Cloud Application Development Continual Assessment 2024

How can Cloud Services be used to Construct a Serverless Microcontroller Temperature Monitoring Application

Deirdre Lee T00126583

**Abstract**

This project investigates the use of cloud services for developing a serverless microcontroller temperature monitoring system. The purpose of the project is to explore the feasibility and effectiveness of using cloud infrastructure, in this case AWS, to enhance the scalability, reliability and efficiency in this proof of concept (POC) temperature monitoring system.

The research methodology involves a review of research papers relating to this topic, online documentation regarding both AWS and micro:bit, as well as user manuals pertaining to microcontroller integration in a number of areas. This study examines the integration of cloud services such as AWS Lambda, DynamoDB, API Gateway, and Amplify.

The main conclusion reached highlights the ease of use of AWS services for developing a serverless microcontroller application, highlighting the potential of cloud-based solutions in Internet of Things (IoT) applications.

*Keywords: Microcontroller, Monitoring, AWS, Serverless, API, Alarm, Notification, DynamoDB, Amplify, POC, IoT*

# **Introduction**

This report takes a look into the development of a serverless microcontroller temperature monitoring application using AWS. It aims to answer the question: How can Cloud Services be used to Construct a Serverless Microcontroller Temperature Monitoring System?

The background to this project stems from the escalating interest in cloud-based IoT solutions, driven by their scalability and efficiency. This investigation is particularly relevant given the continuing demand for IoT solutions, as apparent by current market trends (Luth, 2024), and the potential benefits offered by cloud-based approaches to this field.

The subsequent methodology follows an in-depth literature review, as well as practical experimentation with various AWS services, and the use of a micro:bit microcontroller. Through this approach, the research aims to highlight the feasibility and effectiveness of a cloud-based microcontroller system.

It is important to acknowledge that certain aspects, such as device security and application authentication, will not be explored for the POC system, due to time and research constraints.

Overall, this report aims to provide insights into the integration of cloud services for a small microcontroller application, with a focus on temperature monitoring, in a purely exploratory manner. This exploration is merely a beginning, laying the groundwork for further inquiry in this continually trending field of cloud services incorporated with IoT.

# **Literature Review**

The literature review stage of the project employed various methodologies, including keyword searches in academic databases such as IEEE Xplore and ScienceDirect, along with an examination of relevant journal articles in the field of IoT, cloud computing, and sensor technology. Notable trends observed during the search included an increasing emphasis on cloud-based solutions for IoT applications, particularly in the context of data collection and monitoring in smart city projects, as comprehensively discussed in “Cloud-Based Data Storage and Visualization Tool for Smart City IoT: Flood Warning as an Example Application”. (Leal Sobral et al., 2023)

During the research, it was noted that the devices used for monitoring were Wi-Fi, or LoRaWAN, enabled to some extent, as . Additionally, multiple devices were employed for the monitoring process, ensuring a more accurate reading. The system architecture described in "Cloud-based Sensor Network for Environmental Monitoring" depicts a robust system with three layers, with the middle 'bridge layer' containing the cloud component. Despite not having access to the same range of devices, this project was modelled on the proposed architecture.(Corbellini et al., 2017) The paper by Corbellini et al., in particular, served as the fundamental guide for developing the project system. The methodology presented was reliable, clear, and well-developed, offering alternative scenarios and options, which proved to be incredibly helpful.

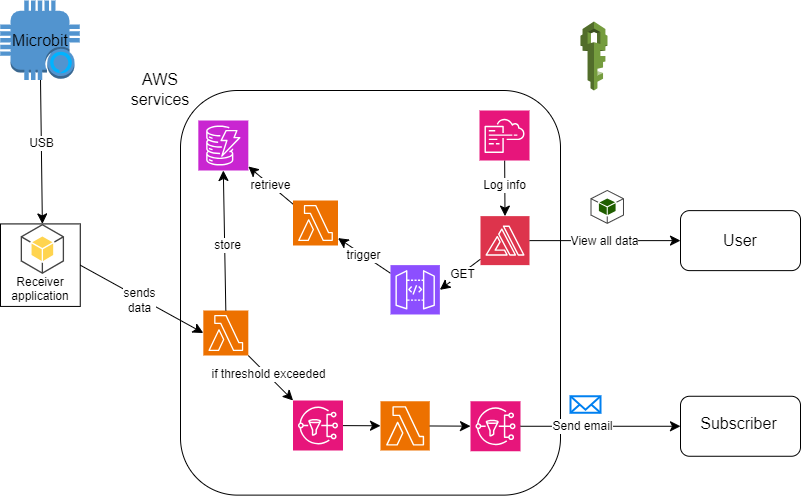
Throughout the research, it was also observed that the devices used for monitoring required a standalone power source. In instances where they were not connected to a device via USB, they could use a battery pack for power, which is relevant in the case of using a micro:bit. For the POC application, it was not deemed necessary to source a battery pack due to there being only one monitoring device, as well as the fact that without an intermediary communication device, the connection via USB was not only vital for power but also for communication with a receiver program. However, this decision also raises questions about the scalability and mobility of the monitoring system in real-world applications, where multiple devices and varied deployment scenarios may be encountered. Future research could explore alternative power solutions and communication protocols to address these challenges and enhance the versatility and reliability of the monitoring system.

An intriguing yet previously unknown concept that emerged through research is LoRa technology. This technology is a wireless communication technology designed for long-range communication with low power consumption, making it suitable for applications such as IoT devices, environmental monitoring, and asset tracking, as outlined in Rapid IoT Prototyping: A Visual Programming Tool and Hardware Solutions for LoRa-Based Devices (López and Lamo, 2023). Micro:bit LoRa devices are available for purchase, so this will be an area of interest for further exploration and development of this project. The identification of LoRa technology as a potential solution for wireless communication in the project presents an exciting avenue for exploration, although this would necessitate the consideration of factors such as cost, power consumption, range, and compatibility with the existing infrastructure, if any. It would also be essential to assess how feasible the integration of Micro:bit LoRa devices would be into the project.

In conclusion, the literature review provided valuable insights that informed the research in several ways. It highlighted the growing trend towards cloud-based solutions for IoT applications, emphasising their scalability, efficiency, and cost-effectiveness.



# **Methodology**



## **Develop Sensor Data Collection**

For this project, Mu Editor was installed to work on the program locally, as suggested in the book ‘*Beginning BBC micro:bit: A Practical Introduction to micro:bit Development’* (Seneviratne, 2018). A basic script was created to check if the temperature received from the micro:bit exceeded predefined thresholds, printing a warning across the microcontroller screen as well as the current temperature. The final simple progam for the micro:bit can be seen in [Appendix Fig 3.1.1.](#_Appendix)

### **Transmit Readings to Laptop Device**

Initially after research Bluetooth was considered as the communication option for this project. However, upon further investigation, it was determined to not be viable for the current scope of the project. This decision was influenced by the need for an intermediary device to facilitate communication over Bluetooth Low Energy (BLE) via a ‘bridge layer’ as outlined in the architure proposed in "Cloud-based Sensor Network for Environmental Monitoring". (Corbellini et al., 2017) The micro:bit v1.3B, being the primary sensor device, cannot transmit data via BLE. While it possesses built-in radio capabilities, it operates on a custom protocol based on, but incompatible with, BLE. To enable BLE communication, an intermediary device is essential. One feasible option involves using a Raspberry Pi, which supports BLE. Acting as the bridge, the Raspberry Pi would receive data from the micro:bit via its radio module, process it, and transmit it over BLE to the receiving device. (BBC micro:bit MicroPython, 2024) Given these considerations, the decision was made to rely on an alternative approach that involves using a USB cable to transmit data directly to a laptop. As part of this setup, the micro:bit code was updated to log readings at specified intervals and trigger events when threshold exceedances occurred (microbit, 2022), the temperature logs being stored in memory. To ensure continuous logging, I developed a Python script that imported the ‘pyserial’ library to establish communication with the micro:bit via the serial port, testing communication initially using PuTTY. (Liechti, 2020) This script reads the data from the micro:bit and stores it in a Python dictionary. This printed data was then received through the program. The final recevier program code can be seen in [Appendix Fig. 3.1.1.1.](#_Appendix)



## **Set up AWS Identity & Access Management (IAM)**

As services were created, any necessary permissions were assigned by creating policies and assigning them to the required roles, following the principle of least privilege. Throughout the description of steps taken, there are included indications of which services this was implemented for. Images of IAM policies, roles and users are shown in [Appendix Fig. 3.2.1 - Fig 3.2.3](#_Appendix_1).

## **Create AWS Lambda Function**

After completing the previous steps of creating the initial microcontroller program and setting up communication with the receiving laptop, it was necessary to begin the cloud application process. The initial plan for the cloud implementation included utilising AWS IoT Core to manage communication with the IoT device. However, since the device is not WiFi-enabled and needs to communicate through the Python receiver program, as a result of the exploration in ['3.1.1 Transmit Readings to Laptop Device’](#_Transmit_Readings_to) this service is not suitable for the POC version. The receiver program created will need to communicate with the cloud, and to do this, it was decided to create a Lambda function to receive the data through a HTTP endpoint. This also necessitated the creation of an API Gateway service to facilitate the connection and eventual storage to the database. This will be discussed further in section [‘5. Set up API Gateway’](#_Set_up_API).

The Lambda function, ‘HttpRequestHandlerLambda’, was created. This function, triggered by a call to API Gateway, manages incoming sensor data, storing it in a DynamoDB table, and triggering alerts via SNS when temperature thresholds are exceeded. (Amazon AWS, 2024) The code for this function can be seen in [Appendix Fig 3.3.1](#_Appendix_1).

The ‘ThresholdExceededAlarm’ Lambda function processes incoming SNS messages, (Amazon AWS, 2024)extracting temperature-related information such as the current temperature, location, and predefined threshold, which are all passed as variables via the previous lambda. (Amazon AWS, 2024) It constructs a warning message if all required information is present, alerting about temperature breaches. The function then publishes this message to a specified SNS topic. Error handling ensures that any issues encountered during execution are logged and properly raised for troubleshooting. The code for this function can be seen in [Appendix Fig 3.3.2](#_Appendix_1)

The ‘HttpGetAllData’ Lambda function uses the DynamodDB scan operation to retrieve all items from the MicroSensor table. It then iterates through each item to check for instances of type Decimal and converts them to type float. Finally, the table items are returned along with a 200 status code. The code for this function can be seen in [Appendix Fig 3.3.3](#_Appendix_1).

## **Set up API Gateway**

For the ‘HttpRequestHandlerLambda’ lambda function created in ['5 Create AWS Lambda Function'](#_Create_AWS_Lambda) it was necessary to create a REST API gateway with a POST endpoint and add it as a trigger to the Lambda function. Unfortunately, as the project progressed, an issue arose with the POST endpoint. Despite a significant amount of time given to debugging at the eleventh hour, which included testing in Postman, using log messages, as well as using the in built AWS API testing function, a resolution was not found. As a result, it was necessary to invoke the ‘HttpRequestHandlerLambda’ lambda function, which was to be triggered by a call to the API Gateway, directly from the receiver code. This solution was not ideal, as the addition of more microcontrollers would be better served through the use of a REST API as planned, but for the POC it was sufficient for demonstration purposes.

On the other hand, to facilitate the retrieval of the table items using the ‘HttpGetAllData’ lambda function, adding a GET endpoint found at ‘/alltempdata’ was necessary and led to no known isses. Calling this endpoint from the application triggers the aforementioned ‘HttpGetAllData’ serverless function flawlessly, returning all the recorded temperature data to the user. [Appendix Fig 3.4.1](#_Appendix_1) shows the layout of the API Gateway resources.

## **Set up Amazon Simple Notification Service (SNS)**

Before creating the ‘ThresholdExceededAlarm’ Lambda function, an SNS topic and subscription were created. A topic called ‘TempThresholdAlarm’, which by default can only be published and received by the topic owner, was also created. The default specifying the topic owner as the only subscriber can be updated but it was deemed sufficient for this POC. Subsequently, a subscription was created, using an email protocol, and defining the author's college email to be the only subscriber. (Amazon AWS, 2024) A subscription confirmation email was then sent to the email address, to which the user was then able to successfully subscribe. After creating the ‘ThresholdExceededAlarm’ function a new IAM policy was added to allow the Lambda function permission to publish to SNS. (AWS Knowledge Center, 2024) The ‘HttpRequestHandlerLambda’ function was then updated to invoke the ‘ThresholdExceededAlarm’ function, passing the current temperature, location, and threshold variables. These variables are then included in the updated message sent to the subscriber. This was achieved by using Amazon SNS to trigger the second Lambda function. (Amazon AWS, 2024) A new topic, called ‘NotifyAlarmLambda’, was created and a subscription with a Lambda protocol set up, which triggers the ‘ThresholdExceededAlarm’ function from the ‘HttpRequestHandlerLambda’ endpoint. The receiving function, ‘ThresholdExceededAlarm’, was then updated to parse the incoming message and perform the SNS actions. Additionally, both Lambda functions were ensured to have the necessary IAM roles to perform the required actions. The ‘HttpRequestHandlerLambda’ function, when invoked, publishes to the SNS topic ‘NotifyAlarmLambda’, providing the necessary variables ‘curr\_temp’, ‘location’, and ‘threshold’. In short, the ‘ThresholdExceededAlarm’, subscribed to the ‘NotifyAlarmLambda’ topic, parses the incoming message details and incorporates them into a message published to the ‘TempThresholdAlarm’ topic. This message is distributed to all subscribers. Overall, the SNS topics and subscriptions were easy to manage and updateable to include multiple subscribers as the application grows. Topic and subscriptions can be seen in [Appendix Fig 3.5.1](#_Appendix_1).

## **Configure Amazon DynamoDB**

### **3.6.1 Database Plan for Storing Sensor Readings**

DynamoDB is used for storing the temperature readings and other associated data. DynamoDB was chosen due to its ability to handle large quantities of data and scale efficiently. It is also suitable for real-time processing, which is crucial for this application. Users will need to be able to access historical data, and there will be continuous write operations as the temperature data is continuously stored as long as the application is running.

Currently, the data being collected includes the temperature reading in degrees Celsius to the nearest whole number. It also records when the threshold is exceeded, as well as holding the value of the upper and lower temperature thresholds. Additionally, the schema needs to include the following information: Sensor ID, a unique identifier for each sensor device; Location, details about the location of each sensor device (such as room name); Time of Reading, a timestamp indicating when each temperature reading was recorded; and Exceeded Threshold, a flag indicating whether the temperature reading exceeded the predefined upper or lower thresholds. There is only one sensor device being used for this POC application, but as part of the non-functional requirement of scalability, the schema needs to reflect the fact that multiple sensors may be present. The decision was made to use a composite primary key for the table, designating ‘sensorID’ for the partition key. This decision was made due to the need for scalability, as the system is expected to interact with multiple sensors beyond the POC phase. By distributing data across multiple partitions based on ‘sensorID’, DynamoDB can efficiently handle increased data or workload. (Amazon AWS, 2024) ‘TimeDate’ was allocated as the sort key. This choice aligns with DynamoDB's design principles and helps with efficient querying, especially for accessing historical data. (Amazon AWS, 2024) By organising data based on the event time, the table supports optimised data retrieval and storage. The microcontroller code was updated to pass only the required variables to the receiver program. Despite spending a lot of time researching and testing, time data was unable to be passed from the microcontroller using MicroPython. Instead, a variable is initialised on the receiver side, which is then stored with the passed variables. The table can be seen in [Appendix 3.6.1.1.](#_Appendix_1)

### **Set up Storage Solution with DynamoDB**

The next step in the project was to set up the DynamoDB table to store the data received by the Lambda function via the API Gateway. A table called ‘MicroSensor’ was created, setting ‘SensorID’ as the partition key and ‘DateTime’ as the sort key as previously discussed. An IAM policy was then created, called ‘DynamoDBReadWrite’, to allow permissions to the Lambda functions to read and write to the table. After creating the policy, it was attached to the ‘HttpRequestHandlerLambda’ role.

## **Set up AWS Amplify**

Continuing with the project, a simple Vue.js application was deployed using AWS Amplify. Amplify's streamlined deployment process was used, with the project first being initialised with Amplify CLI, with necessary settings configured. The build settings were easily changed in Amplify, the build file shown in [Appendix Fig 3.8.1](#_Appendix_1). The application is set up to deploy on a push to the GitHub repository, which is listed in AWS as the source provider. Deployment is also achievable by using a webhook. Finally, the deployment of the Vue application to the cloud was achieved with the command, amplify publish, making the application almost instantly accessible to users. The link to the app from the Amplify dashboard, as well as the actual live website, are shown in [Appendix Fig 3.8.2 – 3.8.3](#_Appendix_1).

# 

# **Results/Data/Findings**

The project achieved its goal of establishing a basic temperature monitoring system by effectively leveraging the micro:bit microcontroller with the receiver program, alongside various AWS services, and the Vue.js application.

The development of various Lambda functions enabled seamless integration with DynamoDB and SNS for efficient data storage, retrieval, and alerting. The integration of API Gateway facilitated smooth communication between the microcontroller receiver application and cloud services. With DynamoDB configured to store temperature readings and associated data, the development of the project illustrated the ease of use and effectiveness of AWS services for data management.

Finally, the deployment of the Vue.js application using AWS Amplify ensured rapid deployment and immediate access for users, further highlighting the user-friendly nature of the AWS services. CloudFormation proved an invaluable tool in debugging the Amplify application.

Overall, the project’s success emphasises the effectiveness of online documentation, which offered thorough guidance throughout the development process.

# **Discussion**

The project faced initial challenges due to the limited capabilities of the micro:bit, which prompted the need for an alternative approach as the initial plan to connect directly using AWS IoT Core was unfeasible. This led to the exploration of alternative methods to establish communication with the microcontroller, ultimately resulting in the adoption of a workaround solution.

Initial limitations were encountered due to unfamiliarity with IoT devices and the complexities of AWS services. A significant portion of the allocated project time was spent researching the various components, testing, as well completing AWS tutorials in an attempt to become more familiar with the services. IAM policies were the most straightforward part of the cloud aspect of the project to work with, as there were many routes to implement it and clear instruction. Despite these challenges, the project progressed, eventually utilising direct USB communication as an alternative to Bluetooth.

While AWS services were generally user-friendly and accompanied by well-documented resources and examples, challenges arose when encountering issues or uncertainties with certain services. In some cases, documentation was found to be outdated or lacking in comprehensiveness, necessitating further research and exploration to arrive at a resolution. Nonetheless, the integration of various AWS services into the project proceeded relatively smoothly.

The extensive array of available AWS services posed a challenge, leading to occasional feelings of becoming overwhelmed, with the need to take a step back and prioritise, refraining from incorporating additional services simply because they were available. Additionally, the absence of a budget constraint limited the exploration of certain paid AWS services, potentially restricting the project's scope and capabilities.

The project navigated through minor challenges such as device limitations, technical knowledge gaps, and budget constraints. However, it importantly highlighted the ease of use and effectiveness of AWS services. Furthermore, the research process revealed the wealth of online resources available for IoT products. This suggests that with the appropriate tools and a strategic approach, developing an efficient, scalable, and secure monitoring system is highly achievable, with the incorporation of AWS services and capabilities significantly benefitting such endeavors.

# **Conclusion**

Moving forward, if the project were to be redone or built upon, priority would be given to enhancing security measures to ensure device and data protection and confidentiality. Additionally, with more time and research, it is believed that additional AWS services could be integrated to further enhance system functionality, such as CloudWatch for monitoring and Cognito for authentication. Existing services could also be implemented more efficiently, with a deeper understanding and refinement of their configurations, such as expanding on Amplify for the application backend. Considering that there was no budget allocated, service costs had to be carefully considered in the decision-making process.

Overall, while the project achieved its objectives, there is plenty of room for improvement and optimisation. With continued exploration and refinement, the system could evolve to meet more specific requirements, as well as offering enhanced performance and security for both the application and users.

Additionally, ongoing monitoring and maintenance would be crucial to ensure the system's reliability and effectiveness over time.

Link to GitHub Repository containing all code files: <https://github.com/EmoSense/therm_alarm>

Link to AWS Hosted Application: <https://main.d39t6hls28dk0h.amplifyapp.com/>



# **Appendix**

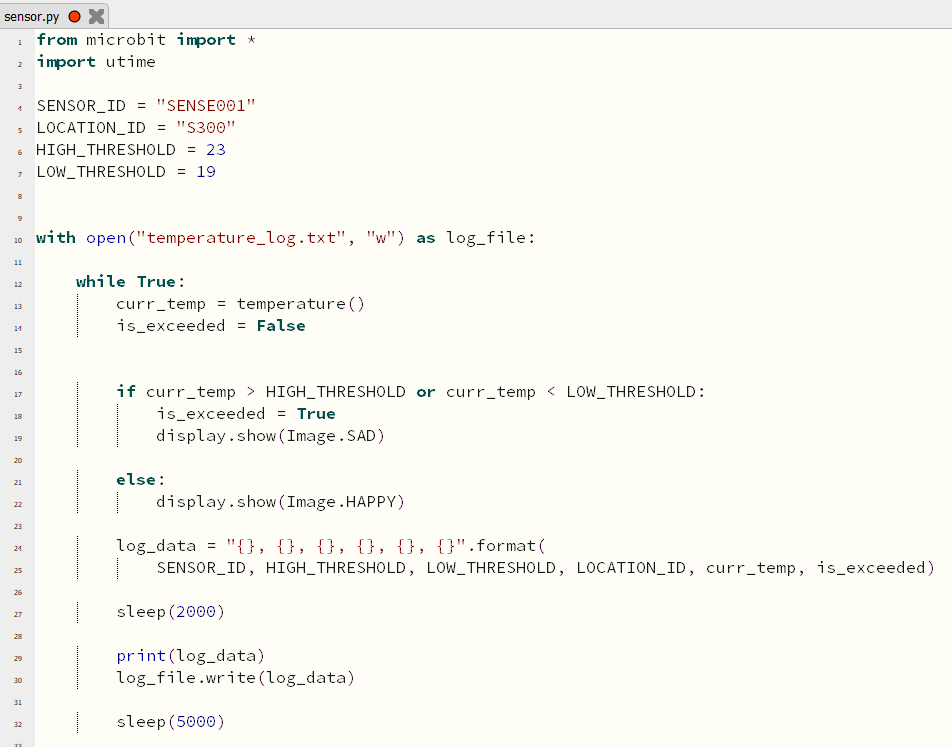


Fig 3.1.1 Micro:bit program

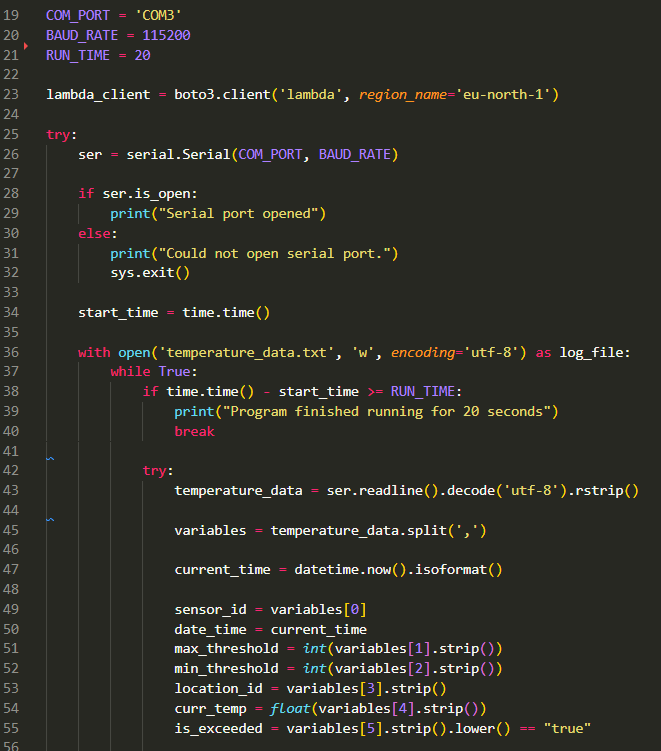


Fig 3.1.1 Receiver Code

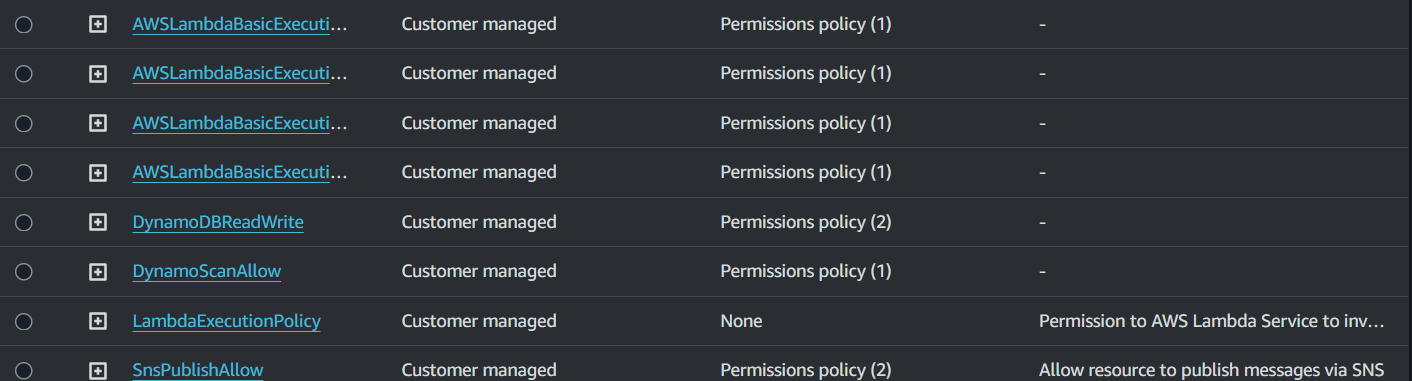


Fig 3.2.1 IAM Policies

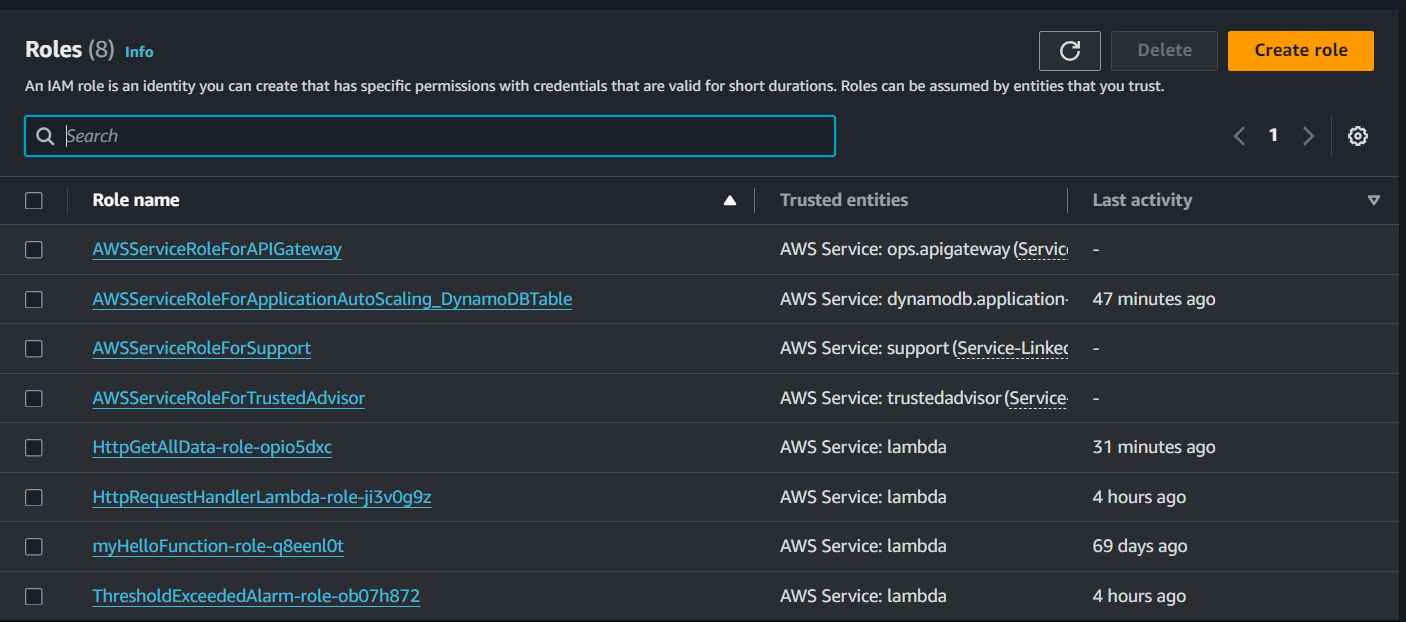


Fig 3.2.2 IAM Roles

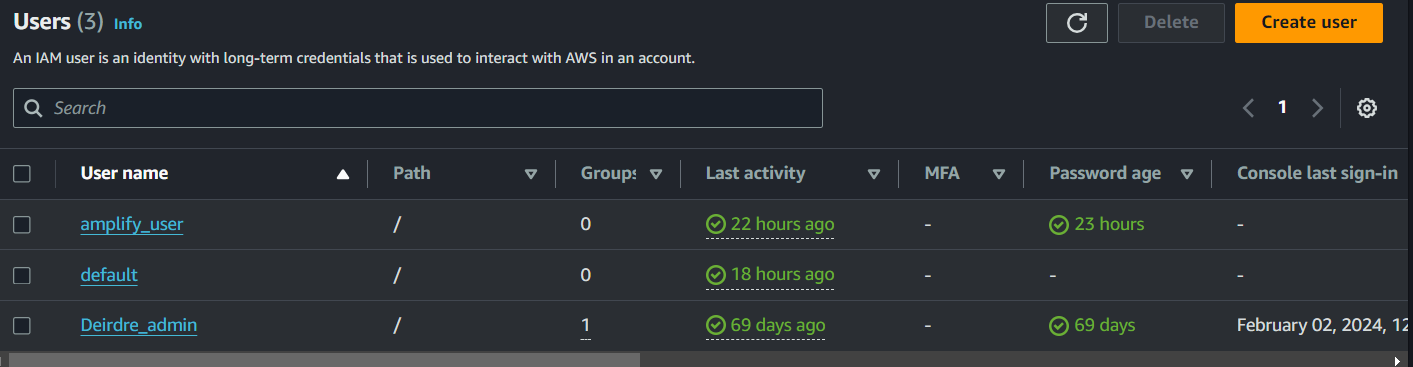


Fig 3.2.3 IAM users



Fig 3.3.1 HttpRequestHandlerLambda



Fig 3.3.2 ThresholdExceededAlarm

**

Fig 3.3.3 HttpGetAllData

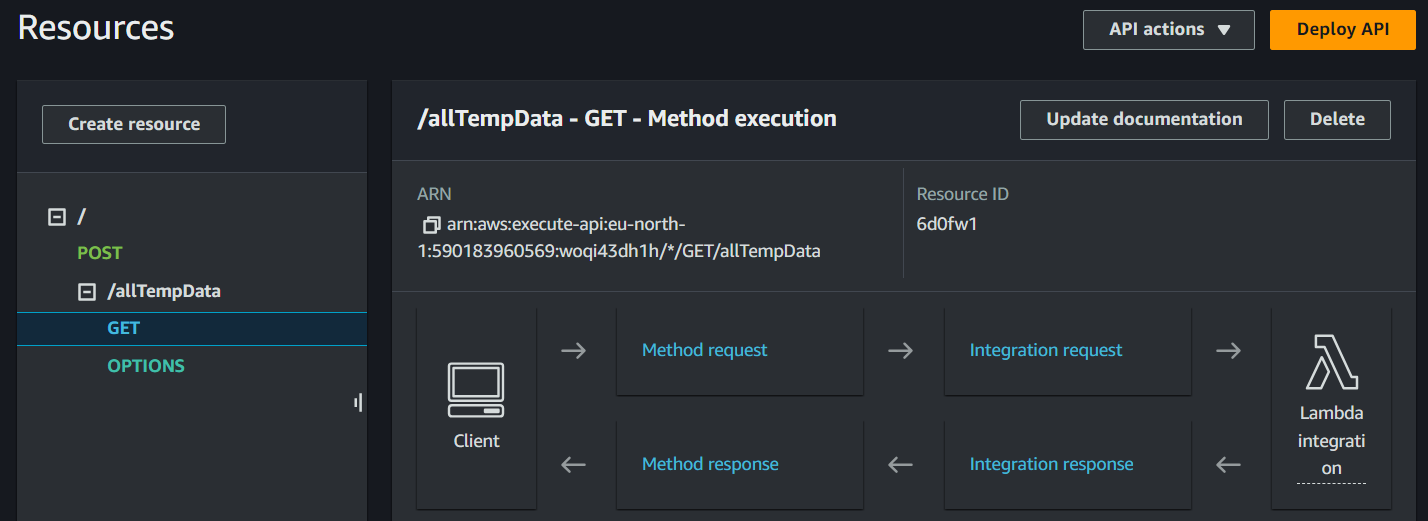


Fig 3.4.1 API Gateway

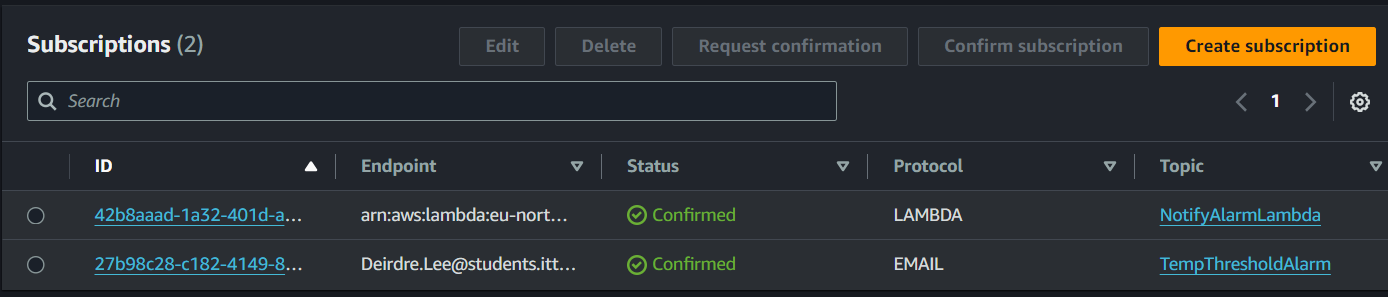


Fig 3.5.1 Subscriptions and Topics

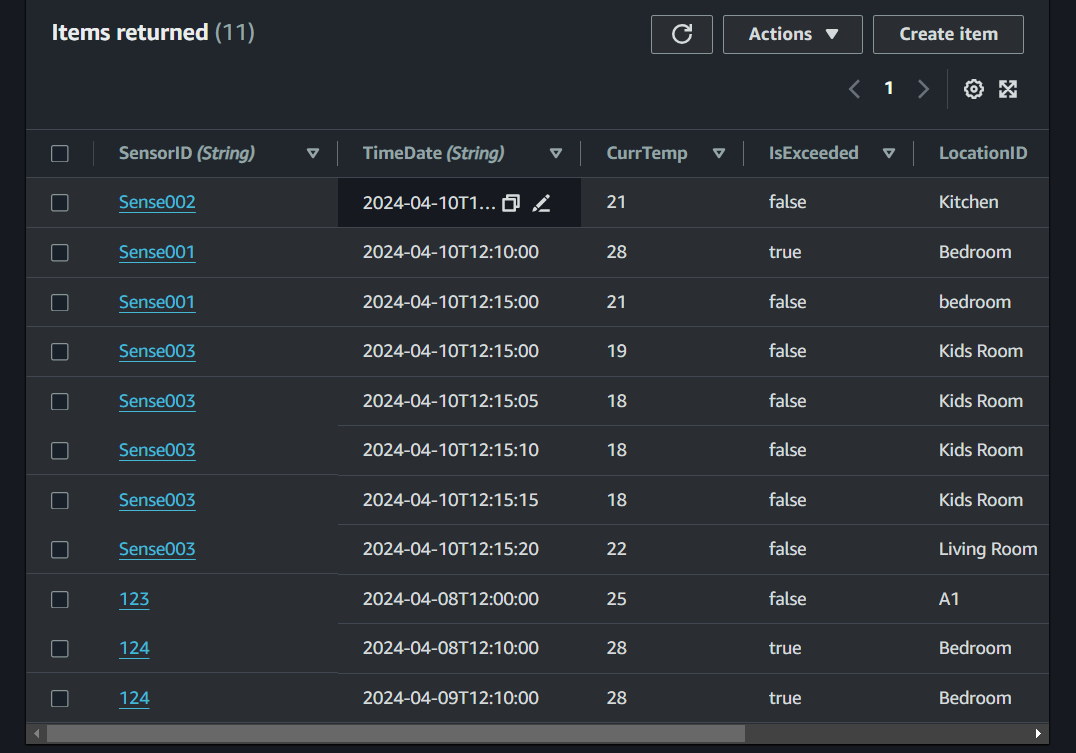


Fig 3.6.1.1 DynamoDB table

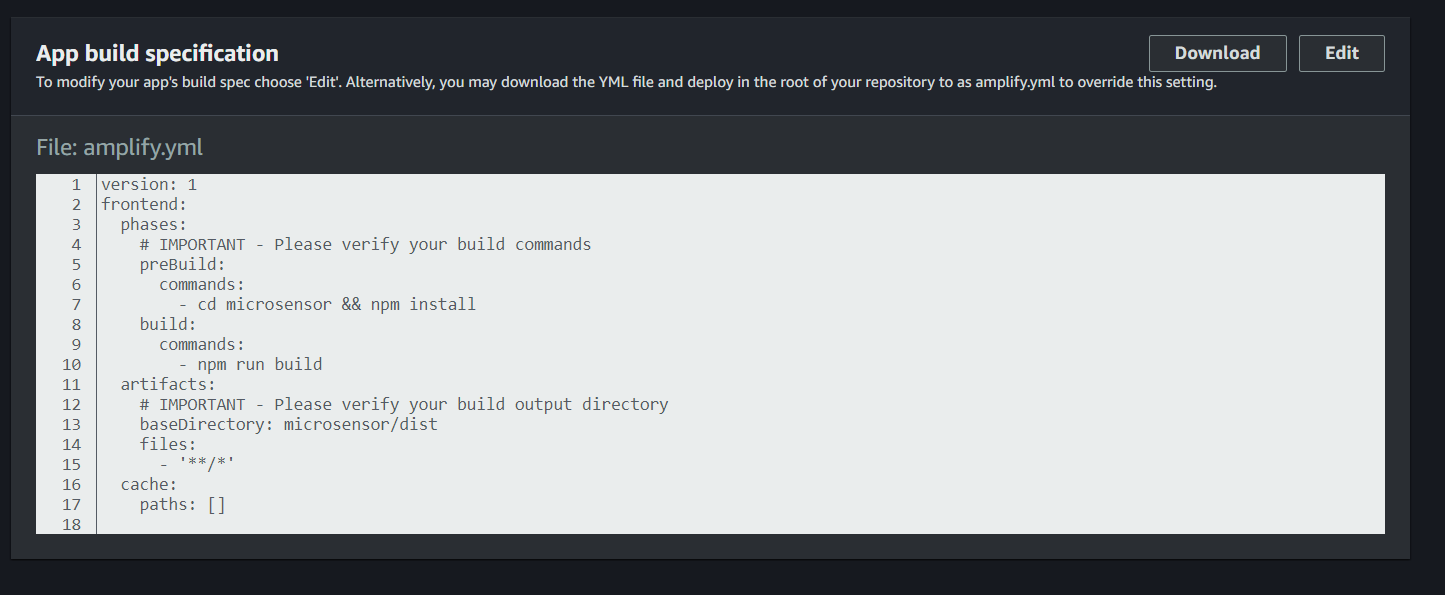


Fig 3.8.1 Vue app build file

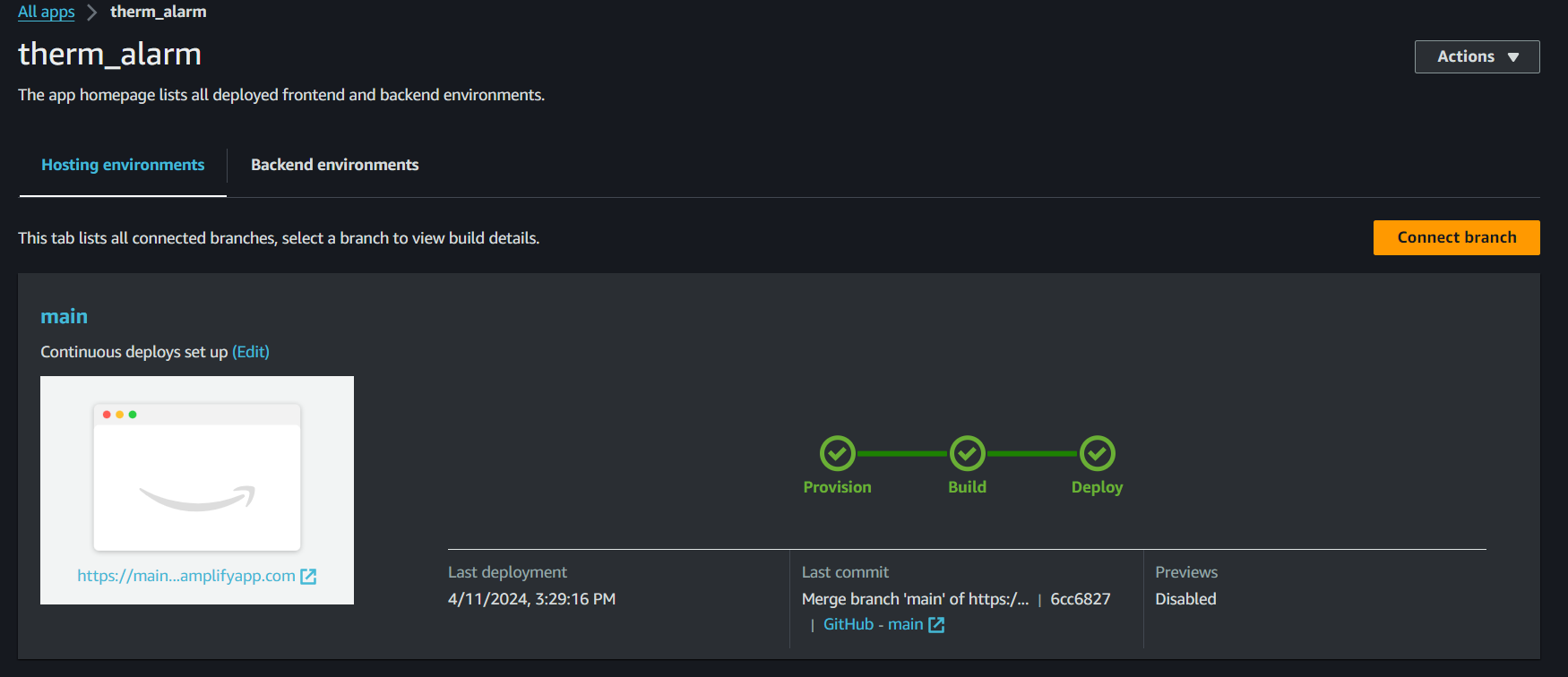


Fig 3.8.2 Amplfy app dashboard

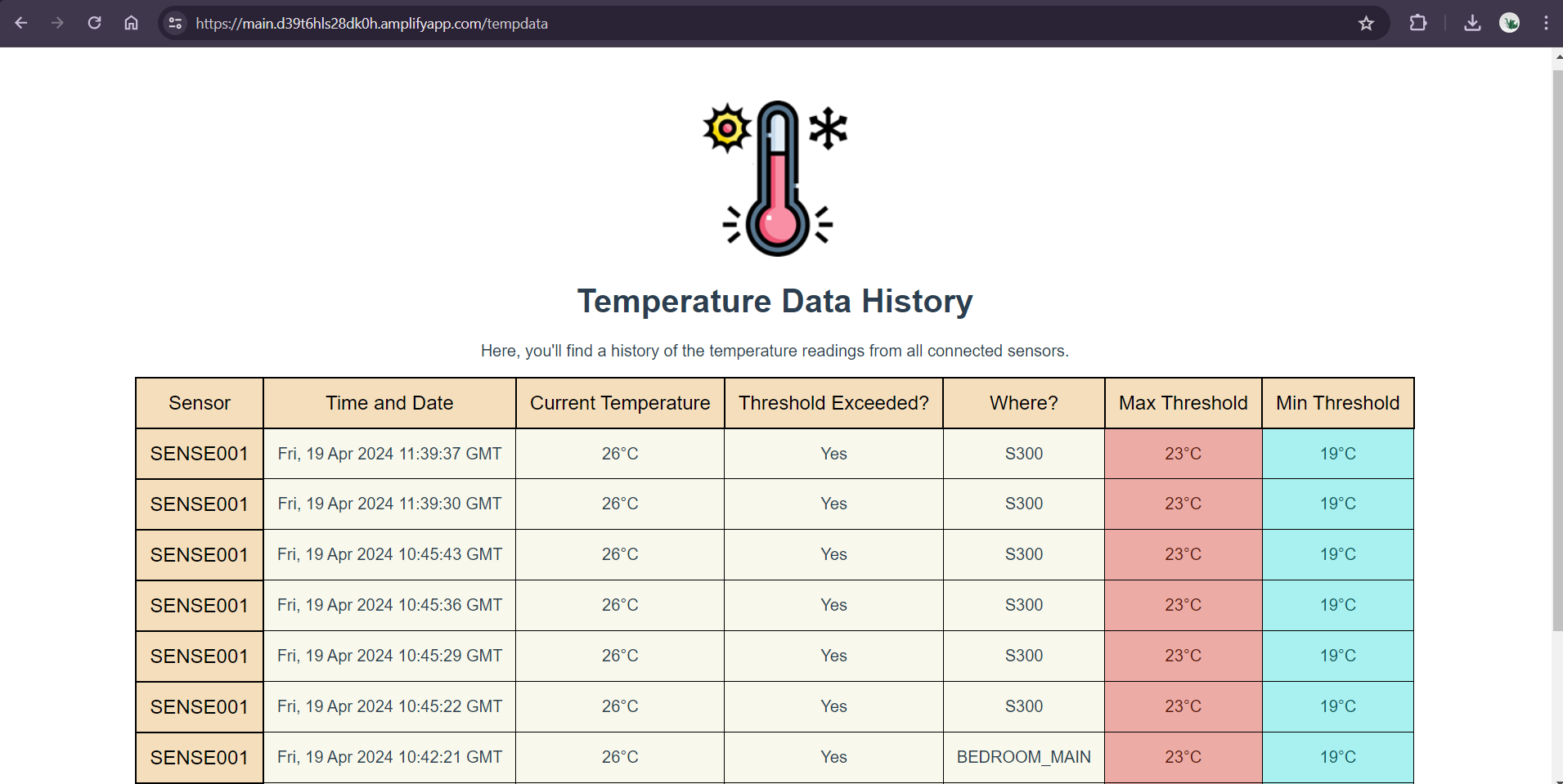


Fig 3.8.3 App data

# **References**

Amazon AWS, 2024. *Best practices for designing and using partition keys effectively.* [Online]   
Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html  
[Accessed 5 April 2024].

Amazon AWS, 2024. *Best practices for using sort keys to organize data.* [Online]   
Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-sort-keys.html  
[Accessed 5 April 2024].

Amazon AWS, 2024. *Fanout to Lambda functions.* [Online]   
Available at: https://docs.aws.amazon.com/sns/latest/dg/sns-lambda-as-subscriber.html  
[Accessed 9 April 2024].

Amazon AWS, 2024. *Getting started with Amazon SNS.* [Online]   
Available at: https://docs.aws.amazon.com/sns/latest/dg/sns-getting-started.html  
[Accessed 9 April 2024].

Amazon AWS, 2024. *https://docs.aws.amazon.com.* [Online]   
Available at: https://docs.aws.amazon.com/lambda/latest/dg/with-sns-create-package.html#with-sns-example-deployment-pkg-python  
[Accessed 7 April 2024].

Amazon AWS, 2024. *Processes a Message from Amazon SNS.* [Online]   
Available at: https://docs.aws.amazon.com/lambda/latest/dg/with-sns-create-package.html#with-sns-example-deployment-pkg-python  
[Accessed 6 April 2024].

Amazon AWS, 2024. *Tutorial: Build a CRUD API with Lambda and DynamoDB.* [Online]   
Available at: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamo-db.html  
[Accessed 8 April 2024].

AWS Knowledge Center, 2024. *How can I publish a message to an Amazon SNS topic using a Lambda function?.* [Online]   
Available at: https://repost.aws/knowledge-center/sns-topic-lambda  
[Accessed 9 April 2024].

BBC micro:bit MicroPython, 2024. *BBC micro:bit MicroPython Radio.* [Online]   
Available at: https://microbit-micropython.readthedocs.io/en/latest/radio.html  
[Accessed 1 April 2024].

Liechti, C., 2020. *pyserial.* [Online]   
Available at: https://pyserial.readthedocs.io/en/latest/shortintro.html#opening-serial-ports  
[Accessed 3 April 2024].

Luth, K. L., 2024. *State of IoT Spring 2024: 10 emerging IoT trends driving market growth.* [Online]   
Available at: https://iot-analytics.com/state-of-iot-10-emerging-iot-trends-driving-market-growth/#:~:text=According%20to%20the%20latest%20148,market%20rather%20than%20a%20disrupter.  
[Accessed 10 April 2024].

microbit help & support, 2021. *Outputing serial data from the micro:bit to a computer.* [Online]   
Available at: https://support.microbit.org/support/solutions/articles/19000022103-outputing-serial-data-from-the-micro-bit-to-a-computer#:~:text=You%20can%20send%20and%20receive,and%20Makecode%20for%20micro%3Abit.&text=In%20Python%2C%20the%20  
[Accessed 4 April 2024].

microbit, 2022. *Serial.* [Online]   
Available at: https://makecode.microbit.org/device/serial  
[Accessed 4 April 2024].

Nuclino, 2024. *A Guide to Functional Requirements.* [Online]   
Available at: https://www.nuclino.com/articles/functional-requirements  
[Accessed 1 April 2024].

Seneviratne, P., 2018. *Beginning BBC micro:bit: A Practical Introduction to micro:bit Development.* 1st ed. Sri Lanka: Apress.

Corbellini, S., Di Francia, E., Grassini, S., Iannucci, L., Lombardo, L. and Parvis, M., 2017. Cloud based sensor network for environmental monitoring. [online] https://doi.org/10.1016/j.measurement.2017.09.049.

Leal Sobral, V.A., Nelson, J., Asmare, L., Mahmood, A., Mitchell, G., Tenkorang, K., Todd, C., Campbell, B. and Goodall, J.L., 2023. A Cloud-Based Data Storage and Visualization Tool for Smart City IoT: Flood Warning as an Example Application. *Smart Cities*, 6(3), pp.1416–1434. https://doi.org/10.3390/SMARTCITIES6030068.

López, J.J. and Lamo, P., 2023. Rapid IoT Prototyping: A Visual Programming Tool and Hardware Solutions for LoRa-Based Devices. *Sensors*, 23(17). https://doi.org/10.3390/S23177511.