Legacy Customer Data Import - Take-Home Test

⚠ **CRITICAL NOTICE :**
This is a **production-ready assessment**. We are NOT looking for TODO lists in your README. We expect a **fully functional, runnable application** that we can test immediately.

Objective

Your client is migrating customer records from a legacy CRM. You are tasked with building a **production-grade** backend system to handle CSV imports. We are looking for more than just "working" code; we need a system designed for **resilience, scalability, and high availability**.

File Format

The client will upload a CSV file with the following structure:

```
full_name,email,date_of_birth,timezone
John Doe,john@example.com,1990-05-15,America/New_York
Jane Smith,jane@example.com,1985-08-22,Europe/London
```

Each row represents one customer record.

Technology Stack

- **Backend:** Node.js with Express
- **Database:** MongoDB
- **Testing:** Jest or Mocha (or equivalent)
- **Containerization:** Docker & Docker Compose

Core Requirements

1. CSV Upload & Processing API

- Provide a REST API endpoint to upload CSV files
- Parse and validate each record according to the rules below
- Process files asynchronously for scalability (use job queues, workers, etc.)
- **RECOMMENDED** Handle large files efficiently (streaming, batch processing)

2. Validation Rules

Field	Validation Rule
`full_name`	Required, non-empty string
`email`	Valid email format, must be unique in the database
`date_of_birth`	Valid ISO 8601 date string, must be in the past
`timezone`	Valid IANA timezone identifier (e.g., "America/New_York")

3. Data Storage

- Store all valid records in MongoDB
- Implement proper indexing (especially on email for uniqueness)
- Track import job metadata (job ID, status, timestamps, results)

4. Import Results API

Provide an endpoint to retrieve import results with the following information:

- Total number of records processed
- Number of successfully imported records
- Number of rejected records
- Detailed list of rejected records with specific error messages
- Import job status (pending, processing, completed, failed)

5. User Management APIs

Implement CRUD operations for customer records:

- **GET** `/api/customers/:id` - Retrieve customer details by ID
- **PUT/PATCH** `/api/customers/:id` - Update customer details (with validation)
- **DELETE** `/api/customers/:id` - Delete a customer
- **GET** `/api/customers` - List customers (with pagination)

Production-Readiness Requirements

1. Docker Configuration

- Provide a `docker-compose.yml` that sets up the entire application stack
- Must include: Node.js application, MongoDB, and any other required services
- Application must start with a single command: `docker-compose up`
- Database must be initialized automatically (no manual setup required)

2. Environment Configuration

- Provide `.env.example` with all required environment variables
- Use environment variables for all configuration (database URLs, ports, etc.)
- Include clear comments explaining each variable

3. Error Handling & Logging

- Implement comprehensive error handling for all endpoints
- Use proper HTTP status codes
- Return consistent error response format
- Implement structured logging
- Log important events: requests, errors, import jobs, validation failures

4. Testing

- Unit tests for validation logic, services, and utilities
- Integration tests for API endpoints
- Tests must be runnable with a single command (e.g., `npm test`)

5. API Documentation

- Provide clear documentation
- Include example requests and responses for all endpoints
- Document all query parameters, request bodies, and response formats
- Include error response examples

6. Database Design

- Implement proper schema design with Mongoose models
- Add appropriate indexes for performance
- Implement email uniqueness at the database level
- Use transactions where appropriate (for data consistency)

Code Quality Expectations

- **Clean Code:** Follow consistent naming conventions, proper code structure, and best practices
- **Modular Design:** Separate concerns (routes, controllers, services, models, validators)
- **DRY Principle:** Avoid code duplication

- **Input Validation:** Validate and sanitize all user inputs
- **Security:** Implement basic security measures

README Requirements

Your README must contain actual instructions, NOT a list of things you "plan to do" or "would implement in production."

Your README.md must include:

1. **Quick Start:** Commands to get the application running (should just be `docker-compose up`)
2. **Prerequisites:** What needs to be installed (Docker, Docker Compose)
3. **API Usage Examples:** Actual curl/HTTP examples for each endpoint
4. **Testing:** How to run tests and check coverage
5. **Project Structure:** Brief overview of directory organization
6. **Design Decisions:** Why you chose certain libraries, patterns, or approaches
7. **Assumptions & Limitations:** Any assumptions made or known limitations
8. **Future Improvements:** What you would add given more time (this is the ONLY place TODO items are acceptable)

Evaluation Criteria

Your submission will be evaluated on the following criteria (in order of importance):

1. **Completeness (30%):** Does it meet all requirements? Can we run it immediately?
2. **Code Quality (25%):** Is the code clean, maintainable, and well-structured?
3. **Production-Readiness (20%):** Error handling, logging, Docker setup, async processing
4. **Testing (15%):** Test coverage, quality of tests, edge case handling
5. **Documentation (10%):** Clarity of README, API docs, code comments

Submission Guidelines

- Submit via GitHub/GitLab repository
- Do NOT include `node_modules` or any build artifacts
- Do include `.env.example` but NOT `.env` with actual credentials
- Ensure your repository is public or grant access to reviewers

Questions?

If you have questions about the requirements, please ask before starting. We prefer clarification upfront rather than assumptions that lead to missing requirements.

Good luck! We're excited to see your solution. .