# AXIEVA PROJECT

# In-Memory Student Search - Project Documentation

## Problem Statement

Design and implement a Spring Boot-based RESTful service that reads a large dataset (in CSV format) containing student details, stores the data in memory, creates an index based on student names to optimize search operations, and exposes APIs to create the index and search for students by name.

## 1. Introduction

The In-Memory Student Search project is a high-performance search system built using Spring Boot. It loads student data from a CSV file into memory, builds an index for fast lookups, and provides REST APIs to search for students by exact name or prefix. The project demonstrates both linear and index-based searching, with optional multi-threaded CSV parsing.

## 2. Architecture

The system follows a simple layered architecture:  
- Controller Layer: Handles HTTP requests and maps them to service methods.  
- Service Layer: Contains the business logic for loading, indexing, and searching student data.  
- Model Layer: Defines the Student entity used for data storage.  
- Indexing: Uses a ConcurrentHashMap to store lowercase student names as keys for fast retrieval.  
- Parallel Processing: Utilizes Java ForkJoinPool for multi-threaded CSV parsing.

## 3. Project Structure

student-search-inmemory/  
 ├── src/main/java/com/example/inmemory/controller  
 ├── src/main/java/com/example/inmemory/service  
 ├── src/main/java/com/example/inmemory/model  
 ├── src/main/resources/students.csv  
 └── pom.xml

## 4. Dataset

The dataset is provided as a CSV file (`students.csv`) with the following columns generated using python script using random madule:  
ID, Name, Age, Class, Grade.  
Example row: `1, Grace Patel, 16, 10A, A`.  
The system can also load CSV files from an absolute path provided by the user.

## 5. Algorithms Used for Searching

- **Linear Search**: Iterates through the entire list of students to find matches (O(n) time complexity).  
**- Index Search**: Uses a pre-built ConcurrentHashMap to look up students by lowercase exact name (O(1) time complexity).  
- **Prefix Search:** Scans index keys to find matches starting with the given prefix (O(k) where k is number of unique names).

## 6. API Endpoints

1. POST /index - Build index from CSV file.  
 - Params: `csvPath` (optional)  
 - Response: Stats about loading and indexing.  
  
2. GET /search - Search students by exact name.  
 - Params: `name` (required), `mode` (linear|exact)  
 - Response: List of matching students.  
  
3. GET /search-prefix - Search students by name prefix.  
 - Params: `prefix` (required)  
 - Response: List of matching students.  
  
4. GET /debug-index - Return the complete index map (for debugging).

## 7. Learnings

- Spring Boot Basics: Learned how to set up a Spring Boot application, define REST controllers, and use dependency injection.  
- Maven: Understood how Maven builds and manages dependencies.  
- Multi-threading in Java: Used ForkJoinPool to parallelize CSV parsing.  
- Data Structures: Learned how to use ConcurrentHashMap for thread-safe indexing.  
- Algorithm Complexity: Compared linear search and hash-based index search.  
- REST API Design: Learned how to create endpoints, handle query parameters, and return JSON responses.  
- File I/O in Java: Learned how to read CSV files from both classpath and absolute paths.  
- OpenCSV Library: Used it for parsing CSV data.  
- Indexing Concepts: Understood how pre-processing data can significantly improve query performance.

## 8. Conclusion

This project showcases a practical implementation of an in-memory search engine using Spring Boot. It highlights the trade-offs between different search algorithms, demonstrates the performance benefits of indexing, and serves as a learning project for multi-threading, REST API development, and Java data structures.

Steps to Use :

Build the full project

mvn clean package

This generates a JAR in target/ (for example student-search-inmemory-0.0.1-SNAPSHOT.jar).

### Run the JAR

java -jar target/student-search-inmemory-0.0.1-SNAPSHOT.jar

### Open the app in browser

* Visit: http://localhost:8080

You’ll see the full frontend + backend served from the same place.

### Use the App

* If you **leave CSV path empty**, it uses the default students.csv in resources.
* If you **enter a full local file path**, it reads that CSV instead.
* Click **Build Index** to load and index the data.

### Step A — Build Index

1. Open http://localhost:3000.
2. If you **want to use default CSV** (bundled in backend), leave the **Optional: absolute CSV file path** field empty.
3. If you **want to use your own CSV**, enter its **full file path** (e.g., D:\Data\my\_students.csv).
4. Click **"Build Index"**.
   * The backend reads CSV, parses it, builds **in-memory index** (multi-threaded).
   * You’ll see stats like:
   * Rows: 100000, Unique names: 100, Parse: 54 ms, Index: 33 ms

### Step B — Search Students

1. Enter a name or prefix in the search box.
2. Choose **mode**:
   * **linear** → Scans all records (slower for big data).
   * **index** → Uses prebuilt in-memory map (fast).
3. Choose **match type**:
   * **exact** → Name must match exactly.
   * **prefix** → Name must start with your search text.
4. Click **Search**.
5. Results table will display matching students.