Architecture Documentation

Architecture

Overview

This solution is composed of two main projects:

1. State CSV Batch Processor (c:\Users\Wictorsama\source\desafio\state-csv-batch-processor)
2. State Aggregator Service (c:\Users\Wictorsama\source\desafio\state-aggregator-service)
   Each project is organized in layers, following best practices for maintainability, scalability, and separation of concerns.

1. State CSV Batch Processor

Purpose

- Reads a CSV file (src/data/phone_data.csv) containing phone records.
- Batches and sends state population data to the aggregator service via HTTP.
  Layered Architecture

a. Entry Point Layer

- File: src/main.ts
- Responsibility:
  - Bootstraps the batch processor app.
  - Configures global pipes and Swagger documentation (if present).
  - Starts the CSV processing workflow.
    b. Application Layer
- File: src/application/services/csv-reader.service.ts
- Responsibility:
  - Provides the CsvReaderService which streams and parses the CSV file.
  - Exposes a method to process each record asynchronously.
  - Handles file reading, parsing, and error management.
    c. Integration Layer
- File: src/main.ts (uses axios)
- Responsibility:
  - Handles HTTP communication with the aggregator service.

    - Sends batches of state population data to the /states/batch endpoint.
      d. Batching/Processing Logic
- File: src/main.ts
- Responsibility:
    - Validates each CSV record.
    - Batches records by a configurable size (BATCH_SIZE).
    - Logs processed and skipped records for monitoring.

2. State Aggregator Service

Purpose

- Receives, aggregates, and stores state population data.
- Exposes REST API endpoints for querying and managing state data.
  Layered Architecture

a. Presentation Layer (Controllers)

- File: src\presentation\controllers\state.controller.ts
- Responsibility:
    - Defines REST API endpoints (e.g., /states/batch , /states/aggregates).
    - Handles HTTP requests and responses.
    - Uses decorators for routing, validation, and Swagger documentation.
      b. Application Layer (Services)
- File: src\application\services\batch-processor.service.ts (and similar)
- Responsibility:
    - Contains business logic for processing batches, aggregating data, and orchestrating operations between controllers and repositories.
      c. Domain Layer (Entities & Interfaces)
- Files:
    - src\domain\entities\state.entity.ts
    - src\domain\repositories\state.repository.interface.ts
- Responsibility:
    - Defines core business models (e.g., State entity).
    - Specifies repository interfaces for data access abstraction.
      d. Infrastructure Layer (Database & Implementations)
- Files:
    - src\infrastructure\database\state.repository.ts
    - src\infrastructure\database\database.module.ts

- Responsibility:
  - Implements data persistence (e.g., MongoDB).
  - Provides concrete repository implementations.
  - Handles database connections and schema definitions.
    e. DTOs and Validation
- Files:
  - src\application\dto\state.dto.ts
- Responsibility:
  - Defines Data Transfer Objects for request/response validation and transformation.
  - Ensures only valid data enters the system.
    f. Queue Layer (Bull + Redis)
- Files:
  - src\app.module.ts (BullModule configuration)
- Responsibility:
  - Uses Bull (a Node.js queue library) with Redis as a backend to manage batch processing jobs.
  - Decouples HTTP request handling from heavy processing, enabling asynchronous and scalable job execution.
  - Allows for monitoring, retries, and distributed processing.

3. Data Flow

1. CSV Processing (State CSV Batch Processor)
   - Reads each row from the CSV.
   - Validates and batches records.
   - Sends batches to the aggregator service.
2. Batch Reception (State Aggregator Service)
   - Receives batch via /states/batch .
   - Enqueues batch jobs in Redis using Bull.
   - Aggregates population by state.
   - Upserts (inserts or updates) state data in the database.
3. Querying Aggregated Data
   - Exposes endpoints like /states/aggregates for clients to retrieve aggregated state population.

4. Example Layer Interaction

- User/Client triggers CSV processing.
- Entry Point Layer (main.ts) starts the process.

- Application Layer (CsvReaderService) streams the CSV.
- Integration Layer (axios) sends data to the API.
- Presentation Layer (Controller) receives the batch.
- Application Layer (Service) processes and aggregates.
- Queue Layer (Bull + Redis) manages background jobs.
- Domain Layer (Entity/Interface) models the data.
- Infrastructure Layer (Repository) persists the data.

5. Benefits of This Architecture

- Separation of Concerns: Each layer has a clear responsibility.
- Testability: Business logic and data access are decoupled, making unit testing easier.
- Scalability: New features or integrations can be added with minimal impact on other layers.
- Maintainability: Code is organized and modular, facilitating easier updates and debugging.
- Asynchronous Processing & Reliability: Redis-backed queues ensure that heavy workloads do not block API requests and can be processed efficiently in the background, supporting retries and distributed processing.
  This architecture ensures each part of the system is focused, maintainable, scalable, and robust for production workloads, with Redis playing a key role in enabling reliable and scalable batch processing.