resolver.js

```
import jwt from 'jsonwebtoken';
import { getData, getUsers, getdataMap, addData } from "./database.js";

// Load data and users from the database
let data = getData();
let users = getUsers();

const resolvers = {
  Query: {
    // Fetch all data only for authenticated users
    getAllData: (parent, args, context) => {
      if (!context.user) {
        throw new Error('Not authenticated'); // Ensure authentication
      }
      return data; // Return all data
    },

    // Fetch a specific record by ID, restricted to authenticated users
    getDatabyId: (parent, args, context) => {
      if (!context.user) {
        throw new Error('Not authenticated'); // Ensure authentication
      }
      return data.find(p => p.id === args.id); // Find and return the data by ID
    },

    // Fetch user-specific data based on their username
    getUserData: (parent, args) => {
      const dataMap = getdataMap(); // Map of users to their data IDs
      const userId = dataMap[args.username]; // Get data IDs for the user
      if (!userId) return []; // Return empty array if no data IDs are found
      return data.filter(person => userId.includes(person.id)); // Return user's specific data
    },

    // Fetch all users (This may not be safe in production without restrictions)
    getUsers: () => users,
  },

  // Additional resolver for User type (if implemented in schema)
  User: {
    // Fetch data owned by a specific user
    userOwnData: (parent) => {
      const dataMap = getdataMap(); // Map of users to their data IDs
      const userId = dataMap[parent.username]; // Get data IDs for the user
      if (!userId) return []; // Return empty array if no data IDs are found
      return data.filter(person => userId.includes(parent.id)); // Return user's specific data
    }
  },

  Mutation: {
    // Add new data entry to the database
    addData: (parent, args, context) => {
      if (!context.user) {
        throw new Error('Not authenticated'); // Ensure authentication
      }

      // Check if data with the same ID already exists
      if (data.find(b => b.id === args.id)) {
        throw new Error('Record already exists'); // Prevent duplicate records
      } else {
        const newData = { ...args }; // Create new data object
        addData(newData); // Persist new data to the database
        data = data.concat(newData); // Update in-memory data
        return newData; // Return the added data
      }
    },

    // Authenticate a user and provide a JWT token
    login: (parent, { username, password }) => {
      // Verify user credentials
      const user = users.find(user => user.username === username && user.password ===
password);
      if (!user) throw new Error('Invalid credentials'); // Invalid credentials

      // Generate a JWT token
      const token = jwt.sign({ username: username }, 'my_secret_key', { expiresIn: '1d' });
      const bearer_token = 'Bearer ' + token;

      // Save token to the user object (non-persistent, temporary storage)
      user.token = token;

      return { "token": bearer_token, username }; // Return token and username
    }
  }
};

export default resolvers;
```

database.js

This code manages a simple JSON-based database using fs for file I/O, allowing persistent storage of users, data, and their associations.

```
import fs from 'fs';
```

```
const DATABASE_FILE = './database.json'; // Path to the JSON database file

// Initialise database content
const initialData = {
    users: [
        { "username": "jk", "password": "sala", 'token': '', "rateLimiting": { "window": 0,
"requestCounter": 0 } },
        { "username": "pl", "password": "pass", 'token': '', "rateLimiting": { "window": 0,
"requestCounter": 0 } }
    ],
    data: [
        { "id": "1", "Firstname": "Jyri", "Surname": "Kemppainen" },
        { "id": "2", "Firstname": "Petri", "Surname": "Laitinen" },
        { "id": "3", "Firstname": "Heikki", "Surname": "Helppo" }
    ],
    dataMap: {
        jk: ["1", "3"], // "jk" owns data with IDs 1 and 3
        pl: ["2"]       // "pl" owns data with ID 2
    }
};

// Loads data from the database file or initialize if missing
const loadDatabase = () => {
    if (fs.existsSync(DATABASE_FILE)) { // Check if the database file exists
        try {
            return JSON.parse(fs.readFileSync(DATABASE_FILE, 'utf8')); // Parse and return
file content
        } catch (error) {
            console.error('Error reading database file:', error);
        }
    }
    saveDatabase(initialData); // If file is missing or invalid, initialize with default data
    return initialData;
};

// Save in-memory database to the file
const saveDatabase = (data) => {
    fs.writeFileSync(DATABASE_FILE, JSON.stringify(data, null, 2), 'utf8'); // Write JSON
data with 2-space indentation
};

// Loads the database into memory on startup
let db = loadDatabase();

// Gets all users
const getUsers = () => {
    return db.users; // Return users from the in-memory database
};

// Gets all data records
```

```
const getData = () => {
   return db.data; // Return data from the in-memory database
};

// Here it retrieves the user-to-data mapping
const getdataMap = () => {
   return db.dataMap; // Return dataMap from the in-memory database
};

// This adds a new data record and save it persistently
const addData = (newData) => {
   db.data.push(newData); // Add new record to in-memory data array
   saveDatabase(db); // Includes the updated database to the file
};

// This updates the user information (e.g., token or rate limiting)
const updateUser = (username, userUpdates) => {
   const userIndex = db.users.findIndex(user => user.username === username); // This
searches for user by username
   if (userIndex >= 0) {
      db.users[userIndex] = { ...db.users[userIndex], ...userUpdates }; // Combines updates
with existing user
      saveDatabase(db); // Includes the updated database to the file
   }
};

export {
   getUsers,    // Exports function to retrieve users
   getData,     // Exports function to retrieve data
   getdataMap,  // Exports function to retrieve the user-to-data mapping
   addData,     // Exports function to add new data
   updateUser   // Exports function to update user details
};
```

Functionality of the Code

1. Initial Content of the Database:
The initialData object includes the users' default data, the records (data) that are connected with them, and a mapping (dataMap) that shows which records belong to which users.

2. Management of Database Files:
 DATABASE_FILE designates the database.json JSON file that is used to store the database.
 At startup, the system tries to load the database from the file. It initialises the file with initialData if it doesn't already exist.

3. Fundamental Tasks:
 loadDatabase: Utilises database.json to retrieve data. It initialises with initialData if the file is invalid or missing.
 saveDatabase: Creates database.json with the current in-memory database.
 getUsers: Provides the in-memory database's user list.

getData: Provides the data record list.
getdataMap: Provides the mapping between users and their data.
addData: Permanently stores a new entry in the database.
updateUser: Modifies a particular user's information (such as token or rate-limiting data) and keeps the modification in effect.

4. Persistent Storage: All modifications (such as addData and updateUser) update the database and store the modifications to the in-memory database (db).JSON.