**REST API and NoSQL**

Documentation

Created by

**UPTOWN IT**

For

**<<CUSTOMER>>**

**PROJECT REFERENCE:**

**DATE:**

**CONTENTS**

[PART ONE – NoSQL Solution 2](#_Toc131370449)

[NoSQL research and technology selection 2](#_Toc131370450)

[Reasoning for using NoSQL 2](#_Toc131370451)

[Benefits of using NoSQL 2](#_Toc131370452)

[Relational Database vs Non-relational Database 2](#_Toc131370453)

[Comparison of NoSQL solutions 3](#_Toc131370454)

[MongoDB setup 4](#_Toc131370455)

[Schema and dataset 5](#_Toc131370456)

[Partitioning 5](#_Toc131370457)

[Indexing and TTL in NoSQL 7](#_Toc131370458)

[Indexes 7](#_Toc131370459)

[Creating special indexes 8](#_Toc131370460)

[PART TWO – REST API 10](#_Toc131370461)

[Build and document REST API Project 10](#_Toc131370462)

[Review Project Requirements 10](#_Toc131370463)

[Selecting REST API framework and IDE 11](#_Toc131370464)

[Build the REST API: Endpoints and methods 12](#_Toc131370465)

[Test methods 13](#_Toc131370466)

[Tests Conducted 13](#_Toc131370467)

[Tests after bug fixes 22](#_Toc131370468)

[Model Validation 23](#_Toc131370469)

[Current Validation Rules 24](#_Toc131370470)

[AppUserBase 24](#_Toc131370471)

[AppuserCreateManyDTO 24](#_Toc131370472)

[AppuserFilter 24](#_Toc131370473)

[AppuserUpdateManyDTO 24](#_Toc131370474)

[WeatherReadingBase 24](#_Toc131370475)

[WeatherReadingFilter 25](#_Toc131370476)

[WeatherReadingUpdateDTO 25](#_Toc131370477)

[WeatherReadingUpdateManyDTO 25](#_Toc131370478)

[WeatherReading 26](#_Toc131370479)

[Complex Model Validation Testing 26](#_Toc131370480)

[Enable Cross-origin Resource Sharing (CORS) 28](#_Toc131370481)

[Pre-Flight Testing 30](#_Toc131370482)

[Evaluate and secure REST API 35](#_Toc131370483)

[How it works 36](#_Toc131370484)

[Documenting REST API endpoints 39](#_Toc131370485)

[PART THREE – REST API and MongoDB 41](#_Toc131370486)

[REST API and MongoDB Integration 41](#_Toc131370487)

[Further queries or object interactions 44](#_Toc131370488)

[Implementation of triggers 46](#_Toc131370489)

[Logging trigger 46](#_Toc131370490)

[Validation trigger 47](#_Toc131370491)

[Data security 49](#_Toc131370492)

[Finalising the project 52](#_Toc131370493)

# PART ONE – NoSQL Solution

# NoSQL research and technology selection

### Reasoning for using NoSQL

1. The amount of data. NoSQL isn’t relational. This particular data does not need to be relational as the reading data contains the all data linked to that reading. E.g location, time… relational databases would use more storage and don’t scale well.
2. Scaling. While there are SQL based database solutions that have sharding, NoSQL does this better. This project want horizonal scaling (More database servers) instead of vertical scaling (Upgrading a single database server). This is better in terms of cost and data safety as there isn’t one point of failure.
3. Speed. This project calls for fast write speeds as data coming in is frequent but the data is infrequently read.

### Benefits of using NoSQL

1. Data safety. Horizontal scaling allows for data to be spread across multiple servers. This means that if one server went down you would still have access to all other servers and their data.
2. Cost. When you buy more machines you don’t need them to have top of the line hardware. However if you were using vertical scaling you would have to upgrade the machine when you want more out of your database server. This gets more expensive the higher the spec. This creates a lot of waste as you will be wasting old hardware

### Relational Database vs Non-relational Database

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TYPE of DATABASE | SUITABILITY by DATA VOLUME | SUITABILITY by DATA TYPE  Structured, Semi-Structured, Unstructured | QUERY COMPLEXITY | ACID COMPLIANCE |
| Relational: SQL | Acceptable speed and acceptable storage efficiency | structured | Technically human readable but difficult with complexity | Fully acid compliant |
| Non-relational:NoSQL | Efficient and fast | Unstructured  Semi-Structured | Defined in code. Difficulty remains the same with complexity | NoSql uses cap theory and is not acid compliant |

### Comparison of NoSQL solutions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NoSQL Interface | MAIN FEATURES | DATA FORMATS | EASE OF USE | BEST FOR  Types of Projects |
| Redis | * Stupidly fast read/write * In memory db * Failure tolerant * Time to live * transactions * Open source | Strings,  numbers (as strings),  Lists,  Sets,  Hashes,  Sorted sets,  Streams,  More… | Fairly easy to use  With good documentation. | Real time data read/write. Such as hosting a messaging service |
| Cassandra | * Reliability * Highly scalable * Fast read/write * Quality tested * Owned by Apache * Open source | Strings,  Numbers,  Booleans,  Date/time,  Precise duration,  Uuid,  More… | Easy to use with great documentation. Good community support. | Projects that require reliability, scalability and speed. |
| CouchDB | * Reliability with crash resistant data storage * Low system requirements * support for clusters * supports both NoSQL and Relational models * document store and not key value. | Strings,  Numbers,  Booleans,  Arrays,  Objects,  Null,  More… | Easy to understand documentation and simple to setup. Good community support. | projects that require reliability, uptime and low cost in a professional business environment. |
| MongoDB | * highly scalable * Fast read/write * document store using Bson. * Sharding * Performance profiling * Time to live * Has drivers for most languages * Automatic failover | All Bson data types such as:  Strings,  Numbers,  Booleans,  Date/Time,  Arrays,  Null,  More… | Easy to use with good documentation and excellent community support. | projects that require uptime, read reliability, and simplicity. |

We have decided to use MongoDB for this project.

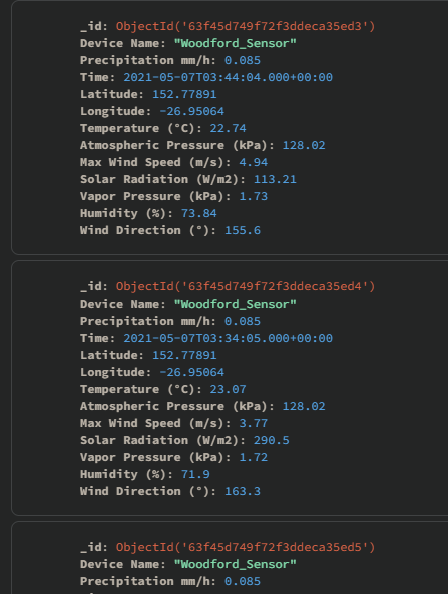
## MongoDB setup

**Data structure and data types:**

* Device Name (String)
  + A device name will most likely be alphanumerical.
* Precipitation mm/h (double)
  + Precipitation will always be a numerical value and may have decimals. Decimals won’t require high accuracy.
* Time (date)
  + Time may need to be searched upon so it should be stored as a date type and not a string
* Latitude (Double)
  + Latitude will always be numerical and requires decimals with high accuracy.
* Longitude (Double)
  + Longitude will always be numerical and requires decimals with high accuracy.
* Temperature (°C) (double)
  + Temperature will always be numerical and may have decimals. Decimals won’t require high accuracy.
* Atmospheric Pressure (kPa) (double)
  + Atmospheric Pressure will always be numerical and may have decimals. Decimals won’t require high accuracy.
* Max Wind Speed (m/s) (double)
  + Max Wind Speed will always be numerical and may have decimals. Decimals won’t require high accuracy.
* Solar Radiation (W/m2) (double)
  + Solar Radiation will always be numerical and may have decimals. Decimals won’t require high accuracy.
* Vapor Pressure (kPa) (double)
  + Vapor Pressure will always be numerical and may have decimals. Decimals won’t require high accuracy.
* Humidity (%) (double)
  + Humidity will always be numerical and may have decimals. Decimals won’t require high accuracy.
* Wind Direction (°) (double)
  + Wind Direction will always be numerical and may have decimals. Decimals won’t require high accuracy.

## Schema and dataset

Below is the schema for Weather readings. This also shows a preview of data being successfully imported into the database.



## Partitioning

* **Storage Requirements:**

current data:

* 120,360 entries
* First date recorded is 31/07/2020
* Last date recorded is 07//05/2021

To present a worst case scenario for data size I have imported all existing data,

Created an ascending and descending index for every key,

And created compound indexes containing every key in ascending and descending order

This has resulted in a total database size of 52342784 bytes or around 50 MB.

Seeing as all current data has been created within a year it would be reasonable to assume that this data size would increase by another 50 MB per year.

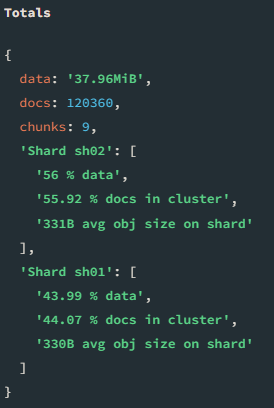
The WeatherData collection could benefit from sharding.

When choosing a key to partition by we should seek even distribution between all shards. This can be accomplished by doing the following:

* avoid high frequency keys. E.g we use the day of the week but almost all entries are added on a Friday.
* Use keys that have high cardinality. E.g keys with a high amount of potential values so that the amount of shards we can use wont be limited.
* Keys that aren’t monotonical. E.g a key that increases but never decreases.

With all this in mind I believe that wind direction would be the best choice.

* Database status and shard distribution after sharding.

The current data distribution is satisfactory as it is very close to 50%.

## Indexing and TTL in NoSQL

There are currently 120360 records that have been produced in 280 days. To calculate the through put per day we can divide 120360 by 280 which yields us our write through put per day of 430. This comes to around 17 writes per hour.

A read Through-put value can be calculated when the client provides current or expected request information.

### Indexes

#### WeatherData

The client has a lot of read requests involving a specific sensor. Due to this we will create a ascending index for the “**Device Name**” key.

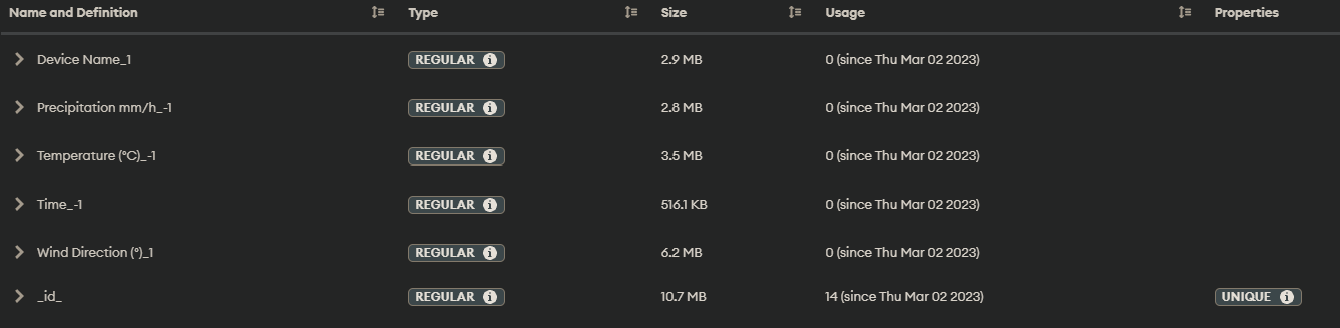
The client wants to be able to find data for a specific station (“Device name”) at a specific date and hour. With this in mind we will need to create a descending index for the “**Time**” key.

For the purpose of sharding we will create an increasing index for the “**Wind Direction (°)**” key.

To assist in finding the maximum temp for a specific date-time range for all sensors we will create a descending index for the “**Temperature (°C)”** key

To assist in finding the max precipitation recorded in the last x amount of time we will create a descending index for the “**Precipitation mm/h**” key

Our current indexes are listed below:



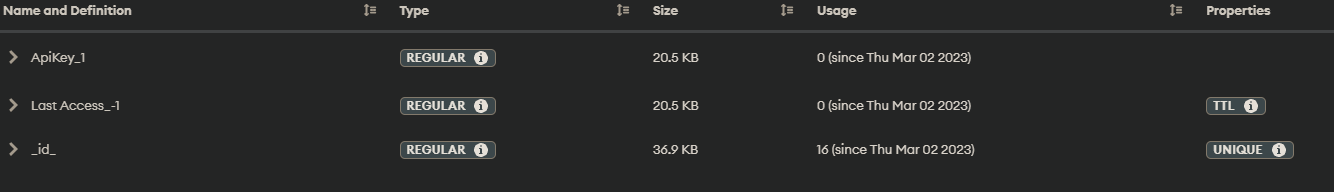
#### Users

The users collection will likely be small and most keys are infrequently queried.

With this in mind I have created a descending index on “**Last Access**”. Due to project requirement this index has a ttl of 30 days.

As the api will frequently be checking api the roles of a user by searching their api key, we will create a ascending index on “**ApiKey**”.

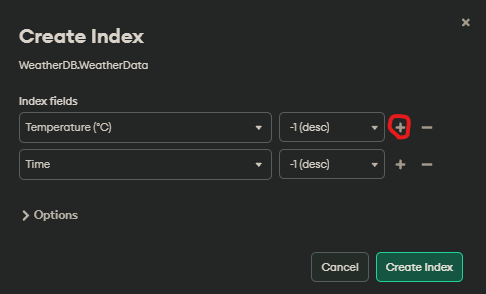
Our current indexes are listed below:



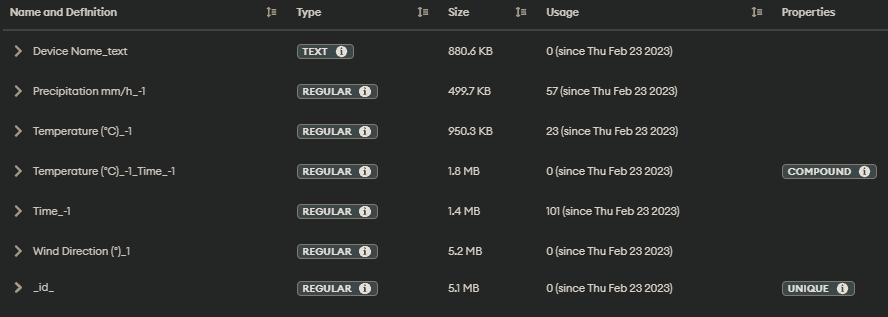
### ***Creating special indexes***

#### Compound index

create an index in MongoDBCompass as normal and when presented with this menu press the “**+**” button circled below in red



After that select the second index in the new drop down menu. Then create the index



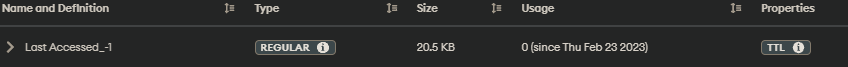
**For this project we have opted to not use a compound index as there was no measurable performance benefit while using one.**

#### TTL Index

create an index in MongoDBCompass as normal. When you reach the menu below expand the options and select “Create TTL”, then type in the time-to-live for the index.

**Note:** The selected key must be of TIME type or TTL wont work.





|  |  |
| --- | --- |
| * + MongoDB database SIGNOFF   + Signing off on this document signifies that the **MongoDB database setup** presented **complies** with the Client’s Business **requirements.** | |
| * + Project Manager or relevant stakeholder: shaun   + Signature: shaun   + Date: 02/03/2023 | * + Web Developer: joshua ward   + Signature: joshua   + Date: 02/03/2023 |
| * + DocumentationAPPROVED   + Please provide feedback on the changes needed. | |
| * + **APPROVAL**  Granted  Not Granted | |

# PART TWO – REST API

# Build and document REST API Project

## Review Project Requirements

The client has requested for our team to develop an API to access weather station readings. **The client wishes to have to following features**:

1. Database encryption at rest.
2. Encryption during transit across the network
3. User authentication and authorization.
   1. To access specific endpoints.
   2. To perform specific functions.
4. Admin users with authentication and authorization.
   1. To create users.
   2. To delete users.
   3. To modify users’ permissions.
5. API authentication.
   1. To perform crud operation on a limited amount of collections in the database.
6. Last login times to be updated when a user successfully queries a endpoint.
7. Data Persistence.
8. Optimised queries using indexes.
9. Data validation upon creation or modification. No document can have the following:
   1. Humidity greater than 100%
   2. Temperature higher than 60 degrees Celsius
   3. Temperature less than -50 degrees Celsius

**The API should be able to perform the following**:

1. Insert a new reading for a weather station.
2. Insert a new user. (ADMIN ONLY)
3. Insert multiple sensor readings for single station.
4. **Find** the maximum precipitation recorded in the last 5 Months for a specific sensor, returning the sensor name, reading date / time and the precipitation value.
5. **Find** the temperature, atmospheric pressure, radiation and precipitation recorded by a specific station at a given date and time (hour).
6. **Find** the maximum Temp(C) recorded for all stations for a given Date / Time range (start and finish date) returning the Sensor Name, reading date / time and the Temperature value.
7. Have at least 1 API endpoint that uses an index key to make the query more efficient.
8. Delete a user by Id. (ADMIN ONLY)
9. Delete multiple non-admin users that have not logged in for more than 30 days. (ADMIN ONLY)
10. Update a specified entries’ precipitation value to a specific value. (ADMIN ONLY)
11. Update access level for at least two users in the same query, based on a date range in which the users were created. (ADMIN ONLY)

**This project would work best with a REST API due to the following**:

* High amount of data with low cohesion between data. (one weather reading does not depend on another)
* Scalability. This project’s data storage will grow. A REST API will continue to operate regardless of data size. If this projects data throughput grows a REST API can scale both horizontally and vertically with minimal adaptation.
* Portability. Users may access data from any device regardless of platform, as all platforms are capable of sending https/http requests.
* Flexibility. Users can request as much or little data needed. Perfect for statistics.
* Cacheable. Data in this project is unlikely to change and a REST API is capable of caching frequently accessed data.

## Selecting REST API framework and IDE

**When choosing an API framework we have to consider a lot of different factors**:

* Reliability
  + API’s need to be running 24/7 as services depend on them to function
* Scalability/Extensibility
  + API’s need to be able to adapt to data needs efficiently. This may mean more API endpoints or more data processing.
* Familiarity
  + When a developer understands a framework they can make better code faster that can be more future proof, bug free, and maintainable.
* Performance
  + An API framework needs to meet or exceed the performance requirement for the project.
* Features
  + An API framework may have features that can grant less code, Improved security, Improved reliability, And improved speeds.
* Documentation
  + An API framework must have good documentation to make programming easier.
* More…

**When choosing a IDE we have to consider similar factors**:

* Familiarity
  + When a developer is able to better utilise a IDE it can lead to faster development times, reduced bugs, and better documentation
* Features
  + IDE’s commonly have features to assist with programming. Such as code prediction, bug predictions, code optimisation recommendations, code debugging/profiling, and more.
* Usability
  + IDE’s should be easy to navigate and program with.

For this project I have decided to use the ASP.NET Core API framework with Swagger as a debugging/testing/documentation frontend.

* This API framework is stateless which introduces a high level or reliability.
* I believe this framework exceeds this projects current and future data through-put requirements (17 writes / hour).
* This API framework is highly extensible and feature rich. This will make current and future development faster and easier.
* This API framework has good documentation and community support.
* I am familiar with this API framework.

For this project I have chosen to use Microsoft Visual Studio Professional 2022 as I am most familiar with it and it has a high level of usability.

## Build the REST API: Endpoints and methods

This project requires numerous endpoints and each endpoint must follow the restful api standards, must provide expected functionality, must perform validation on provided data, and must provided appropriate responses.

This project requires authorization and authentication. This means we will have two different controllers:

* Users
  + Endpoints for performing CRUD operations on users
* WeatherReadings
  + Endpoints for performing CRUD operations on Weather Readings
  + Endpoints for statistics

With all this in mind if have created the following endpoints



A picture containing application

Description automatically generated

## Test methods

### Tests Conducted

All testing was done with an admin api key.

The request made to get testing results will be listed in the screenshot under “Curl”

|  |  |  |  |
| --- | --- | --- | --- |
| METHOD | ENDPOINT | METHODS RESPONSE | MATCH OCCURRED  Yes/No |
| GET | Users | 200 OK | Yes |
| POST | Users | 201 Created | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| PUT | Users | 201 Created | Yes |
| GET | Users/{id} | 200 OK | Yes |
| DELETE | Users/{id} | 204 No Content | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| PUT | Users/{id} | 204 No Content | Yes |
| PATCH | Users/{id} | 400 Bad Request | No |

|  |  |  |  |
| --- | --- | --- | --- |
| PATCH | Users/Many | 200 OK | Yes |
| PUT | Users/Many | 200 OK | Yes |
| DELETE | Users/Many | 200 OK | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| POST | Users/Many | 201 Created | Yes |
| GET | WeatherReadings | 200 OK | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| POST | WeatherReadings | 201 Created | Yes |
| PUT | WeatherReadings | 201 Created | Yes |
| GET | WeatherReadings/{id} | 200 OK | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| DELETE | WeatherReadings/{id} | 204 No Content | Yes |
| PUT | WeatherReadings/{id} | 204 No Content | Yes |
| PATCH | WeatherReadings/{id} | 204 No Content | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| POST | WeatherReadings/Many | 201 Created | Yes |
| DELETE | WeatherReadings/Many | 200 OK | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| PUT | WeatherReadings/Many | 200 OK | Yes |
| PATCH | WeatherReadings/Many | 200 OK | Yes |
| GET | WeatherReadings/  Statistics/  MaxPrecipitationForLastFiveMonths/  {deviceName} | 200 OK | Yes |
| GET | WeatherReadings/  Statistics/  ReadingsAtSpecificHourForStation/  {deviceName} | 200 OK | Yes |
| GET | WeatherReadings/  Statistics/  MaxTempBetWeenTimes/ | 200 OK | Yes |

### Tests after bug fixes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| METHOD | ENDPOINT | METHODS RESPONSE | BUG | MATCH OCCURRED  Yes/No |
| PATCH | Users/{id} | 204 No Content | Endpoint was using wrong validator. | Yes |

## Model Validation

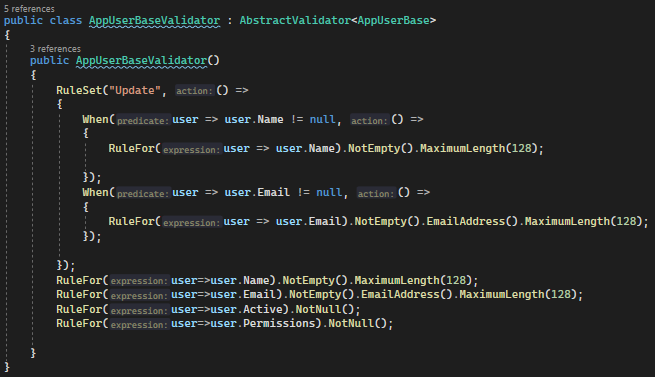
All endpoints consume and produce models. These models will need to be validated to ensure stability and completion of project requirements. In this project I have opted to use “Fluent validation” as it provides more validation options compared to asp.net core’s validation attributes.

Fluent validation works on the basis of validators for models and incorporates object oriented programming principles as well as asynchrony.

Fluent validation is static and any rules defined will be enforced thus no testing is required for simple validation rules such as must be email address or must be greater than or equal to.

Any complex validation rules will be tested in the next section

Example of a validator:



Fluent validation allows for automatic validation of a model upon creation or manual validation when requested in code. For this project I have chosen manual validation to ensure that all models from the client are validated but models from the database are not. This is done for future proofing purposes as validation rules may change over time.

Example of manual validation:

Text

Description automatically generated

See <https://docs.fluentvalidation.net/en/latest/> for documentation on fluent validation.

## Current Validation Rules

### AppUserBase

If updating a user:

1. If the username is being changed then the username must
   1. Not be empty
   2. Have a maximum length of 128
2. If the email is being updated then the email must
   1. Not be empty
   2. Be a valid email
   3. Have a maximum length of 128

Otherwise

1. Name must not be empty
2. Name must have a maximum length of 128
3. Active must be defined
4. Permissions must be defined

### AppuserCreateManyDTO

Use [AppUserBase](#_AppUserBase) validator

### AppuserFilter

1. If email is defined then email must be a valid email
2. If LastAccessedStart is defined then it must be greater than or equal to the unix epoch
3. If LastAccessedEnd is defined then it must
   1. Be less than or equal to the max unix time
   2. Greater than or equal to LastAccessedStart
4. If Limit is defined then it must be between 1 and 10000
5. If skip is defined then it must be greater than or equal to 0
6. If ApiKey is defined then:
   1. ApiKeys must not be defined
   2. Must be a valid ApiKey (complex)
7. If ApiKeys is defined and isn’t empty then it must contain only valid ApiKeys(complex)
8. If id is defined then:
   1. Ids must not be defined
   2. Id must be a valid ObjectID (complex)
9. If Ids is defined and isn’t empty then it must contain only valid ObjectIDs(complex)

### AppuserUpdateManyDTO

Use [AppUserBase](#_AppUserBase) validator with the ‘Update’ ruleset

### WeatherReadingBase

1. If updating a weather reading then don’t require the device name. otherwise require the device name
2. Device name must have a maximum length of 128
3. Latitude must be between -180 and 180
4. Longitude must be between -180 and 180
5. Temperature must be between -50 and 60
6. Humidity must be between 0 and 100
7. Wind direction must be between 0 and 360
8. Max wind speed must be greater than or equal to 0
9. Precipitation must be greater than or equal to 0
10. Solar radiation must be greater than or equal to 0
11. Vapour pressure must be greater than or equal to 0
12. Atmospheric pressure must be greater than or equal to 0

### WeatherReadingFilter

For this Validator no fields are required

1. Atmospheric pressure max must be greater than or equal to 0
2. Atmospheric pressure min must be greater than or equal to 0
3. If atmospheric pressure max is defined then it must be greater than or equal to atmospheric pressure min
4. Device name must have a maximum length of 128
5. Humidity max must be between 0 and 100
6. Humidity min must be between 0 and 100
7. If humidity max is defined then it must be greater than or equal to humidity min
8. Latitude max must be between -180 and 180
9. Latitude min must be between -180 and 180
10. If latitude max is defined then it must be greater than or equal to latitude min
11. Longitude max must be between -180 and 180
12. Longitude min must be between -180 and 180
13. If longitude max is defined then it must be greater than or equal to longitude min
14. Max wind speed max must be greater than or equal to 0
15. Max wind speed min must be greater than or equal to 0
16. If max wind speed max is defined then it must be greater than or equal to max wind speed min
17. Precipitation max must be greater than or equal to 0
18. Precipitation min must be greater than or equal to 0
19. If precipitation max is defined then it must be greater than or equal to precipitation min
20. Solar radiation max must be greater than or equal to 0
21. Solar radiation min must be greater than or equal to 0
22. If solar radiation max is defined then it must be greater than or equal to solar radiation min
23. Temperature max must be between -50 and 60
24. Temperature min must be between -50 and 60
25. If temperature max is defined then it must be greater than or equal to temperature min
26. Start time must be greater than or equal to the unix epoch
27. End time must be less than or equal to the max unix time
28. If end time is defined then it must be greater than or equal to start time
29. Vapour pressure max must be greater than or equal to 0
30. Vapour pressure min must be greater than or equal to 0
31. If vapour pressure max is defined then it must be greater than or equal to vapour pressure min
32. Wind direction max must be between 0 and 360
33. Wind direction min must be between 0 and 360
34. If wind direction max is defined then it must be greater than or equal to wind direction min
35. Limit must be between 1 and 10000
36. Skip must be greater than 0
37. Sort key must be a valid key (complex)
38. Sort order must be equal to “asc” or “desc”
39. If id is defined then:
    1. Ids must not be defined
    2. Id must be a valid ObjectID (complex)
40. If Ids is defined and isn’t empty then it must contain only valid ObjectIDs (complex)

### WeatherReadingUpdateDTO

Use [WeatherReadingBase](#_WeatherReadingBase) validator with the ‘update’ ruleset

### WeatherReadingUpdateManyDTO

1. Id must not be empty
2. Id must be a valid Id
3. Use [WeatherReadingUpdateDTO](#_WeatherReadingUpdateDTO) validator with ‘update’ ruleset

### WeatherReading

1. Use [WeatherReadingBase](#_WeatherReadingBase) validator with ‘update’ ruleset
2. Id must not be empty
3. Id must be a valid Id

## Complex Model Validation Testing

All complicated model validation rules will be tested using a Admin ApiKey and will be using dummy data

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Rule | Valid | Invalid |
| [AppUserFilter](#_AppuserFilter) | 6b Must be a valid ApiKey |  |  |
| [AppUserFilter](#_AppuserFilter) | 7 If ApiKeys is defined and isn’t empty then it must contain only valid ApiKeys |  |  |
| [AppUserFilter](#_AppuserFilter) | 8b Must be a valid ObjectID |  |  |
| [AppUserFilter](#_AppuserFilter) | 9 If ObjectIDs is defined and isn’t empty then it must contain only valid ObjectIDs |  |  |
| [WeatherReadingFilter](#_WeatherReadingFilter) | 37 Sort key must be a valid key |  |  |
| [WeatherReadingFilter](#_WeatherReadingFilter) | 39b Id must be a valid ObjectID |  |  |
| [WeatherReadingFilter](#_WeatherReadingFilter) | 40 If Ids is defined and isn’t empty then it must contain only valid ObjectIDs |  |  |

## Enable Cross-origin Resource Sharing (CORS)

As we are testing CORS and not the methods, any post, put, patch methods will not have data and will have validation errors. However these requests will still be able to demonstrate CORS

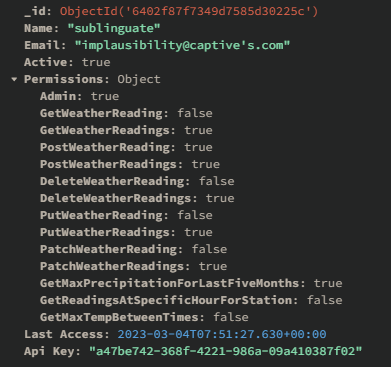
|  |  |  |
| --- | --- | --- |
| INSTANCE/METHOD  GET, POST, PUT & DELETE | CROSS-ORIGIN REQUEST | OUTPUT  RESULT |
| GET |  | Used CORS |
| POST |  | Used CORS |
| PUT |  | Used CORS |
| GET |  | Used CORS |
| PUT |  | Used CORS |
| DELETE |  | Used CORS |
| PUT |  | Used CORS |
| POST |  | Used CORS |
| DELETE |  | Used CORS |

## Pre-Flight Testing

|  |  |  |
| --- | --- | --- |
| INSTANCE/METHOD  GET, Post, PUT and Delete | PRE-FLIGHT REQUEST | OUTPUT  RESULT |
| GET |  |  |
| POST |  |  |
| PUT |  |  |
| GET |  |  |
| PUT |  |  |
| DELETE |  |  |
| PUT |  |  |
| POST |  |  |
| DELETE |  |  |

## Evaluate and secure REST API

To implement users and control what endpoints a user can access we will be using ApiKey based authentication with permissions that define what endpoints the user can access. On every user object is a permissions object. This permissions object will contain any defined permissions and if the user has that permission, represented via true/false. See schema below



* The Admin permission has access to every endpoint
* all non-user endpoints have accompanying permissions for access.
* A user’s permissions will only be valid when the user is active. Disabled users will not be able to access any endpoints regardless of permissions

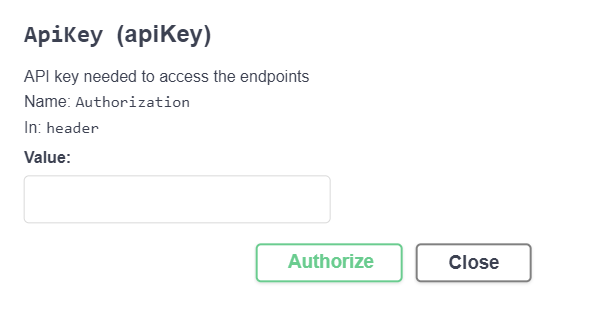
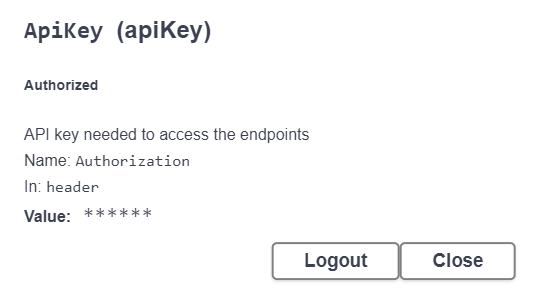
|  |  |  |  |
| --- | --- | --- | --- |
| ENDPOINT NAME | HTTP METHOD | AUTHORISATION DETAILS | AUTHENTICATION REQUIRED |
| ​/Users | GET | Admin | ApiKey Required |
| ​/Users | POST | Admin | ApiKey Required |
| ​/Users | PUT | Admin | ApiKey Required |
| ​/Users​/{id} | GET | Admin | ApiKey Required |
| ​/Users​/{id} | DELETE | Admin | ApiKey Required |
| ​/Users​/{id} | PUT | Admin | ApiKey Required |
| ​/Users​/{id} | PATCH | Admin | ApiKey Required |
| ​/Users​/Many | PATCH | Admin | ApiKey Required |
| ​/Users​/Many | PUT | Admin | ApiKey Required |
| ​/Users​/Many | DELETE | Admin | ApiKey Required |
| ​/Users​/Many | POST | Admin | ApiKey Required |
| ​/WeatherReadings | GET | GetWeatherReadings | ApiKey Required |
| ​/WeatherReadings | POST | PostWeatherReading or  PutWeatherReading | ApiKey Required |
| ​/WeatherReadings | PUT | PostWeatherReading or  PutWeatherReading | ApiKey Required |
| ​/WeatherReadings​/{id} | GET | GetWeatherReading | ApiKey Required |
| ​/WeatherReadings​/{id} | DELETE | DeleteWeatherReading | ApiKey Required |
| ​/WeatherReadings​/{id} | PUT | PutWeatherReading | ApiKey Required |
| ​/WeatherReadings​/{id} | PATCH | PatchWeatherReading | ApiKey Required |
| ​/WeatherReadings​/Many | POST | PostWeatherReadings | ApiKey Required |
| ​/WeatherReadings​/Many | DELETE | DeleteWeatherReadings | ApiKey Required |
| ​/WeatherReadings​/Many | PUT | PutWeatherReadings | ApiKey Required |
| ​/WeatherReadings​/Many | PATCH | PatchWeatherReadings | ApiKey Required |
| ​/WeatherReadings​/Statistics​/MaxPrecipitationForLastFiveMonths​/{deviceName} | GET | GetMaxPrecipitationForLastFiveMonths | ApiKey Required |
| ​/Statistics​/ReadingsAtSpecificHourForStation​/{deviceName} | GET | GetReadingsAtSpecificHourForStation | ApiKey Required |
| ​/Statistics​/MaxTempBetWeenTimes | GET | GetMaxTempBetweenTimes | ApiKey Required |

### How it works

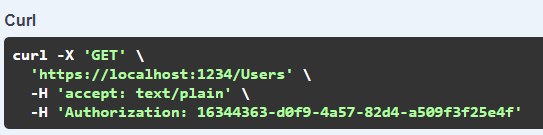
The Api key is a GUID generated when a user is created or replaced.

When performing authentication the request to the Api endpoint must contain a “Authorization” header that contains the users ApiKey. The api will check that the ApiKey is valid, if the user is active, and if the user has relevant permissions to access the requested endpoint.

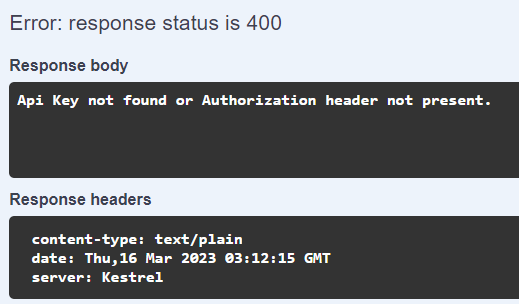
Authenticating look like this:

When a request is made the authorizaion header must be present



When the authorization header is not present the api will response with 400



The ApiKey must be valid and must have the required permissions for the endpoint

Graphical user interface, text

Description automatically generated

When the ApiKey is invalid the Api will respond with 400

Graphical user interface, application

Description automatically generated

When the Apikey provided doesn’t exist or doesn’t have permission for the endpoint the Api will respond with 401

Graphical user interface, text, application

Description automatically generated

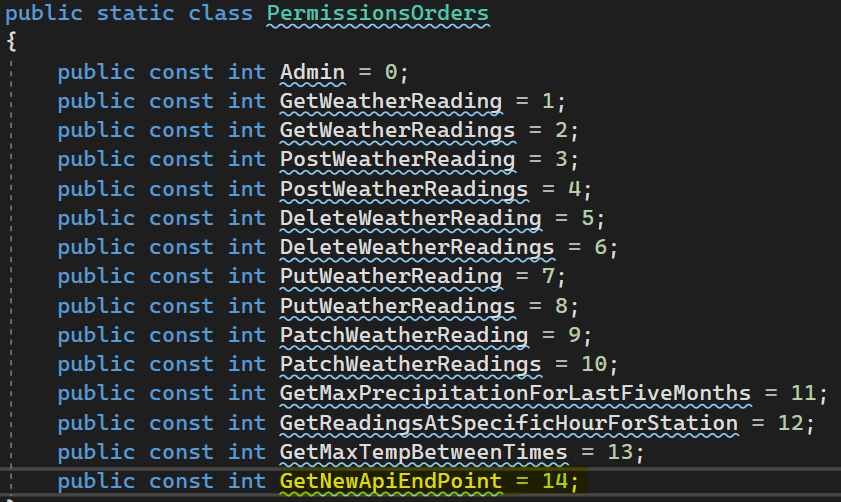
When an endpoint needs authorization use the “ApiKey” attribute and add the allowed permissions.  

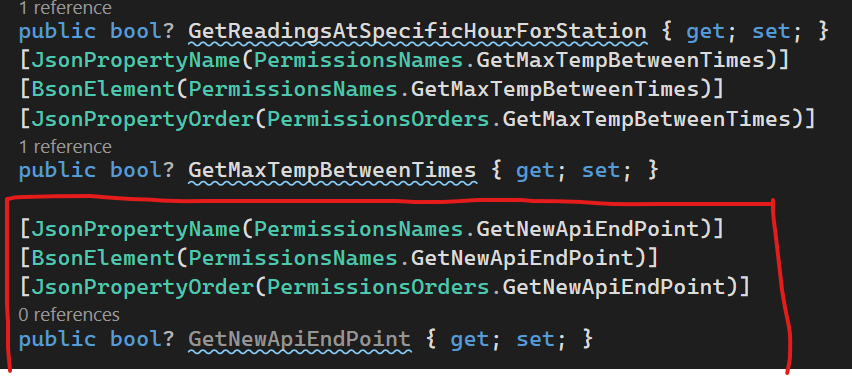

To create a new permission

1. Open the PermissionsNames class and add its name



1. Open the PermissionsOrders class and add its order. (this will define how its presented in swagger but is not necessary)



3. Open the Permissions class and add the permission  


## Documenting REST API endpoints

Documentation helps current and future developers to understand how a program works and how to use it.

Documentation in a REST Api shows how to use the API. This includes details such as endpoints, methods, response codes, responses, request body, request parameters, request headers, response headers, etc…

It is important to choose the right tools for documenting a API to make it easier to understand, test, and use an API

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TOOLS | MAIN FEATURES | COST  Free/Paid | EASE OF USE | BEST FOR  Type of Project | OVERALL RATING  (L)1-5 (H) |
| Swagger | 1. UI visualisation of endpoints, parameters, models, and responses.  2. Automatic client side validation in UI  3. client side testing | Free | 1. Easy integration with asp.net core.  2. Easy to interact with and to test API endpoints  3. Web based | 1. Static Api Endpoints.  2. Presentable documentation.  3. Private documentation | **3** |
| RapiPDF | Generating a pdf from openapi specification  Open source | Free | Simple configuration options and easy to generate. (provide openapi spec) | 1. Public documentation  2. Static Api endpoints | **3** |
| DocGen by Lucybot | 1. UI visualisation of endpoints, parameters, models, and responses.  2. client side testing | Free for non commercial projects | 1. Relatively easy to setup, (provide openapi spec).  2. Runs as standalone application on server.  3. Easy to interact with because its web based | 1. Private documentation  2. Static endpoints  3. Presentable documentation. | **4** |

**Swagger** has been chosen



The API documentation helps show how this API has met the project requirements.

I believe that the API documentation will help in showing users how to use the API.

Specifically, it helps explain types, models, requests, and responses.

this covers the following:

* Parameters
  + name
  + data type
  + valid values
  + description
* endpoints
  + name/location
  + description
  + methods
  + accept types
  + body
  + responses
* schemas
  + required values
  + names
  + valid values
  + description
  + data types

This API Documentation will prove useful when interacting with the API and improves the projects overall useability.

|  |  |
| --- | --- |
| * + REST API Endpoints Documentation SIGNOFF   + Signing off on this document signifies that the **documentation** presented **complies** with the Client’s Business **requirements.** | |
| * + Project Manager or relevant stakeholder   + Signature: shaun   + Date: 16/03/2023 | * + Web Developer   + Signature: joshua   + Date: 16/03/2023 |
| * + Documentation APPROVED   + Please provide feedback on the changes needed. | |
| * + **APPROVAL**  Granted  Not Granted | |

# PART THREE – REST API and MongoDB

# REST API and MongoDB Integration

To demonstrate that the client API is connected to the database and is working as expected below are a list of requests and responses.

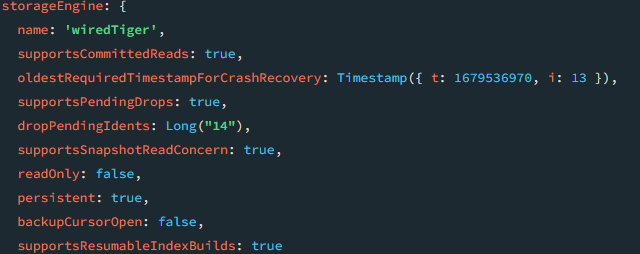
|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Method | Endpoint | Request/Response |
| Insert a single data object | POST | WeatherReadings | 201 Created |
| Perform multiple inserts in a single operation | POST | WeatherReadings/Many | 201 Created |

|  |  |  |  |
| --- | --- | --- | --- |
| Query a single object | GET | Users/{id} | 200 OK |
| Use batch object to retrieve multiple objects in one operation | GET | WeatherReadings | 200 OK |
| Include an index in a query | GET | WeatherReadings/{id}  (Id is an index) | 200 OK |
| Create and run a query that returns multiple attributes | GET | WeatherReadings/  Statistics/  MaxPrecipitation  ForLastFiveMonths/  {deviceName} | 200 OK |
| Delete a single object | DELETE | WeatherReadings/{id} | 204 No Content |
| Delete multiple objects | DELETE | WeatherReadings/Many | 200 OK |
| Update a single object | PATCH | WeatherReadings/{id} | 204 No Content |

|  |  |  |  |
| --- | --- | --- | --- |
| Update multiple objects | PATCH | WeatherReadings/Many | 200 OK |

## Further queries or object interactions

Although the business requirement don’t state any requirements for data persistence, this projects nature will require it. MongoDB has data persistence enabled by default and this can be seen in the wired tiger config



**Triggers**

Email

To: <manager>

From: Joshua Ward

Content:

Hey <manager>, I had some ideas for some triggers that may be useful in the Weather Readings REST Api project.

1. Logging – a trigger that will log all changes made to weather readings. This may be useful in rolling back changes to a weather reading if incorrect data was entered.
2. Validation – ensure that all weather readings are validated upon modification or creation. If an entry doesn’t fix within the logical limits of the data entry It will be deleted.

Feel free to get back to me with your opinion and any feedback or changes needed for these triggers.

Best regards –

Joshua Ward

**Feedback**:

Both are great ideas and will be implemented

## Implementation of triggers

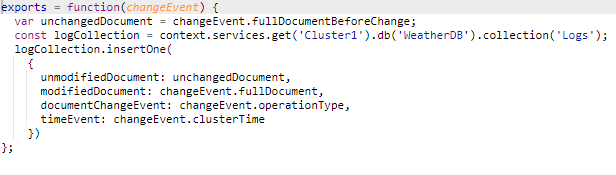
### Logging trigger

The logging trigger tracks all changes made to all weather readings and saves a history of the changes to a collection named “Logs”.

Trigger operation types:

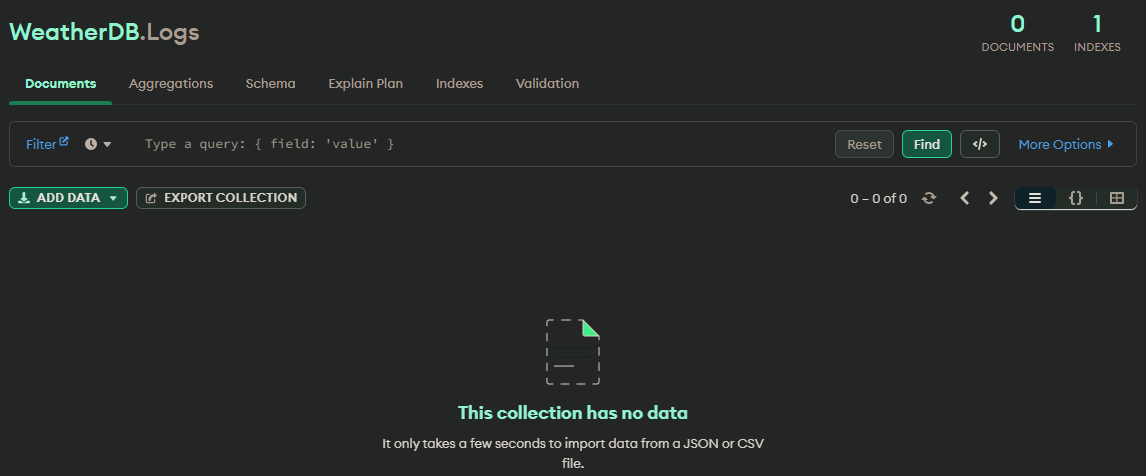


Source code:

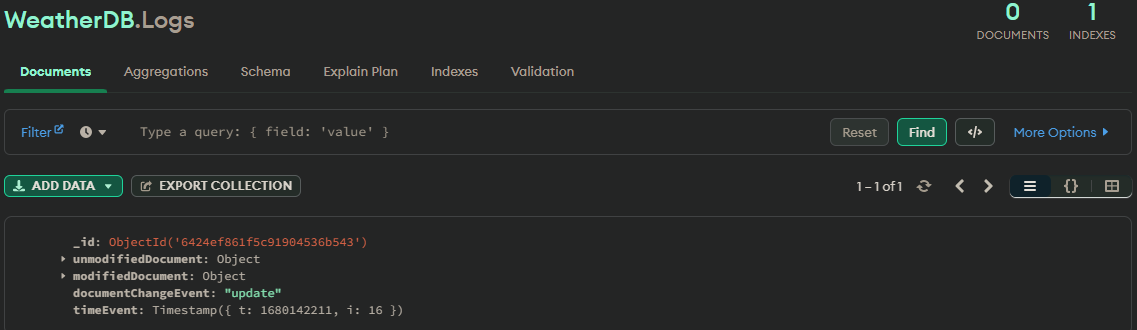


Demo:

“Logs” collection before any changes:



“Logs” collection after updating a weather Reading:



### Validation trigger

The Validation trigger ensures that all weather readings meet validation requirements when updated, inserted, or replaced. At the moment our current validation requirements for this trigger are:

Humidity can not be greater than 100%

Temperature can not be greater than 60

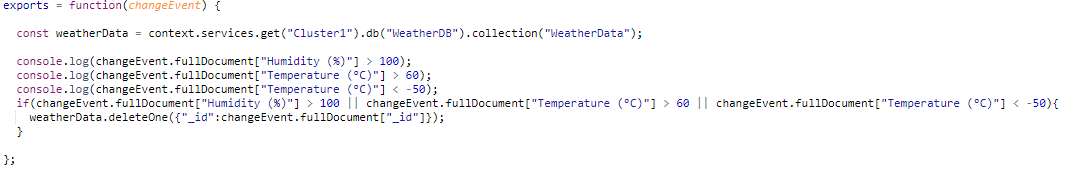
Temperature can not be less than -50.

Any weather readings that don’t match these requirements will be deleted.

Trigger operation types:

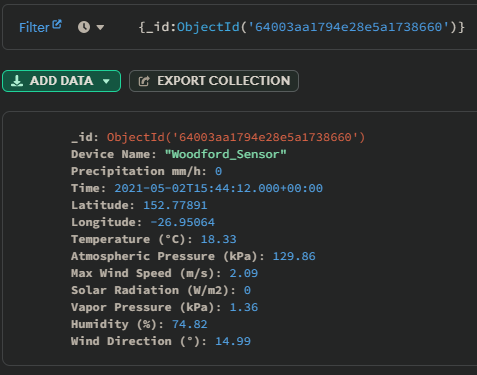


Source code:

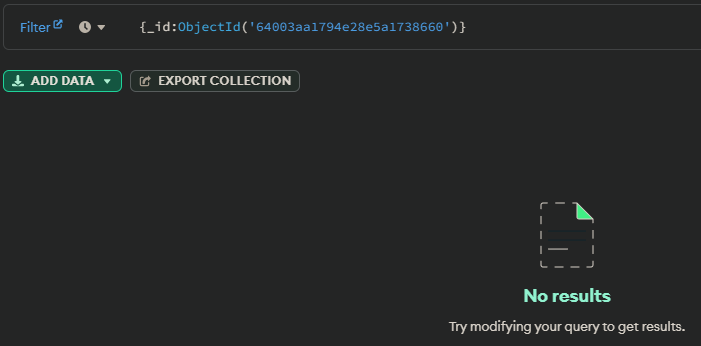


Demo:

Weather reading with id 64003aa1794e28e5a1738660 before setting humidity to 102%:

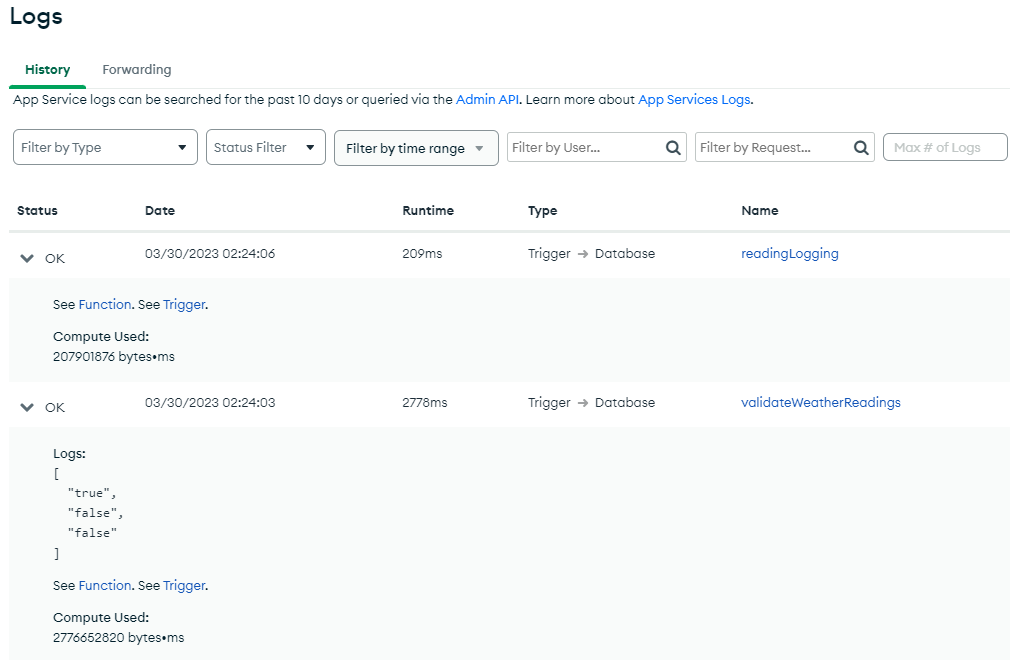


Weather reading with id 64003aa1794e28e5a1738660 after setting humidity to 102%:



(Not Found/Deleted)

Trigger Log after setting humidity to 102 on weather reading with id 64003aa1794e28e5a1738660:



## Data security

The project requires:

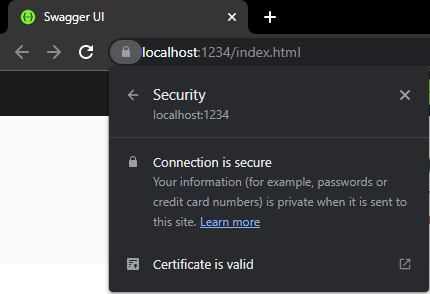
* Data to be encrypted at rest and transit
* Database authentication
* Database authorisation

**1.** Data is encrypted at rest on MongoDB atlas by default[[Source]](https://www.mongodb.com/docs/atlas/security-kms-encryption/#:~:text=Atlas%20encrypts%20all%20cluster%20storage%20and%20snapshot%20volumes%20at%20rest%20by%20default).

This mean that all data in the collections is encrypted when not in use.

Data is encrypted in transit via SSL/TLS between the Api Server and the MongoDb Atlas database by default[[Source]](https://www.mongodb.com/basics/mongodb-encryption#:~:text=MongoDB%20Atlas%20uses%20encryption%20in,pass%20access%20and%20authentication%20controls.).

Data is encrypted in transit via SSL/TLS between the Api Server and the client. This can be seen in the clients browser via the https padlock seen below:



**2.** Database authorisation is completed via Users and permissions.

This project has a user called ApiServer:

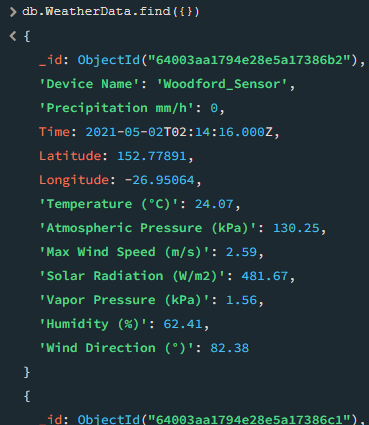


ApiServer only has read/write access to the weather readings and users.

This prevents the user from creating/deleting/viewing any other databases or collections.

Test:

While logged in as ApiServer through MongoSH I could only access WeatherData and Users.



I could not access the logs collection.



I could not create a new collection.



**3.** Database authentication is required is completed via Users and SCRAM authentication.

The MongoDB Atlas server wont let anyone connect without first authenticating as a valid user.

This database has a user called ApiServer:

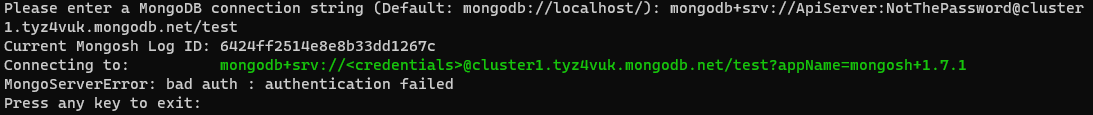


Test:

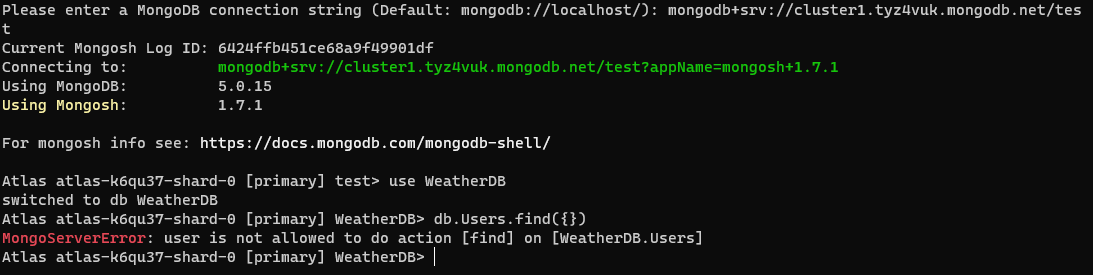
Authenticating with the correct connection string:



Authenticating with an incorrect password:



Connecting without credentials:



(Login succeeds but does not have any permissions)

## Finalising the project

|  |  |
| --- | --- |
| * + REST API and NoSQL Projects SIGNOFF   + Signing off on this document signifies that the project and **documentation** presented **complies** with the Client’s Business **requirements.** | |
| * + Project Manager or relevant Stakeholder   + Signature: shaun   + Date: 30/03/2023 | * + Web Developer   + Signature: Joshua   + Date: 30/03/2023 |
| * + Documentation APPROVED   + Please provide feedback on the changes needed. | |
| * + **APPROVAL**  Granted  Not Granted | |