User Data Retrieval API

Project Overview

The User Data Retrieval API is designed to provide flexible and dynamic access to user data. Unlike traditional APIs with hard-coded fields, this API utilizes a flexible schema that allows for customized querying and filtering based on user requirements.

Key Features

1. **Flexible Schema:** The user search API uses a JSON based schema which defines attributes available for search and configuration which allow us to define the field and configuration without making changes to the codebase.
2. **Advanced Filtering**: Users can apply filters to retrieve data based on multiple criteria, including support for range queries.
3. **Configurable Fields**: Users can specify which fields to include in the response, optimizing the data returned.
4. **Sorting**: The API supports sorting of records. Users can specify the order in which records should be sorted.
5. **Error Handling**: Meaningful error messages and status codes ensure users receive clear feedback on the requested data.
6. **Limits**: Users can specify a limit on the number of records returned, which provides flexibility for pagination.

Technology Stack

Backend: Node.js, Express

Database: MongoDB

Test: Jest

Documentation: Swagger

Code Coverage: Jest coverage

Project Explanation

This project involves creating an API that allows users to retrieve data about individuals with a high degree of flexibility. The API is designed to work with a schema that can adapt to different data structures without requiring code changes. This flexibility ensures that the API can handle a variety of data formats and requirements.

Key Components

- **Schema File (user_schema.json)**This file defines the structure of the data that the API can handle. The schema which has been used in building this service includes various fields such as userId, courseId, location, and progress. Each field has specific rules, such as data type and whether it is required or not.

- Configuration (config):

  The config section of the schema file specifies two important things:

  - **Fields to be Returned in the Response:**
    - This defines which fields should be included in the API response by default. If a user does not specify which fields they want, the API will return the fields listed in this configuration.
    - Example: userId, courseId, location, progress, and batchEnrollmentDate.
  - **Fields Allowed to be in Range:**
    - This specifies which fields can be filtered using a range (e.g., values between a minimum and maximum). Only fields listed here can be used with range-based queries.
    - Example: progress, batchEnrollmentDate

Sample Schema File

```
{
 "schema": {
   "type": "object",
   "properties": {
     "_id": {
       "type": "string"
     },
     "courseId": {
       "type": "string",
       "required": true
     },
     "batchId": {
       "type": "string",
       "required": true
     },
     "userId": {
       "type": "string",
       "required": true
     },
     "location": {
       "type": "string",
       "required": true
     },
                                    ## add other fields here
     "progress": {
       "type": "number",
       "required": true
     },
     "batchEnrollmentDate": {
       "type": "date"
     },
     "createdAt": {
       "type": "date"
```

```
    },
    "updatedAt": {
      "type": "date"
    }
  },
  "config": {
    "response_fields_to_be_returned": [
      "userId courseId location progress batchEnrollmentDate"
    ],
    "fields_allowed_to_be_in_range": ["progress", "batchEnrollmentDate"]
  }
 }
}
```

Developer setup on local machine

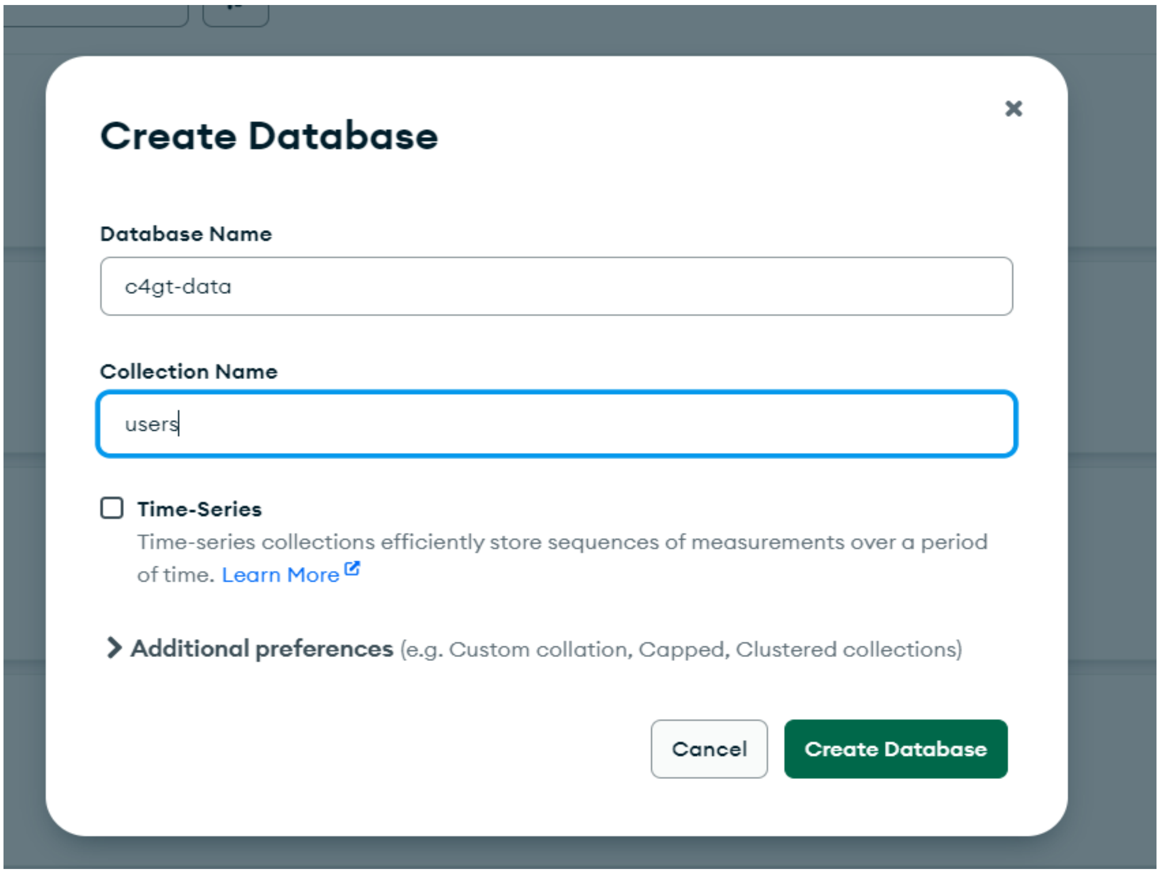
To run the server on your local machine, follow these steps:

1. **Install Node.js:** Ensure that Node.js is installed on your machine. If you haven't installed it yet, you can follow the official Node.js installation guide.
2. **Ensure MongoDB is installed and running on your local machine.** If you haven't installed it yet, you can follow the official MongoDB installation guide.
3. **Clone the GitHub Repository.** Open the terminal in the directory you want to clone the project

```
git clone https://github.com/iGOT-MissionKarmayogi/C4GT-Communications-Console
```

4. **DB setup:**
   - Using MongoDB Compass:
     - Open MongoDB Compass.

- Create a new database named **c4gt-data** and a collection named **users**.



Create Database

Database Name

c4gt-data

Collection Name

users

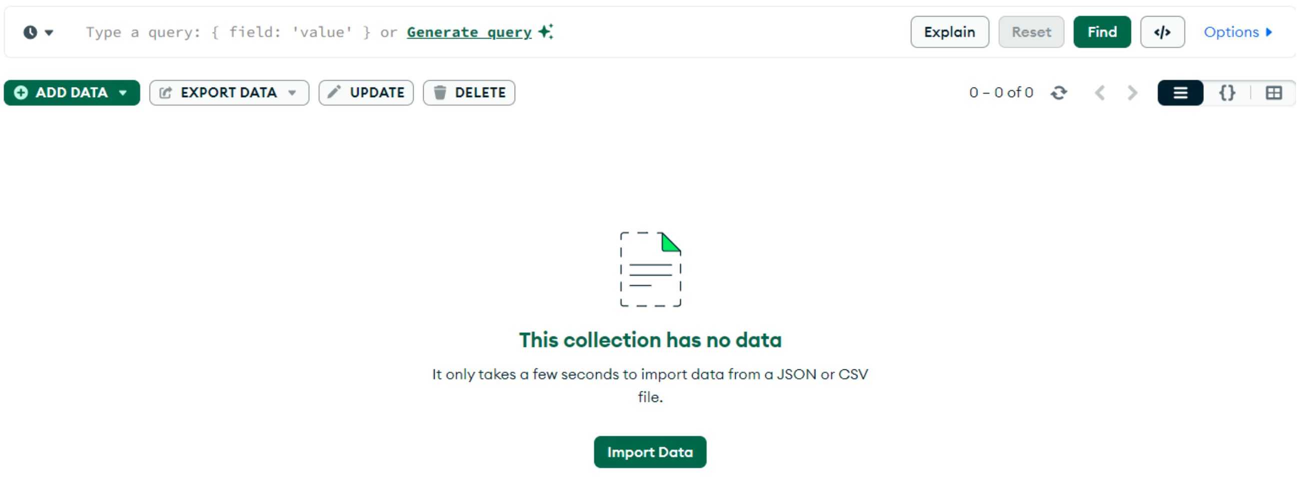
☐ Time-Series

Time-series collections efficiently store sequences of measurements over a period of time. Learn More

> Additional preferences (e.g. Custom collation, Capped, Clustered collections)
Cancel
Create Database

- Import data from the data.json file present in the sample-data folder.

`


Type a query: { field: 'value' } or Generate query

This collection has no data

It only takes a few seconds to import data from a JSON or CSV file.

Steps to import data into MongoDB using Docker

1. **Ensure MongoDB is Running**:
2. Create a file named mongo.yml and add the following content:

```
version: '3.1'
services:
  mongo:
    image: mongo:latest
    restart: always
    ports:
     - 27017:27017
    volumes:
     - $HOME/c4gt/c4gt_db:/data/db
    container_name: mongo_c4gt
```

   - Start your MongoDB container using the following command:

     ```
     docker-compose -f mongo.yml  up -d
     ```

3. **Open MongoDB Compass**:
   - Launch MongoDB Compass to connect to your MongoDB instance.
4. **Connect to MongoDB**:
   - Use the connection string mongodb://localhost:27017 to connect.
5. **Create Database and Collection**:
   - Create a database named c4gt-data.
   - Inside that database, create a collection named users.
6. **Import Data from JSON File**:
   - In MongoDB Compass, navigate to the users collection.
   - Click on the **Collection** tab, then select **Import Data**.
   - Choose the **data.json** file from the sample-data folder on your local machine.

5. Checkout the following branch:

   ```
   git  checkout user-search
   ```

6. **Install Dependencies:** Navigate to your project directory in the terminal and install the necessary dependencies using npm:

   ```
   npm install
   ```

7. **Configure Environment Variables:**
   - Create a .env file in the root directory of your project if it doesn't already exist.
   - Add the following environment variables to configure the server:

```
APPLICATION_PORT=3000
APPLICATION_ENV=development
MONGODB_URL=mongodb://localhost:27017/c4gt-data
SCHEMA_PATH=../schema/user_schema.json
```

APPLICATION_PORT: The port on which the server will run. Default is 3000.

APPLICATION_ENV: The environment in which the application is running. Set to development for local development.

MONGODB_URL: The URL for connecting to the MongoDB instance. Ensure MongoDB is running locally on port 27017.

SCHEMA_PATH: You can configure the SCHEMA_PATH in two ways, depending on your preference

1. **Local Schema File**: You can set the path to a local schema file that the application will use. This is useful when the schema is stored within your project directory.

```
SCHEMA_PATH=../schema/user_schema.json
```

2. **Cloud URL Schema:** Alternatively, you can point to a schema file hosted on a cloud URL.

```
SCHEMA_PATH=Put URL here
```

8. **Run the Server:** Start the server with the following command:

```
npm start
```

This will launch the server on http://localhost:3000 (or the port specified in APPLICATION_PORT).

Libraries

Here are the key libraries and dependencies used in the project:

- **axios (^1.7.2):** A promise-based HTTP client for making requests to external APIs and handling responses.
- **body-parser (^1.20.2):** Middleware for parsing incoming request bodies in a middleware before your handlers, available under the req.body property.
- **cors (^2.8.5):** Middleware to enable Cross-Origin Resource Sharing, allowing the server to handle requests from different origins.
- **dotenv (^16.4.5):** A module to load environment variables from a .env file into process.env, keeping configurations separate from code.
- **express (^4.19.2):** The main web framework for building RESTful APIs and handling HTTP requests.
- **fs (^0.0.1-security):** A Node.js built-in module to interact with the file system, used here for file operations like reading schemas.
- **mongoose (^8.4.4):** An Object Data Modeling (ODM) library for MongoDB and Node.js, providing a straightforward schema-based solution to model data..
- **nodemon (^3.1.4):** A utility that monitors for any changes in your source and automatically restarts your server, making development faster.
- **path (^0.12.7):** A Node.js built-in module that provides utilities for working with file and directory paths.
- **swagger-autogen (^2.23.7):** A tool to automatically generate Swagger/OpenAPI documentation from your codebase.
- **swagger-ui-express (^5.0.1):** A tool for serving auto-generated Swagger UI from your Express applications.
- **url (^0.11.3):** A Node.js built-in module to handle and parse URLs.

- **uuid (^10.0.0):** A library for generating RFC-compliant UUIDs (Universally Unique Identifiers) for identifying resources uniquely.

API Endpoint Details

HTTP Method	**URL Path**	**Description**	**Controller**
POST	/user/v1/search	Allows users to search based on specified criteria.	UserSearchController.userSearch
GET	/health	Performs a health check to ensure the service is running.	HealthCheckController.healthCheck

Swagger API Documentation

The Swagger API documentation for this project is available at the following route:
/doc
You can access the Swagger UI interface by navigating to `http://localhost:<port>/doc` in your web browser, where `<port>` is the port number your Express server is running on (e.g., 3000).

This interface allows you to explore the available API endpoints, view detailed request/response schemas, and even test the API directly from the browser.

Folder Structure Overview

```
│
├── api-collection/
│   └── user-search-api.postman_collection.json
│
├── controllers/
│   ├── healthCheckController.js
│   └── UserSearchController.js
│
├── coverage/
│
├── docs/
│   └── swagger-output.json
│
├── models/
│   └── userModel.js
│
├── node_modules/
│
├── routes/
│   ├── healthCheckRoute.js
│   └── userSearchRoutes.js
│
├── schema/
│   └── user_schema.json
│
├── tests/
│   └── user.test.js
│
├── utils/
│   ├── appErrors.js
│   ├── errorResponse.js
│   └── routerManager.js
│
├── .env
├── .env.sample
├── .gitignore
├── Dockerfile
├── jest.config.js
├── package.json
├── package-lock.json
├── README.md
├── server.js
└── swagger.js
```

Project Folder Structure Explanation

This section provides an overview of the key directories and files in the project, detailing their roles and how they contribute to the overall functionality of the application.

1. controllers/

- **healthCheckController.js**: Manages the health check endpoint, ensuring the application is running smoothly.
- **UserSearchController.js**: Contains the logic for handling user search requests.

2. routes/

- **healthCheckRoute.js**: Defines the API route for the health check endpoint.
- **userSearchRoutes.js**: Defines the API routes for user search operations.

3. models/

- **userModel.js**: Represents the user data structure using Mongoose schema, defining how user information is stored in the database.

4. utils/

- **appErrors.js**: Handles custom error definitions used throughout the application.
- **errorResponse.js**: Manages the format of error responses sent back to clients.
- **routerManager.js**: Contains helper functions for managing routes.

5. tests/

- **user.test.js**: Contains test cases for verifying the functionality of user search related operations.

6. schema/

- **user_schema.json**: Defines the JSON schema for user data, ensuring consistency and validation.

7. docs/

- **swagger-output.json**: Stores the API documentation using Swagger.

8. api-collection/

- **user-search-api.postman_collection.json**: A collection of API requests for testing the user search functionality using Postman.

9. coverage/

- A directory that will contain the code coverage reports generated after running tests, helping to ensure all parts of the code are properly tested.

10. Root-Level Files

- **.env** and **.env.sample**: Store environment variables for configuring the application.
- **Dockerfile**: Contains the instructions for building a Docker image of the application.
- **jest.config.js**: Configuration file for setting up the Jest testing framework.
- **server.js**: The main entry point of the application where the server is initialized.
- **swagger.js**: Sets up the Swagger middleware for API documentation.