOs defined in contracts.ts.

c. CLI / Runtime Configuration Boundary

Validate all CLI-derived configuration (defaults, ranges, constraints) using Zod before invoking the library.

Validation errors:

- Are non-retryable
- Should fail fast with clear messages

The review will explicitly check:

- That schemas exist
- That they are actually used
- That no unvalidated unknown flows into the domain layer

---

3. Type-Safe Boundaries

Clear type-safe boundaries should exist between:

- File system input → validated domain types
- Axios HTTP responses → validated DTOs
- CLI layer → library API

After validation, the rest of the system should operate only on typed data.

---

4. Domain Modeling with Unions

Discriminated unions should be used to model:

- Criteria evaluation (matched / not matched with reason)
- Per-item processing (success / failure with metadata)

Avoid boolean flags or nullable fields where a union expresses the state more precisely.

---

5. Concurrency Control

- Requests should be executed with bounded concurrency
- No unbounded Promise.all
- Early termination once minMatches is reached

---

6. Timeout Handling

- Each HTTP request should have a timeout
- Default timeout: 30 seconds
- Timeout should surface as an error with meaningful context

---

7. Retry Policy

A reusable retry utility should be implemented and used.

Retry should:

- Be configurable (attempts, backoff)
- Apply only to transient failures:
    - Network errors
    - Timeouts
    - HTTP 429
    - HTTP 5xx
- Never apply to:
    - Validation errors
    - Parsing errors
    - Non-429 4xx responses

Retry logic should be centralized and not scattered across the codebase.

---

8. Error Handling

Use plain Error, but:

- Preserve root cause (cause where supported)
- Add contextual information (Pokémon name, attempt number, timeout, HTTP status)
- Do not flatten errors into strings
- Do not let a single item failure abort the entire run

Partial failure is expected and should be reported.

CLI Requirements

The CLI should:

- Accept:
    - Input file path
    - Concurrency

    - Timeout (default 30000 ms)
    - Retry count
    - Minimum matches
- Validate all inputs using Zod
- Invoke the library API defined by contracts.ts
- Write the three output JSON files
- Exit with clear, user-friendly error messages

Optional: flags to simulate timeouts / 5xx / 429 responses in order to demonstrate retry behavior.

Code Quality Expectations

The solution will be evaluated on:

- Proper and idiomatic use of TypeScript (unions, generics, narrowing, readonly, inference)
- Clear separation between:
    - I/O
    - Validation
    - Domain logic
    - Orchestration
    - Utilities
- Centralized and well-designed retry and timeout handling
- Correct bounded concurrency behavior
- Readability and maintainability