Weather API

src/index.ts

This is the entry point for our app.

First we connect to the mongodb database.

Then we start on the express http server.

src/app.ts

This is the root of our express app

It initializes middleware for error handling and rate limiting.

Also handles controllers.

src/controller.ts

Contains the interface for a controller.

A controller can have a path and an express.Router

src/exception.ts

This contains our custom exception class for error handling in REST api.

This is then later caught by error middleware.

src/seed Data.ts

This contains code for seeding databases.

It uses faker.js to generate city names.

Then randomly set temperature, precipitation and windSpeed values using randInt function.

src/model/weather.model.ts

This contains the mongoose model for our weather data.Each entry has cityName, temperature, windSpeed and precipitation.

The `WeatherController` class contains five methods:
//
// 1. `get_city_names`: It gets the `page` parameter from the request query or uses the default value of 0.
// It then retrieves a maximum of 10 weather data entries from the database, skips entries based on the page number,
// and returns the `cityName` property of the entries as a JSON response.
//
// 2. `add_weather`: It creates a new `WeatherModel` instance with the request body data, saves the data to the database, and returns the saved data as a JSON response.
//
// 3. `get_weather`: It retrieves the weather data of a specific city from the database based on the `cityName` parameter in the request URL.
// If no data is found, it throws an exception with a 404 status code. Otherwise, it returns the retrieved data as a JSON response.
//
// 4. `update_weather`: It creates a new `WeatherModel` instance with the `cityName` parameter from the request URL and the request body data.
// It then validates the data, updates the weather data of the specified city in the database, and returns a JSON response with a success message.
//
// 5. `delete_weather`: It deletes the weather data of a specific city from the database based on the `cityName` parameter in the request URL.
// If no data is found, it throws an exception with a 404 status code. Otherwise, it returns a JSON response with a success message.
//
// The `WeatherController` class constructor sets up the various routes for the class methods using the `express.Router()` method.
// The `get_weather`, `update_weather`, and `delete_weather` methods all have `:cityName` as part of their URL, which indicates that they expect a city name to be provided as a URL parameter.
// The `add_weather` method expects weather data to be provided in the request body, and the `get_city_names` method does not expect any parameters.
//
// In summary, the `WeatherController` class provides a RESTful API for creating, reading, updating, and deleting weather data for various cities.

src/middleware/rateLimiter.middleware.ts

This code implements rate limiting middleware for an Express application, to limit the number of requests a given IP address can make in a given period of time.

The function first checks the IP address of the incoming request by accessing the `req.socket.remoteAddress` property. If the IP address cannot be retrieved, the function returns an error message by calling the `next` function with a new `Exception` object.

The function then checks if the IP address has already made requests by looking up the IP address in a `Map` called `requestMap`. If the IP address exists in the `Map`, the function checks the timestamp of the oldest request and compares it to the current time. If the difference is less than one hour and the number of requests made is greater than the `rate` limit (which is set to 100 in this case), the function returns an error message by calling the `next` function with a new `Exception` object.

if the difference is greater than one hour, the function removes the oldest request timestamp from the `timeStamps` array. Then it pushes the current timestamp to the `timeStamps` array to keep track of the incoming request.

If the IP address does not exist in the `Map`, the function adds the IP address to the `Map` with the current timestamp as the first request timestamp.

Finally, if the request does not exceed the rate limit, the function calls the `next` function to move on to the next middleware function.

Overall, this code helps to prevent a single IP address from making too many requests to the server, which could potentially cause performance issues or deny service to other users.