

CREATING AN ESP32 TEMPERATURE SENSOR WITH AWS IOT CORE AND FLUTTER

Court Onni

Bachelor Thesis

Degree programme in Business Information Technology
Bachelor of Business Administration

2024

Business Information Technology
Bachelor of Business Administration

Author(s)	Onni Court	Year	2024
Supervisor(s)	Juha Orre		
Title	Creating an ESP32 temperature sensor with AWS IoT Core and Flutter		
Number of pages	47 + 38		

The aim of this thesis was to create an ESP32 IoT temperature sensor prototype that publishes its data to AWS Cloud, a Flutter app that reads the data and a document that guides the reader through setting up the sensor and cloud environment. In doing so, this thesis hoped to answer the questions on how to create an IoT temperature sensor and what it can be used for.

This thesis references specifications, AWS documentations, research papers and articles. The result of this thesis was an ESP32 IoT temperature sensor that uses AWS IoT Core and its Rules engine to publish temperature data into AWS DynamoDB, a Flutter app that reads the temperature data from AWS DynamoDB, and a document that guides the user through the steps to setup the sensor and cloud environment. A case study on the possible usage of the temperature sensor in data centre temperature monitoring was also produced.

The results of this thesis can be used by anyone as a starting point in their journey into learning IoT sensor development and AWS IoT Core.

Keywords Internet of things, sensors, AWS, ESP32, Flutter, application development

Tietojenkäsittelyn koulutus
Tradenomi (AMK)

Tekijä(t)	Onni Court	Vuosi	2024
Ohjaaja(t)	Juha Orre		
Työn nimi	ESP32-lämpötila-anturin kehittäminen	käyttäen	
	AWS IoT Corea ja Flutteria		
Sivumäärä	47 + 38		

Tämän opinnäytetyön tarkoitus oli kehittää ESP32-IoT-lämpötila-anturin prototyyppi, joka lähettää lämpötila datan AWS-pilveen, Flutter-sovellus, joka lukee tätä tietoa, sekä dokumentti, joka opastaa lukijaa anturin ja pilviympäristön asennuksessa. Opinnäytetyön tavoite oli vastata kysymyksiin, miten IoT-lämpötila-anturi kehitetään ja mihin sitä voidaan käyttää.

Opinnäytetyössä viitataan standardeihin, AWS-dokumentteihin, tutkimuspapereihin ja artikkeleihin. Opinnäytetyön tuloksena syntyi ESP32-IoT-lämpötila-anturi, joka käyttää AWS IoT Corea ja sen Rules-moottoria lämpötilatietojen tallentamisessa AWS DynamoDB -tietokantaan. Lisäksi kehitettiin Flutter-sovellus, joka lukee lämpötilatietoja tietokannasta ja dokumentti, joka opastaa lukijaa anturin kehittämisessä ja pilviympäristön asennuksessa. Opinnäytetyö tuotti myös tapaus-tutkimuksen lämpötila-anturin mahdollisesta käytöstä datakeskuksen lämpötilan valvonnassa.

Kuka tahansa voi käyttää näitä tuloksia lähtökohtana IoT-anturin kehittämisessä ja AWS IoT Coren oppimisessa.

Avainsanat

esineiden internet, anturit, AWS, ESP32, sovellusten kehittäminen

CONTENTS

1	INTRODUCTION	7
2	OVERVIEW OF TECHNOLOGIES AND CONCEPTS USED.....	9
2.1	Internet of Things.....	9
2.2	The MQTT protocol.....	9
2.2.1	MQTT topics and wildcards	10
2.2.2	Quality of Service levels	10
2.2.3	Client Identifiers and message payloads	11
2.2.4	Understanding MQTT operation	12
2.2.5	Authentication and Authorization methods	12
2.3	Transport Layer Security	13
2.4	AWS IoT Core	13
2.4.1	Key features explained	14
2.4.2	Fleet provisioning options	15
2.5	AWS Amplify.....	16
2.6	Flutter	16
2.7	ESP32 microcontrollers and the ESP-IDF development framework ...	17
2.8	Bluetooth Low Energy.....	17
2.8.1	The Generic Attribute Profile and Universally Unique Identifier (UUID)	17
2.8.2	Differences between Bluetooth Classic and Bluetooth Low Energy	18
2.8.3	Security features in Bluetooth Low Energy	18
3	DESIGN OF THE SOFTWARE.....	21
3.1	ESP32 temperature sensor program functionality	21
3.1.1	State 1: Device provisioning	21
3.1.2	State 2: Register sensor and State 3: Publishing temperature data	23
3.2	AWS IoT Core certificates	24
3.3	Flutter app functionality.....	25
4	SOFTWARE IMPLEMENTATION.....	27
4.1	Publishing temperature data to AWS IoT Core.....	27
4.2	Device registration using fleet provisioning by claim	29
4.3	DynamoDB table structure.....	33

4.4	AWS API Gateway functionality.....	33
5	CASE STUDY: DATA CENTRE TEMPERATURE MONITORING	37
5.1	Problem statement.....	37
5.2	Solution.....	38
5.3	Results.....	38
5.4	Conclusion.....	39
6	DISCUSSION	40
	REFERENCES	41
	APPENDICES.....	45

SYMBOLS AND ABBREVIATIONS USED

AIoT	Artificial Intelligence of Things
ARC	Argonaut RISC Core
ATT	Attribute Protocol
AWS	Amazon Web Services
BLE	Bluetooth Low Energy
Bluetooth BR/EDR	Bluetooth Basic Rate/Enhanced Data Rate
CBOR	Concise Binary Object Representation
ClientId	Client identifier
CSRK	Connection Signature Resolving Key
DDoS	Denial-of-service attack
ESP-IDF	Espressif IoT Development Framework
GATT	The Generic Attribute Profile
HTTP	Hypertext Transfer Protocol
IoT	The Internet of Things
JSON	JavaScript Object Notation
MITM	Man-in-the-middle
MQTT	Message Queuing Telemetry Transport
mTLS	Mutual TLS
NFC	Near Field Communication
OOB	Out-of-band
QoS	Quality of Service
SIG	Bluetooth Special Interest Group
SoC	System on a chip
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UUID	Universally unique identifier

1 INTRODUCTION

The Internet of Things (IoT) refers to a collection or network of devices that communicate and exchange data with other devices and systems over the internet. According to multiple sources, in recent years, IoT has been on the rise (George Brown College 2023), and according to one of the sources, it is expected to rise to 22 billion by 2025 (Oracle 2023). IoT can be used in a myriad of ways to improve our lives, such as in home automation to automate daily tasks and in retail to track and monitor products (Amazon Web Services 2023q).

IoT devices are usually resource constrained low power and low-cost devices fitted with sensors. As these sensors produce a lot of data, it is important to store this data in an easy to access and secure place where it can be analysed. Storing this data on the device is not practical as these devices do not offer much in terms of storage, and fitting these devices with storage will bring its cost up. One way to store this data is by sending it to the Cloud, where it can be stored and analysed by more powerful computers.

The purpose of this thesis is to design and implement a temperature sensor that publishes its data to an IoT cloud platform, namely AWS IoT. I implemented this temperature sensor using a popular microcontroller with great IoT capabilities, ESP32, more specifically the ESP32-C3. Furthermore, I go over implementing some AWS IoT Core's services, such as fleet provisioning by claim, publishing data over a secure MQTT connection, and using the rules engine to save data into Amazon DynamoDB. The program receives Wi-Fi provisioning data over Bluetooth LE. All of this was implemented using Embedded C and the ESP-IDF framework.

I also implement a Flutter mobile app that is used with this temperature sensor. Using this app, the user can send Wi-Fi provisioning data to the device over Bluetooth LE and read the stored temperature data from the cloud using AWS API Gateway. The result is a document that goes over the step on how to produce a temperature sensor that uses AWS IoT as its IoT Cloud platform. Anyone interested in creating a temperature sensor, learning AWS IoT Core, or just seeing the results, can use this document and its public source code for their benefit.

This thesis aims to answer the question: how to create an ESP32 temperature sensor which uses AWS IoT Core and the ESP-IDF framework. Furthermore, this thesis hopes to shed light on good security practises when implementing an IoT applications, such as mutual authentication over TLS (mTLS). This thesis also aims to provide answers on what a temperature sensor can be used for.

2 OVERVIEW OF TECHNOLOGIES AND CONCEPTS USED

This chapter provides you with some background information on the technologies and concepts used in this thesis. All these technologies are used in some form or shape in this thesis, as such it is important that you know even the very basics of them. More information on why some of these technologies were picked over their counterparts is elaborated on in chapters 3 and 4.

2.1 Internet of Things

As mentioned in the introduction, IoT refers to a network of devices that communicate and exchange data with other devices and systems over the internet. Usually, these devices are fitted with sensors that collect data (Amazon Web Services 2023q).

As IoT devices are normally connected to the internet and handle personal or private data, it is important that security is implemented well. If not, a compromised IoT device can cause a lot of damage, as was the case with the Mirai botnet, which infected IoT devices running on an ARC processor and was used to DDoS services. Some of these IoT devices were baby monitors, vehicles, and routers.

2.2 The MQTT protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe messaging protocol that was invented in 1999 by Dr. Andy Stanford-Clark and Arlen Nipper (MQTT 2022). MQTT was designed to be easy to implement (ISO/IEC 20922:2016) and usable by resource constrained devices with limited bandwidth (HiveMQ 2023a). This, along with its small code footprint, low overhead and low power consumption makes it a great fit with IoT devices that are usually low on resources. Since MQTT runs over the TCP/IP protocol (ISO/IEC 20922:2016), any device that implements it can run MQTT (HiveMQ 2023b).

The MQTT protocol consists of two main components: the client and the broker (HiveMQ 2023b). According to the MQTT 3.1.1 specification, a client is "a program or device that uses MQTT" (ISO/IEC 20922:2016 § 1:1.2). Two types of

MQTT clients exist: subscribers and publishers. A publisher is a client that publishes messages to a MQTT topic. A subscriber, on the other hand, is a client that receives messages from a MQTT topic. A MQTT client can be both the publisher and the subscriber.

A MQTT broker is the backend software that receives messages from publishers and distributes them to subscribers (HiveMQ 2023b). It acts as an intermediary between clients. Its responsibilities include handling connections, receiving, filtering, and routing messages, session management, and authorising and authenticating clients.

2.2.1 MQTT topics and wildcards

A MQTT topic is a UTF-8 string that is used to filter messages. An example of a topic is ``sensors/temperature/livingroom``. MQTT topics support wildcards. Wildcards are a convenient way to subscribe to multiple topics simultaneously. MQTT supports single- and multi-level wildcards. (HiveMQ 2019.)

'+' is a single-level wildcard. As an example, the topic ``sensor+/livingroom`` allows the subscription of multiple topics, such as: ``sensor/temperature/livingroom`` and ``sensor/humidity/livingroom``. However, it does not allow the subscription of topics such as: ``sensor/temperature/kitchen``, ``device/temperature/livingroom``, and ``sensor/temperature/celsius/livingroom``.

'#' on the other hand, is a multi-level wildcard. It can only be used at the end of a topic string. As an example, topic ``sensor/#`` allows the subscription of topics such as: ``sensor/temperature/livingroom``, ``sensor/kitchen``, and ``sensor/humidity/basement/rightcorner``, but not topics like ``device/kitchen`` and ``camera/livingroom``.

2.2.2 Quality of Service levels

Quality of Service (QoS) provides reliability when transmitting and receiving messages over MQTT. MQTT version 3.1.1 provides three (3) QoS levels: QoS 0: at most once, QoS 1: at least once, and QoS 2: exactly once.

With QoS 0, the messages are sent only once with no acknowledgment of whether they were received. This QoS provides the least overhead and should only be used when a missed message is acceptable. For example, when sending temperature data to the cloud.

QoS 1 on the other hand send the message at least once. The sender will wait to receive a PUBACK packet within a time frame, if it does not, it will resend the message until it receives one. This QoS level is good when messages need to be transmitted at least once, but where duplicates are fine.

With QoS 2 the message transmitted will be sent once and only once. This QoS has the most overhead and as such should be used only when a message needs to be reliable send and where duplicates can cause harm. (ISO/IEC 20922:2016.)

2.2.3 Client Identifiers and message payloads

The Client identifier (ClientId) is a unique identifier that the broker uses to keep track of the client's current state (HiveMQ 2023b). According to the MQTT 3.1.1 specification, the ClientId must be a UTF-8 encoded string between 1 and 23 bytes/characters containing only the characters: "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" (ISO/IEC 20922:2016 § 3:3.1.3.1). However, the server can choose to allow ClientId's that fall outside of this rule. Such as a ClientId that is zero in length, as well as the ones that are longer than 23 bytes and contain characters that are not included in the string above.

The payload is the data that a publishing message will send. This payload is transmitted in byte format, which means the client can choose to send any type of data. The ESP32 temperature sensor sends JSON encoded data to AWS, this is because the Fleet provisioning API requires either JSON or CBOR encoded data, as well as because the AWS rules engine requires the message payload to be JSON encoded. (Amazon Web Services 2023j.)

2.2.4 Understanding MQTT operation

For a MQTT client to publish or subscribe to messages, it needs to first connect to a broker. An MQTT client is always the one to start the connection (ISO/IEC 20922:2016 § 1:1.2). A client establishes a connection with a broker by sending a CONNECT message to it and getting back a CONNACK message and a status code (HiveMQ 2023b). Once connected, this client can publish and subscribe to messages. Depending on the QoS level used, the broker may also send a packet (such as PUBACK) back to the publishing client.

The client can also subscribe to topics to receive messages. Once the client has sent a SUBSCRIBE message to the broker, it will send back a SUBACK in acknowledgement. Similarly, the client unsubscribes from topic by sending a UNSUBSCRIBE message to the broker. To end a connection, a DISCONNECT packet can be sent by the client to the server. A DISCONNECT package is not necessary to be sent to close a connection.

2.2.5 Authentication and Authorization methods

MQTT supports three authentication methods. One of these methods is authentication with a username and password. The MQTT Connect message provides optional username and password fields that can be used for authentication purposes when connecting. This is a strong way to authenticate clients. (HiveMQ 2015b.) Another method uses X.509 certificate, this is the recommended method to authenticate clients. The last method uses the Client Identifier and lets the broker decide whether to allow a specific client to connect. For example, it can allow a connection with a ClientId of "client123" but not "client1". This method of authentication is very weak and should not be used when security is of concern. (HiveMQ 2015a.)

Once connected, a MQTT client can subscribe to and publish messages on any topic, even those that it has no business with. This can lead to security problems. With proper authorization, the client can be restricted to only subscribe and publish to topics that it has business with, such as topics that include their ClientId. This is something that the broker will need to implement.

2.3 Transport Layer Security

Transport Layer Security (TLS) is an encryption protocol that enables two networks to exchange data privately and securely. It is based on the Secure Sockets Layer (SSL), which is why you often see these terms used interchangeably. (Cloudflare 2023b.) TLS was first released in 1999 and was originally intended to be version 3.1 of SSL. However, its name was changed to TLS to indicate it is no longer associated with Netscape, the developer of SSL. (Cloudflare 2023b.)

Mutual TLS (mTLS) is a method of mutual authentication. It is used to ensure that both parties are who they claim to be and prevents man-in-the-middle (MITM) attacks. This is done using TLS certificates and a technique called public key cryptography, which uses a public key and a private key. The message is encrypted with the receiving server's public key, and decrypting a message can only be done with the correct private key. Anyone can get access to a server's public key by viewing the server's TLS certificate. (Cloudflare 2023a.) AWS IoT Core supports TLS protocols 1.3 and 1.2 (Amazon Web Services 2023n).

2.4 AWS IoT Core

AWS IoT Core is an IoT service offered by AWS that can be used to connect IoT devices to other IoT devices and AWS cloud services (Amazon Web Services 2023o). The relationship between IoT devices, AWS IoT and AWS services is shown in Figure 1.

This chapter goes over the key features of AWS IoT Core and Fleet provisioning. These features play an integral role in the ESP32 temperature sensor and Flutter app.

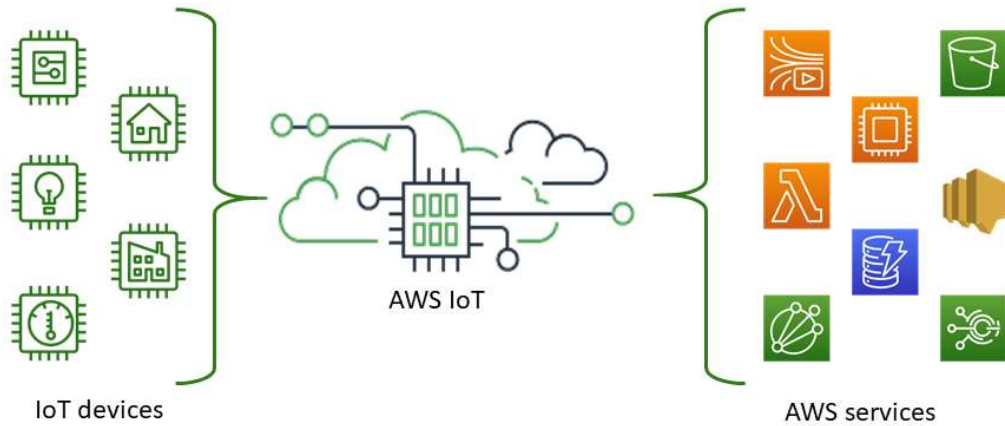


Figure 1. AWS IoT connects IoT devices to AWS services (Amazon Web Services 2023o)

2.4.1 Key features explained

AWS IoT Core offers many different features. Some of these are: AWS IoT Device SDK, Device Advisor, Device Gateway, Message Broker, Device Shadow, and Rules Engine. These features are explained below.

The AWS IoT Device SDK provides an easy and quick way to connect devices to AWS IoT Core. These SDKs support three different messaging protocols: MQTT, HTTP and WebSocket. AWS IoT Device SDKs support multiple different programming languages: C++, Python, JavaScript, Java, and Embedded C. All Device SDKs are open source and under the MIT license.

Device Advisor is a cloud-based test capability that provides pre-built tests for IoT device validation during development. It is used to accelerate IoT device software development.

According to AWS, the Device Gateway serves as the entry point for IoT devices connecting to AWS (Amazon Web Services 2023b). It manages connected devices and ensures that devices can securely communicate with AWS IoT Core. The Device Gateway supports the previously mentioned messaging protocols, MQTT, WebSocket, and HTTP. Furthermore, according to AWS, Device Gateway can be scaled to automatically support over a billion devices without requiring you to manage any infrastructure (Amazon Web Services 2023b).

AWS' Message Broker is a Pub/Sub messaging agent based on the MQTT version 5.0 standard that securely transmits messages between devices. It supports one-to-one, one-to-many and everything in between. AWS IoT Core uses Transport Layer Security (TLS) to securely send messages. AWS IoT Core offers support for SigV4, X.509 certification and custom authentication, such as token-based authentication. The Message Broker is responsible for device authentication when using AWS IoT authentication.

The AWS IoT Device Shadow is a service that saves a device's state to the cloud so that it is always "online". A device posts its state to its shadow which resides in the cloud, this allows other services to read this device's state regardless of whether it is online/connected or not. Apps and services can also request changes to this shadow, once the device comes online again, it will receive its state from the cloud and match its state to its shadow. (Amazon Web Services 2023c.)

The Rules Engine routes messages to AWS services, such as Lambda and DynamoDB (Amazon Web Services 2023m). With the Rules Engine, it is possible to save temperature data to a DynamoDB database.

2.4.2 Fleet provisioning options

AWS IoT Core offers multiple different ways to provision devices, Fleet provisioning being one of these methods. Fleet provisioning can be done using a claim or by a trusted user (Amazon Web Services 2023l).

When provisioning by claim, its manufacturer saves certificates and a private key on the device, which the device will use to connect to AWS IoT Core and register itself with (Amazon Web Services 2023l). These claim certificates are usually very restrictive in permissions and should be configured to only be able to register a device (or 'thing' as it is called by AWS) and nothing more.

Provisioning by a trusted user is when a trusted user registers a device/thing in its deployable location. The user uses their IAM account to provision this device/thing and saves the generated certificates and keys on the device (Amazon Web Services 2023l).

AWS IoT Core also offers pre-provisioning hooks that can be used to validate parameters passed before the device is provisioned (Amazon Web Services 2023k). For example, checking that the device's serial number is valid.

2.5 AWS Amplify

AWS Amplify is AWS' complete solution for developing full-stack applications that use AWS Cloud. One of its features is a cross-platform backend for mobile and web. In addition to that, AWS Amplify also supports building frontend UIs with AWS Amplify Studio and hosting web apps with AWS Amplify Hosting.

AWS Amplify Studio allows UI development using a visual development environment. With it, a developer can build a web or mobile app's frontend with minimal coding. (Amazon Web Services 2023a.) Amplify Studio also offers integration with Figma. Allowing a developer to import a Figma design to generate React code. (Amazon Web Services 2023h.)

Amplify Libraries are open-source client libraries provided by AWS Amplify that allow mobile and web developers to easily interact with their backends. As of right now, Amplify Libraries offers nine different integrations: JavaScript, React, React Native, Angular, Vue, Next.js, Android, Swift and Flutter. Depending on the integration, some features may be missing.

2.6 Flutter

Flutter is an open-source, cross-platform software development framework developed by Google. It supports mobile (iOS and Android), web, desktop (Windows, macOS, and Linux), and embedded. Flutter programs are written in the Dart programming language.

Flutter apps UIs are built out of widgets. Widgets are the core building blocks of Flutter applications. Flutter uses its own graphics engine to build these widgets instead of relying on the platform's built-in widgets. It is possible to make an app look like its native counterpart using Flutter Cupertino widgets for iOS and Material Components for Android. (Amazon Web Services 2023p.)

2.7 ESP32 microcontrollers and the ESP-IDF development framework

ESP32 is a series of system-on-a-chip (SoC) microcontrollers by Espressif. They are popular due to their low price, Wi-Fi and Bluetooth capabilities, and Arduino compatibility. ESP32 chips have been engineered to be ultra-low power enough to be usable in mobile, wearable and IoT applications (Espressif Systems 2023a).

The ESP32 family of chips is comprised of many different chips with different capabilities and their own focus of functionality, with the ESP32-S series focusing more on Artificial Intelligence of Things (AIoT) and the ESP32-C series being a cost-effective RISC-V chip.

The Espressif IoT Development Framework (ESP-IDF) is used to develop ESP32 devices. It is open-source and works on all their chips (except for their newest ones, which do not have support yet). C/C++ is the programming language used. (Espressif Systems 2023b.)

2.8 Bluetooth Low Energy

Bluetooth Low Energy (LE) is a wireless technology aimed at low power operation. Bluetooth LE is not a replacement to the classic Bluetooth BR/EDR (Basic Rate/Enhanced Data Rate), rather an alternative, with one of its goals being power efficiency (Woolley 2023, 7).

This chapter explains some of the inner workings of Bluetooth LE, such as the Generic Attribute Profile, Universally Unique Identifiers and Bluetooth LE security, all of which make it possible for Bluetooth LE devices to communicate effectively. This chapter also explains some differences between the classic Bluetooth BR/EDR and Bluetooth LE.

2.8.1 The Generic Attribute Profile and Universally Unique Identifier (UUID)

The Generic Attribute Profile (GATT) is a layer of the Bluetooth LE stack that defines higher level data types known as services, characteristics, and descriptors. These data types are used to exchange data between two or more devices. (Woolley 2023, 67.)

Services are a way to group characteristics together. A service may have one or more characteristics. Characteristics are individual pieces of data. These characteristics have a type, a value and a set of properties that indicate the data is used. For example, the Heart Rate Measurement characteristic has the notify property that notifies when its data changes (Bluetooth SIG 2011, 9-10). Descriptors are optional pieces of metadata that some characteristics have. They help in standardizing the interpretation of characteristic values. For example, a temperature data characteristic's descriptor could specify the temperature unit (Celsius or Fahrenheit).

All services, characteristics, and descriptors have a universally unique identifier (UUID) that identifies them (Woolley 2023, 9). These UUIDs are usually either 16-bit or 128-bit. 16-bit UUIDs are defined by the Bluetooth Special Interest Group (SIG). An example of a 16-bit UUID is the heart rate service, which has a UUID of 0x180D (Bluetooth SIG 2023). 128-bit UUIDs are custom services, characteristics, and descriptors. Anyone can implement these UUIDs (Woolley 2023, 68).

2.8.2 Differences between Bluetooth Classic and Bluetooth Low Energy

While both operate on the 2.4GHz band, Bluetooth LE does so over 40 channels with 2MHz spacing. Bluetooth Classic, on the other hand, does so over 79 channels with 1 MHz spacing. Bluetooth Classic is faster with support for data rates up to 3 MB/s, in contrast with Bluetooth LE's 2 MB/s.

Bluetooth Classic supports only Point-to-Point communication, while Bluetooth LE supports Point-to-Point, Broadcast and Mesh. Bluetooth LE also supports positioning features such as presence, proximity, and direction.

2.8.3 Security features in Bluetooth Low Energy

Pairing is the process that creates secret keys that are shared between the devices (Bluetooth SIG 2021, 266). These keys are used for different purposes, such as device authentication, encryption, and identity. In contrast to Bluetooth Classic (BR/EDR), these keys are generated by the host (e.g., a smartphone).

Bluetooth Classic's key generation is performed by the controller. Pairing is always initiated by the client. (Bluetooth SIG 2021, 274.)

Bonding is the act of storing secret keys for subsequent connections (Bluetooth SIG 2021, 266). This allows devices to recognise and authenticate each other without repeating the key exchange.

Device authentication is the process in which two devices' identities are verified (share the same keys). Data is signed using a Connection Signature Resolving Key (CSRK), and the receiver verifies that this signature is correct with their CSRK. (Bluetooth SIG 2021, 266, 274-275.)

Bluetooth LE data is encrypted using AES-CCM cryptography. Message integrity, on the other hand, is used for protection against message forgery. (Bluetooth SIG 2021, 266, 274.) Bluetooth LE uses four association models: Just Works, Numeric Comparison, Out of Band, and Passkey Entry. Numeric Comparison can only be used with LE Secure Connections. (Bluetooth SIG 2021, 274.)

The Numeric Comparison association model can be used when both devices can display numbers and select "yes" or "no". Both devices display the same six-digit number, if this number is not the same on both devices, the user will select "no" and the pairing is not successful. (Bluetooth SIG 2021, 270.)

The Just Works association model is used when at least one of the devices does not have I/O capabilities (cannot display or input numbers). When pairing, the device with I/O capabilities is asked to accept the connection. Just Works does not protect against passive eavesdropping. (Bluetooth SIG 2021, 271, 274.)

With the Out-Of-Bound (OOB) association model, the secret keys are first exchanged by another method other than Bluetooth LE. An example of another method is Near-Field Communication (NFC). (Bluetooth SIG 2021, 271.)

The Passkey Entry association model is used when one device has input capabilities and the other has output capabilities. The output capability device displays a six-digit number, which the user will then input into the other device. (Bluetooth SIG 2021, 272.)

The use of these models depends on the I/O capabilities of the devices. Numerical Comparison and Passkey Entry offer protection against man-in-the-middle attacks. (Bluetooth SIG 2021, 269-270.)

3 DESIGN OF THE SOFTWARE

In this section, I go over how the programs work. As mentioned before in the introduction chapter, the software consists of two major programs. An ESP32 temperature sensor program and a Flutter app.

The ESP32 temperature sensor program publishes temperature data to AWS IoT core, which in turn inserts it into a DynamoDB table via an AWS Rule. The Flutter app reads temperature data from AWS Cloud and provisions the ESP32 temperature sensor by sending the Wi-Fi data and the device's name over BLE to the ESP32 device. The ESP32 temperature sensor needs the data sent over BLE to connect to Wi-Fi and register itself with AWS IoT Core.

3.1 ESP32 temperature sensor program functionality

This section goes over the workings of the ESP32 temperature sensor program, the AWS cloud side of things will be gone over later. The ESP32 temperature sensor program consists of three states. The program changes states depending on whether the device has the required data to advance in its storage.

State 1 heavily depends on the Flutter app and will not proceed further without the required data provided by the app. States 2 and 3, on the other hand, do not depend on the Flutter app and instead depend on the Wi-Fi station and AWS cloud.

3.1.1 State 1: Device provisioning

As you may be able to guess, this is the state that the program takes on its very first start up. When in state 1, the program creates a BLE server and waits for data to be sent from the Flutter app. The data that the sensor waits for are the Wi-Fi SSID, Wi-Fi password and device name.

The program can also transition into this state if it does not find a Wi-Fi SSID or password in its Non-Volatile Storage (NVS). Once the device has received all the required data, it will attempt to connect to the Wi-Fi station with the SSID and password it has received. If the connection succeeds, it saves the data into NVS

and reboots; otherwise, it will just reboot and start all over again (as shown in Figures 2 and 3).

The BLE server uses the Just Works association model; this is done because the ESP32 device does not have any default I/O capabilities, and while some could be added, it is beyond the scope of this project. The data sent over is encrypted by Bluetooth LE.

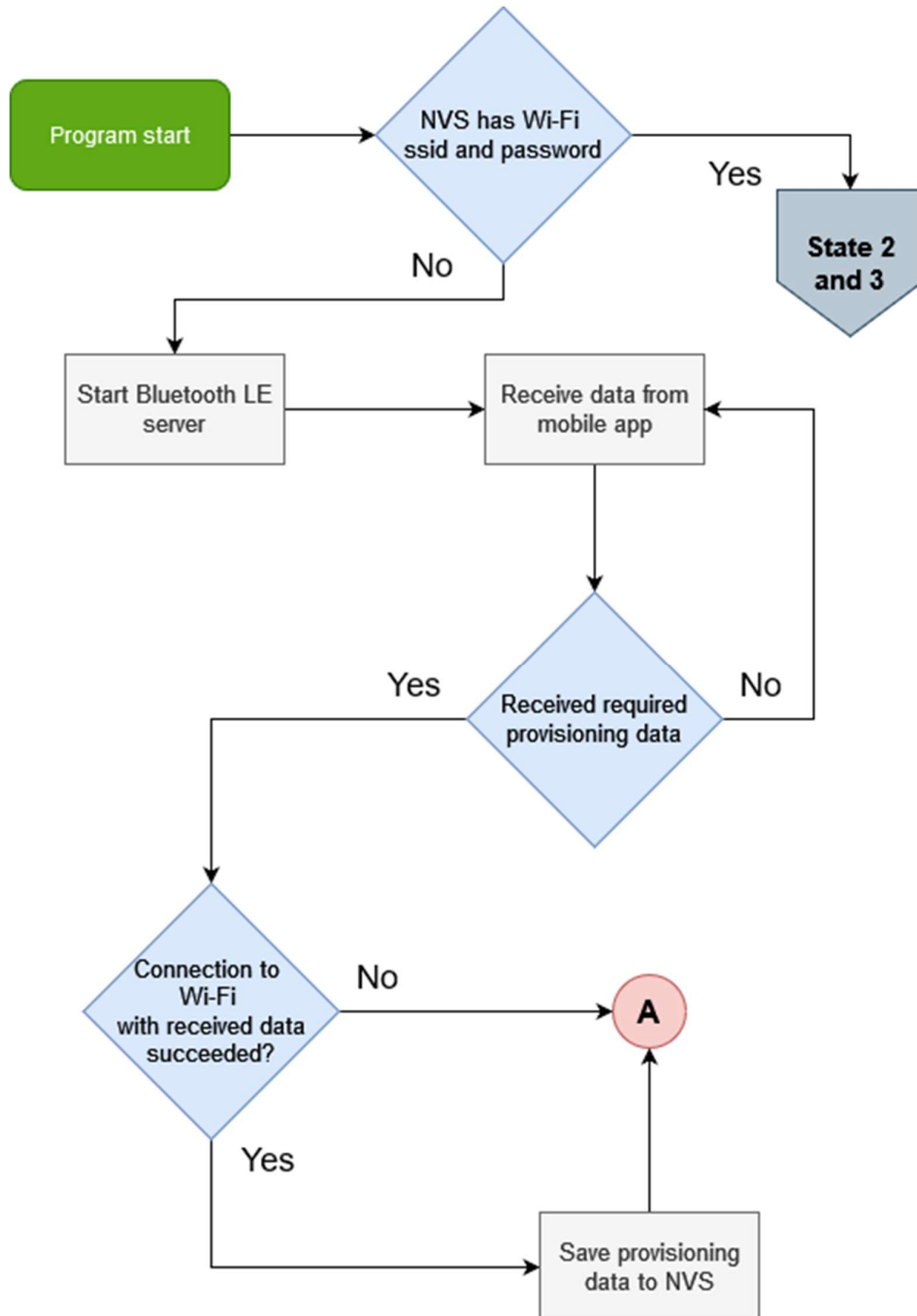


Figure 2. State 1 device provisioning flowchart

Figure 3 is the continuation of the 'A' connector that Figures 2, 4 and 5 use. It describes the device reboot flow that is used to reset the device's state. This is most likely not a good way to reset the device's state, as it takes time, but it is the easiest to implement.

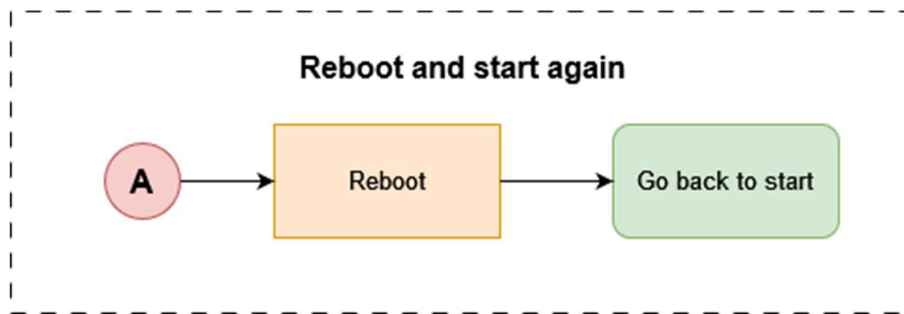


Figure 3. Reboot and start again flowchart

3.1.2 State 2: Register sensor and State 3: Publishing temperature data

After state 1, the program transitions into state 2. In state 2, the device proceeds to register itself with AWS IoT Core. As shown by Figure 4, to proceed to this state, the device needs to have a Wi-Fi SSID and password inside its NVS, be able to connect to the Wi-Fi station, and not have been registered before. If the device fails to connect to Wi-Fi, it erases the Wi-Fi SSID and password from its NVS and regresses back to state 1.

Once the device has registered itself with AWS IoT Core and its credentials have been saved into NVS, it reboots itself. Provided all has gone well (certificates have been saved into NVS), it proceeds into state 3 and starts publishing temperature data.

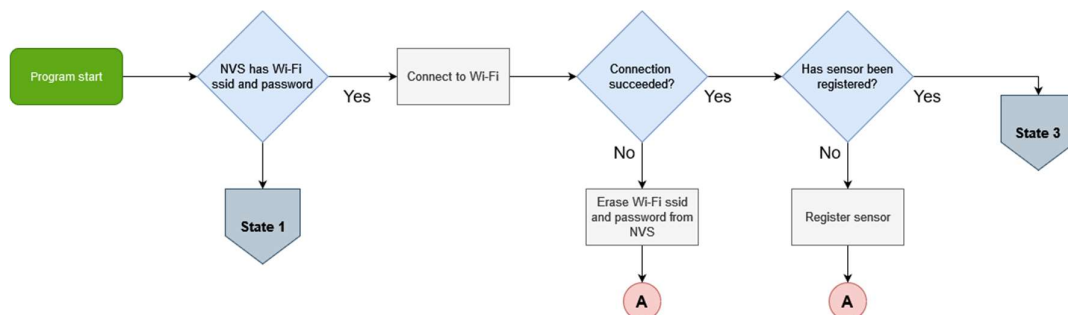


Figure 4. State 2 register sensor flowchart

Once in the third and last state, the device proceeds to publish temperature data over MQTT to AWS IoT Core. State 3 goes through the same check as state 2 (Figure 5), checking the existence of Wi-Fi credentials inside NVS and connecting to Wi-Fi. Unlike state 2, state 3 should have the required credentials to connect to AWS IoT Core and publish data. If it does not, it regresses back into state 2. How the device publishes temperature data to the AWS cloud is explained in chapter 4.1.

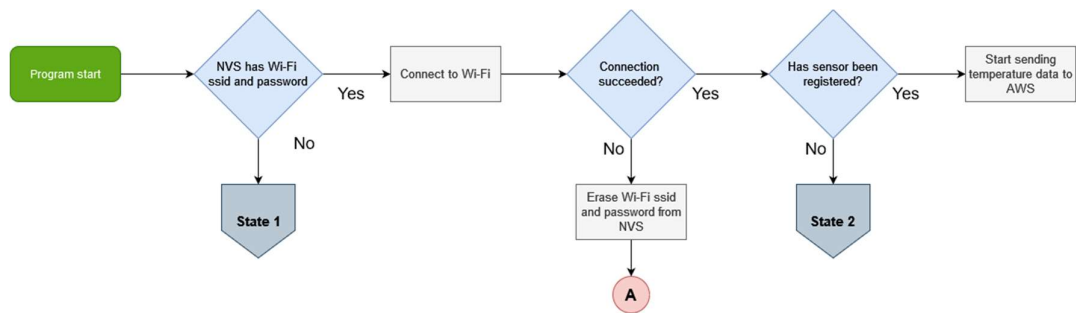


Figure 5. State 3 publishing temperature data flowchart

3.2 AWS IoT Core certificates

AWS IoT offers the ability to connect and authenticate clients using X.509 certificates. These certificates can be generated by AWS IoT, like in this thesis, or signed by another certificate authority (CA). X.509 certificates are preferred over other methods as they provide more security. For example, X.509 allows the manufacturer to burn private keys onto the device to ensure that no sensitive cryptographic material leaves the device. (Amazon Web Services 2023r.)

In my case, AWS IoT generates X.509 certificates for me. I then flash these certificates onto the ESP32 device and use them to make the initial connection to AWS IoT Core, execute device registration, and receive the device's own X.509 certificates, which it uses for subsequent connections and for publishing temperature data. After the newly registered device has received its own certificates, it no longer has a need for the X.509 certificates that were initially flashed onto it.

3.3 Flutter app functionality

The Flutter app serves two purposes: reading temperature data from the DynamoDB table and providing data over BLE. First, to use the app, the user needs to sign in to an account. A user cannot create an account and must request an administrator to create one for them. Once an account has been created and handed over to the user, the user can sign in to it and get access to the rest of the app. The home screen of the app displays all registered devices. The user can decide to select one of these devices to see all the temperature data that it has posted.

As mentioned before, the Flutter app is used to provision the ESP32 device. This is done by connecting to the ESP32 device's Bluetooth LE server and sending it the Wi-Fi SSID, password, and the name that the device will use when registering itself with AWS IoT Core. Figure 6 shows the program flow of the Flutter app.

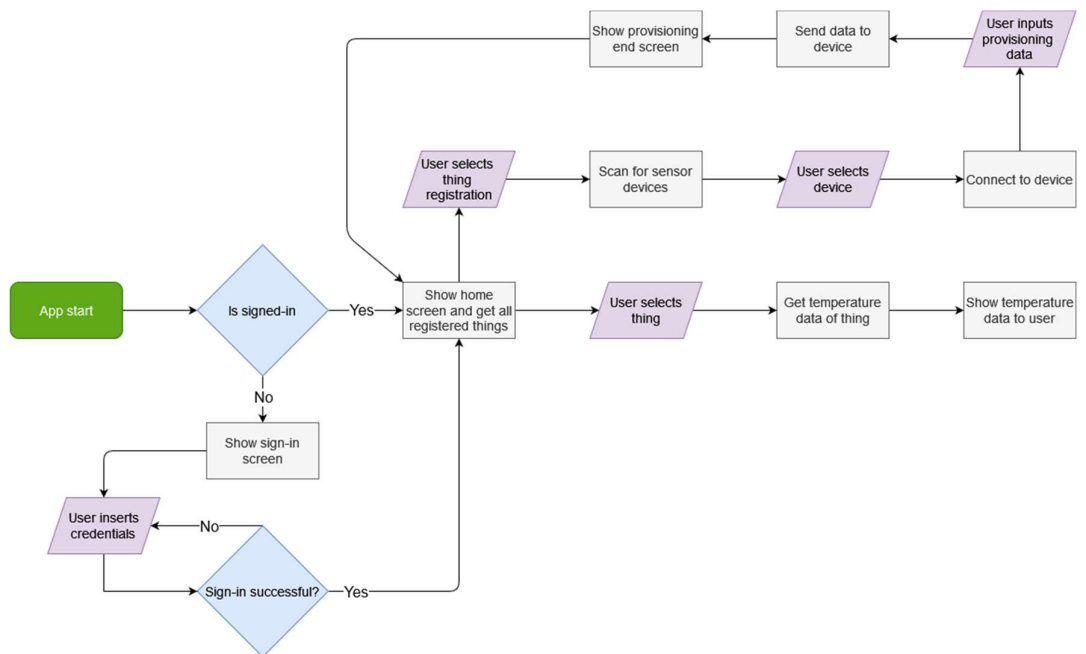


Figure 6. Flutter app flowchart

Figure 7 shows the relationship that the Flutter app has with AWS Cloud. AWS Amplify is used to implement user authentication. AWS Amplify will in turn use AWS Cognito. The app uses AWS API Gateway to fetch temperature data from the DynamoDB table. AWS API Gateway provides an easy-to-use interface and

a way to authorize API calls. Just like with signing in with AWS Amplify, API Gateway will use AWS Cognito.

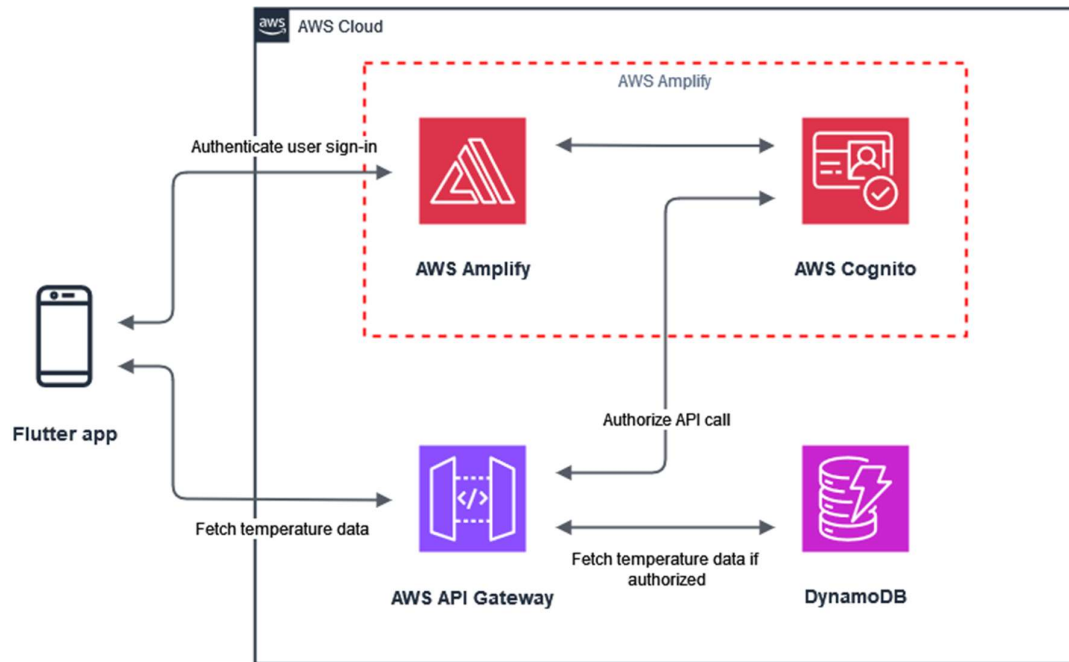


Figure 7. Flutter app relationship with AWS Cloud

4 SOFTWARE IMPLEMENTATION

This chapter goes over the implementation of the temperature sensor and the Flutter app. The prototype program simulates a real-life temperature reading, meaning that no actual temperature sensor is used. However, a real sensor can be easily added.

To complement this thesis, a separate document was created that guides the reader through the steps to setup the temperature sensor and the Flutter app for themselves (Appendix 1). This chapter references this guide, and some of its contents are elaborated on. Furthermore, this chapter also elaborates on parts not covered in the guide.

4.1 Publishing temperature data to AWS IoT Core

The ESP32 sensor publishes its temperature data to AWS IoT Core through MQTT. In my program, the topic to which the device publishes is ‘device/thingName/temperature/data’ with ‘thingName’ being the name of the device. For example, with a name of ‘thing1’, its topic would be ‘device/thing1/temperature/data’.

An AWS IoT Rule that listens to the topic of ‘device+/temperature/data’ is created; once it receives a message, it invokes a DynamoDBv2 action and inserts the received data into a DynamoDB table (Figure 8). With the wildcard character of ‘+’, the rule can subscribe to multiple different topics, such as ‘device/thing1/temperature/data’ and ‘device/thing2/temperature/data’, to name a few.

AWS IoT Core has many rule actions; in this case, DynamoDB and DynamoDBv2 actions are the ones that matter (Amazon Web Services 2023d). Both DynamoDB and DynamoDBv2 actions insert the message’s data into a DynamoDB table. DynamoDB action inserts the message’s data into one column on a DynamoDB table, while DynamoDBv2 inserts the message into multiple columns.

Table 1. DynamoDB action insert output

Primary key	Sort key	Data
primary key	timeStamp	{"Temperature": {"N": 31}, "Humidity": {"N": 73}}

As an example, suppose that a message with a payload of '{"Temperature": 31, "Humidity": 73}' is published. The DynamoDB action would insert this message into a DynamoDB table in the form shown in Table 1, while the DynamoDBv2 action would insert the same data in the form shown in Table 2.

Table 2. DynamoDBv2 action insert output

Primary key	Sort key	Temperature	Humidity
primary key	timeStamp	31	73

As you can see, the DynamoDBv2 action inserts the data into its own columns; this is preferable to the DynamoDB action and is used as this project's AWS Rules action. The AWS IoT Rule uses the following SQL statement to insert temperature data into the DynamoDB table.

```
SELECT temperature, cast(timestamp() as String) as timestamp, cast(topic(2) as String) as thingname, FROM 'device/+ /temperature/data'
```

This statement extracts the temperature value from the payload and the device's name from the topic with the 'topic(2)' function (Appendix 1 17(37)). The statement's result is then passed to the DynamoDBv2 action by AWS, which in turn saves the data to a DynamoDB table.

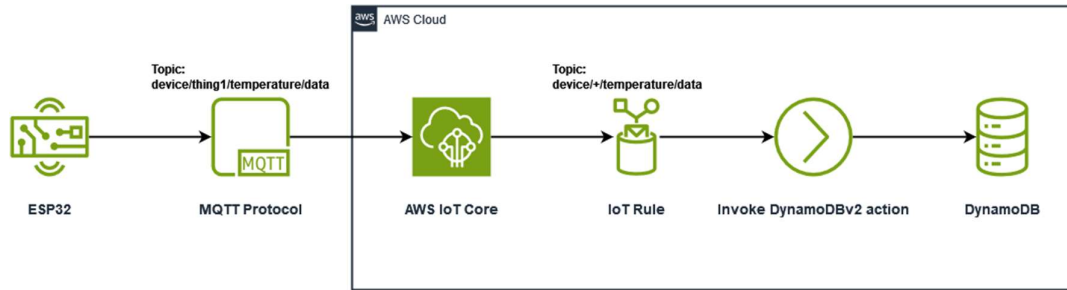


Figure 8. EPS32 publishing temperature data to AWS IoT Core

The piece of code shown in Figure 9 has been taken from the ‘temperature_publish_task’ function located in the esp32-temperature-sensor repository’s mqtt.c file, line 551. It publishes temperature data to AWS IoT Core. To explain the code a little, line 2 creates the publish topic by substituting ‘%s’ with the device’s name. Line 4 creates the payload of the message, and line 5 does the actual publishing, with ‘esp_mqtt_client_publish’ being a function provided by ESP-IDF. The publishing is done with a QoS of 0.

```

1. ...
2. snprintf(temperature_topic, 256, "device/%s/temperature/data", thing_name);
3. ...
4. snprintf(payload, 128, "{\"temperature\": %d}", temperature);
5. msg_id = esp_mqtt_client_publish(client, temperature_topic, payload, 0, 0, 0);
6. ...
  
```

Figure 9. Function that publishes temperature data to AWS IoT Core

As mentioned at the start of this chapter, the prototype program does not publish any real temperature readings but simulated ones. In the likely case that you want to publish real readings, just make the ‘temperature’ variable hold the value, and it will be published to the cloud.

4.2 Device registration using fleet provisioning by claim

As mentioned in the introduction of this thesis, the program uses Fleet provisioning by claim to register devices with AWS IoT Core. Fleet provisioning by claim allows the user to flash the same TLS certificate on all the devices and lets the devices register themselves with AWS IoT Core. This saves time by not having

to manually register a device with AWS IoT Core and flashing the certificates (that are unique for each device) to the device before the IoT device is delivered.

One of the more important decision factors when choosing between Fleet provisioning and regular device registration is security. With regular device registration, security is ensured by manually creating device specific certificates, while with Fleet provisioning by claim this is done by making the initial certificates as restrictive as possible, allowing only the essential permissions to register a device with AWS IoT Core. Should you notice any misuse of the certificates, you can revoke them and prevent them from being used to register new things. Fleet provisioning does present a larger risk in the event of compromised claim certificates; as such, you should go for manual device registration if possible.

It is important that the relationship between certificates and AWS IoT policies be understood. AWS IoT uses X.509 certificates to authenticate TLS clients (in this case, the client is the IoT device). On the other hand, AWS IoT Core policies are used to authorise clients. AWS IoT Core policies can be attached to X.509 certificates, Amazon Cognito identities and device groups.

As mentioned before in Chapter 3.3, this project uses X.509 certificates. X.509 certificates are created for Fleet provisioning by claim and an AWS IoT Core policy is attached to these certificates. The attached policy restricts the device's access to only the essentials that it needs to register itself with AWS IoT Core. The registration happens using the device provisioning MQTT APIs provided by AWS IoT Core. Using these APIs, the device registers itself with AWS IoT and receives its own X.509 certificates with a different, more permissive policy attached. The device uses this more permissive policy to publish temperature data to AWS IoT Core.

The device provisioning MQTT APIs provide three operations: `CreateCertificateFromCsr`, `CreateKeysAndCertificate`, and `RegisterThing` (Amazon Web Services 2023g). This project uses the `CreateKeysAndCertificate` and `RegisterThing` operations. The `CreateKeysAndCertificate` operation creates and returns X.509 certificates (created and signed by Amazon Root certificate authority) and a `certificateOwnershipToken`. The `certificateOwnershipToken` is then supplied to the `RegisterThing` operation, which registers the device with AWS IoT Core. If the

RegisterThing operation succeeds, the device is free to use the X.509 certificates provided by the CreateKeysAndCertificate operation. If the operation fails, the X.509 certificates provided by the CreateKeysAndCertificate operation will not work.

The CreateKeysAndCertificate operation provides the following topics:

- `$aws/certificate/create/payload-format/accepted`
- `$aws/certificate/create/payload-format/rejected`
- `$aws/certificate/create/payload-format`

RegisterThing operation provides the following topics:

- `$aws/provisioning-templates/templateName/provision/payload-format/accepted`
- `$aws/provisioning-templates/templateName/provision/payload-format/rejected`
- `$aws/provisioning-templates/templateName/provision/payload-format`

Here are the steps on how the device registers itself with AWS IoT Core. First, it connects to the MQTT endpoint of AWS IoT Core. Next, the device subscribes to the `'$aws/certificate/create/payload-format/accepted'` topic; in this case, `'payload-format'` is JSON, but AWS does support `'cbor'` as well. Similarly, the device also subscribes to the `'$aws/certificate/create/json/rejected'` topic. After subscribing to the topics, the device publishes a message to the `'$aws/certificate/create/json'` topic with an empty payload. Next, the device parses the data received from the `'$aws/certificate/create/json/accepted'` topic. The data received looks like the following JSON, with the "string" parts replaced with a value provided by AWS IoT Core.

```
{
  "certificateId": "string",
  "certificatePem": "string",
  "privateKey": "string",
```

```
"certificateOwnershipToken": "string"
}
```

Next, the device subscribes to the '\$aws/provisioning-templates/templateName/provision/json/accepted' topic. In this case, 'templateName' will correspond to the name of the Fleet provisioning template. Similarly, the device subscribes to the '\$aws/provisioning-templates/temperature-sensor-template/provision/json/rejected' topic. The device then publishes a message to the '\$aws/provisioning-templates/temperature-sensor-template/provision/json' topic with the following JSON payload. The value of CertificateOwnershipToken is the one that was previously received from the '\$aws/certificate/create/json/accepted' topic. ThingName is the name of the device doing the registration.

```
{
  "certificateOwnershipToken": "string",
  "parameters": {
    "ThingName": "string",
  }
}
```

If the device gets a message from the '\$aws/provisioning-templates/temperature-sensor-template/provision/json/accepted' topic, it means that the device registration was successful and the device can proceed by saving certificateId, certificatePem, privateKey and thingName into NVS. Alternatively, if the device gets a message from the '\$aws/provisioning-templates/temperature-sensor-template/provision/json/rejected' topic, it means that the registration failed.

When publishing important messages such as these, it is optimal to do so with a QoS level of 2. This would ensure that the messages would be received and published exactly once. With QoS 0, should the message not be received even once, the user would have to do the whole device registration process again. With QoS 1, should the message be received more than once, we would create unnecessary certificates and possibly introduce security vulnerabilities. Unfortunately, AWS IoT Core does not support QoS 2 (Amazon Web Services 2023i). As such, QoS 1 with a way to handle the possibility of multiple replies should be

used. To not overcomplicate things, the prototype program sends all messages with a QoS of 0.

4.3 DynamoDB table structure

The DynamoDB table that stores the temperature data consists of three columns. These columns are 'thingname' which is the primary key and of type string, 'timestamp' which is the sort key and of type string, and 'temperature' which is of type number. (Appendix 1 15(37).)

Setting the device's name (thingname) as the primary key allows us to query the table for matches for a specific device. Using the Query action is preferable to the Scan action, as it is the more efficient action (Amazon Web Services 2023e). This is because the Query action performs the lookup using the hash value of the primary and/or secondary keys. This also means that the Query action is less flexible than the Scan action, which does not rely on hash values.

4.4 AWS API Gateway functionality

As mentioned in Chapter 3.3, the Flutter app uses the AWS API Gateway to fetch temperature data from the temperature data DynamoDB table. AWS API Gateway is used because it allows for easy interfacing with DynamoDB. My project's API Gateway has two GET endpoints. Both endpoints have a mapping template that describes the actions, variables, and functions that API Gateway needs to interface with DynamoDB.

The first endpoint is '/temperature/{thingName}', which fetches all temperature data of the device specified in '{thingName}'. This data is fetched from DynamoDB with the Query action. For example, '/temperature/thing1' fetches all temperature data published by the device 'thing1'. The following JSON is the mapping template for this endpoint:

```
{
  "TableName": "temperature_data",
  "PrimaryKey": "thingname",
  "KeyConditionExpression": "thingname = :val",
```

```

"ExpressionAttributeValues": {
  ":val": {
    "S": "$input.params('thingName')"
  }
}

```

This template does the following:

- “TableName” provides the DynamoDB table that is queried.
- “PrimaryKey” is this table’s primary key.
- “KeyConditionExpression” describes the condition that must be passed for the item to be retrieved; in this case, ‘thingname’, which is the primary key of this table, must be equal to the value stored in the variable ‘:val’.
- “ExpressionAttributeValues” is a list of variables. This template only has the ‘:val’ variable, which holds the value of the `$input.params('thingName')` function. `'$input.params('thingName')` fetches the ‘thingName’ parameter from the URL.

The second endpoint, `/things`, fetches all devices that have published temperature data. This endpoint returns only the names of these devices and not any of their temperature data. It does so by executing the DynamoDB Scan action. This endpoint is used by the Flutter app to list all registered devices. The following JSON is the mapping template for this endpoint:

```

{
  "TableName": "temperature_data",
  "ProjectionExpression": "thingname"
}

```

This template does the following:

- “TableName” provides the DynamoDB table, which is scanned.

- “ProjectionExpression” describes the attributes (columns) that are retrieved by the scan. Meaning only the ‘thingname’ column is retrieved.

The API is secured by an Amazon Cognito user pool. This is done to prevent unauthorized users from accessing the temperature data posted by these devices. To invoke this API, the user must pass an identity token inside the ‘Authorization’ HTTP header. For the user to get this token, they must be logged in. (Amazon Web Services 2023f.) Figure 10 shows the flow of actions that happen when the Flutter app calls the ‘/temperature/{thingName}’ API endpoint. It also shows an example response to this call.

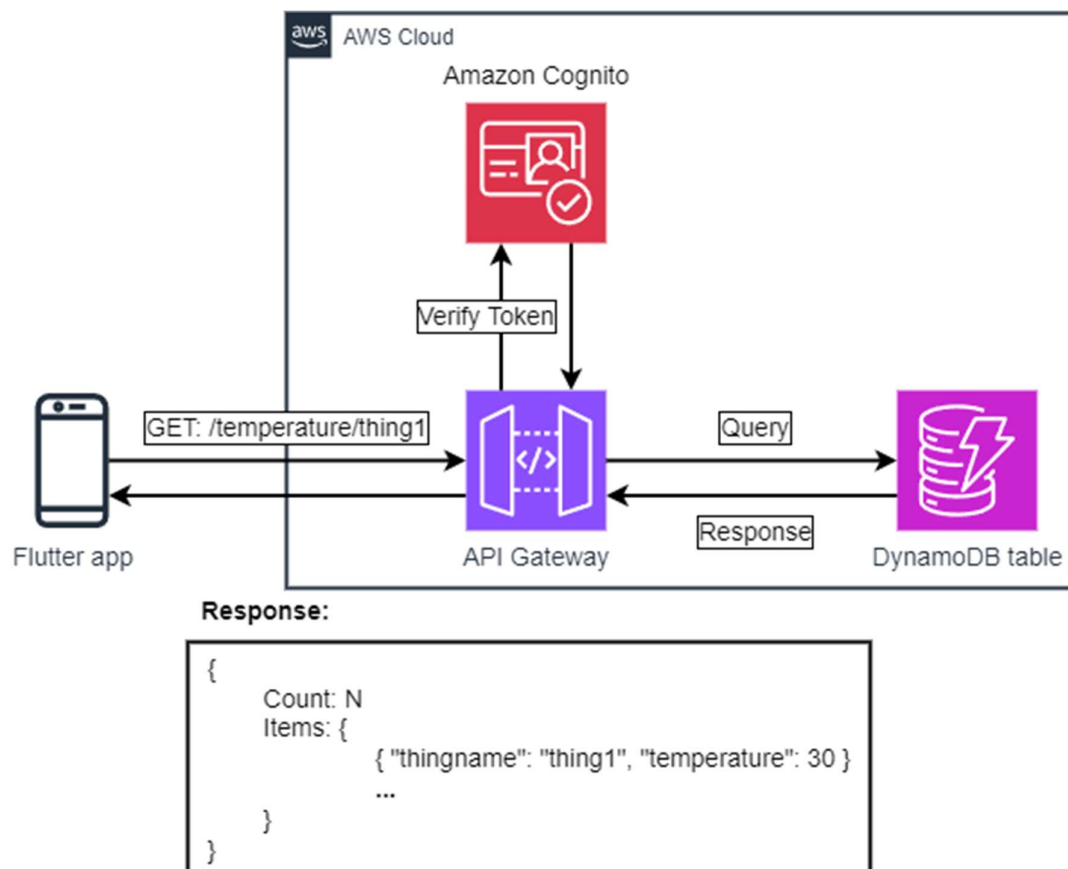


Figure 10. Flutter app calling API endpoint

AWS Amplify is used to create the Amazon Cognito user pool. AWS Amplify is a convenient way to automatically create the AWS services and resources needed for app authentication. Furthermore, AWS Amplify provides an easy-to-use library that can be used to interact with Amazon Cognito, saving you time from writing

your own functions. However, the user pool will need to be attached to the API manually.

5 CASE STUDY: DATA CENTRE TEMPERATURE MONITORING

For modern businesses, data centres play an important role by serving as a centralised facility where they can store, manage, and analyse their data efficiently and securely. Often this data is something the business generates and relies on to provide their services. As such, for businesses to keep providing their services, it is vital that the data centre(s) that they rely on work without problems.

One of the key challenges that data centres face is heat management. The servers of data centres generate massive amounts of heat that, if not properly controlled, could lead to equipment overheating and subsequent equipment failure. However, just keeping equipment from overheating is not enough, cooling needs to be done efficiently to cut down on energy consumption, operating costs, and environmental impact (Green Revolution Cooling 2022).

This chapter is a case study on how the ESP32 temperature sensor designed and implemented in this thesis could be used for data centre temperature monitoring. This case study references studies that use equipment like the ESP32 temperature sensor produced in this thesis.

5.1 Problem statement

Data centres employ many different methods to cool their equipment, legacy data centres usually cool their equipment with air by using a Computer Room Air Conditioning (CRAC) unit, which maintains the temperature and humidity (Ebrahimi, Jones & Fleischer 2014, 626). Many newer data centres cool their equipment using liquid-based cooling (Ebrahimi et al. 2014, 626-627). With liquid-based cooling, heat is absorbed by a liquid and transported away from the equipment. Regardless of the methods used, one thing that stays constant is the need to monitor the temperature of the equipment in real-time.

Without real-time temperature monitoring, the heat generated could lead to equipment overheating. Equipment overheating will reduce the lifespan of the equipment, increasing the risk of hardware failure and degrading performance. All of the above will negatively impact businesses that rely on the data centre.

Data centres also consume huge amounts of energy, which, according to Rozite, Bertoli and Reidenbach (2023), is estimated to be 240-340 TWh, which is around 1-1.3% of global electricity. According to Bangalore et al. (2023) and Polonelli, Brunelli, Bartolini and Benini (2019, 169), 40-60% of the energy consumed by data centres is used just for cooling. With real-time temperature monitoring, cooling could be made more efficient, which would reduce energy consumption.

5.2 Solution

The solution for these problems is real-time temperature monitoring. The data produced could be used to diagnose possible problems such as equipment overheating, ensuring equipment reliability, optimizing power efficiency, and compliance with regulations such as the European Code of Conduct for Energy Efficiency in Data Centers. (Polonelli et al. 2019, 169.)

The ESP32 temperature sensor designed and implemented in this thesis could be used to capture data centre temperature readings in real-time. This data, which is published to the cloud, can then be analysed and monitored. Furthermore, since the data is published to the cloud in real-time, it can be monitored remotely. With some tweaking, real-time alerts can be implemented to catch temperature deviations. And, thanks to AWS IoT Core and AWS IoT fleet provisioning, the sensor is easy to scale and deploy without having to manage infrastructure.

5.3 Results

The ESP32 temperature sensor prototype designed and implemented in this thesis has not been deployed in a real-world data centre, as such, there are no concrete results to be shown. However, similar IoT temperature sensors have been previously deployed by others. The IoT sensors in question differ in the hardware and communication protocols used, however, the main aspect of data centre temperature monitoring stays the same. As such, I believe that the results obtained from these studies would closely resemble those achieved by deploying the ESP32 temperature sensor in a real-world data centre.

In the study *Wireless sensor network for data-centre environmental monitoring*, the authors deploy temperature and humidity monitoring sensors into a data centre and run them for 24 hours. Their results go on to show that wireless sensors can be an effective tool for data centre temperature monitoring, being able to catch temperature variations and areas where cooling is insufficient or excessive. (Rodriguez et al. 2011, 3-5.)

In another study conducted by Thomas Scherer, Clemens Lombriser, Wolfgang Schott, Hong Linh Truong & Beat Weiss, a similar temperature sensor is deployed into an IBM data centre. The results of this study show how temperature sensors can detect hot spots, which are areas with a higher temperature than their surroundings. These hotspots are undesirable and pose the risk of equipment overheating, and as such, need to be eliminated. (Scherer, Lombriser, Schott, Truong & Weiss 2012, 6-7.)

5.4 Conclusion

In conclusion, while the ESP32 temperature sensor prototype has not been deployed in a real-world data centre, the findings from similar studies provide valuable insights. By leveraging the results obtained from these studies, I believe that it is reasonable to expect that the ESP32 temperature sensor would exhibit similar effectiveness in temperature monitoring.

6 DISCUSSION

The purpose of this thesis was to design and implement an ESP32 temperature sensor prototype that uses AWS cloud, a Flutter app that reads data from the cloud and a document that guides the reader through the steps needed to setup the sensor and cloud environment. Furthermore, the thesis also hoped to provide some insight into IoT security practices and the possible use cases for the temperature sensor.

The result of this thesis was an IoT temperature sensor that publishes its readings to AWS cloud, a Flutter app that reads the temperature data from the cloud and a document that guides the reader through the steps to set these up for themselves. TLS was used to secure the data when publishing it to the cloud. This thesis also provides a hypothetical scenario where the IoT temperature sensor could be used to measure temperature data in a data centre and provides findings from previous studies that reinforce this belief.

The document produced by this thesis can be used by anyone as a starting point in their journey to produce IoT sensors. The document in question also provides insight into AWS services such as AWS Amplify and AWS API Gateway.

This thesis uses reliable sources as much as possible. Some of these sources are official specifications, AWS documentations and research papers. For the few times that blog posts are used, the posts in question are from reliable sources.

Future research could deploy the temperature sensor into a real-life data centre and see if the hypothetical scenario devised in this thesis holds any real weight. Future development could implement real-time alerts for temperature deviations.

REFERENCES

Amazon Web Services 2023a. Amplify Studio. Accessed on 31 August 2023 <https://docs.amplify.aws/console/>

Amazon Web Services 2023b. AWS IoT Core Features. Accessed on 31 August 2023 <https://aws.amazon.com/iot-core/features/>

Amazon Web Services 2023c. AWS IoT Device Shadow service. Accessed on 31 August 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html>

Amazon Web Services 2023d. AWS IoT rule actions. Accessed on 1 November 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/iot-rule-actions.html>

Amazon Web Services 2023e. Best practices for querying and scanning data. Accessed on 21 November 2023 <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-scan.html>

Amazon Web Services 2023f. Control access to a REST API using Amazon Cognito user pools as authorizer. Accessed on 10 November 2023 <https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-integrate-with-cognito.html>

Amazon Web Services 2023g. Device provisioning MQTT API. Accessed on 8 November 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/fleet-provision-api.html>

Amazon Web Services 2023h. Figma to code (React). Accessed on 31 August 2023 <https://docs.amplify.aws/console/uibuilder/figmatocode/>

Amazon Web Services 2023i. MQTT. Accessed on 1 November 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html>

Amazon Web Services 2023j. MQTT message payload. Accessed on 31 August 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/topicdata.html>

Amazon Web Services 2023k. Pre-provisioning hooks. Accessed on 31 August 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/pre-provisioning-hook.html>

Amazon Web Services 2023l. Provisioning devices that don't have device certificates using fleet provisioning. Accessed on 31 August 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/provision-wo-cert.html>

Amazon Web Services 2023m. Rules for AWS IoT. Accessed on 31 August 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html>

Amazon Web Services 2023n. Transport security in AWS IoT Core. Accessed on 31 August 2023

<https://docs.aws.amazon.com/iot/latest/developerguide/transport-security.html>

Amazon Web Services 2023o. What is AWS IoT? Accessed on 31 August 2023

<https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html>

Amazon Web Services 2023p. What is Flutter? Accessed on 31 August 2023

<https://aws.amazon.com/what-is/flutter/>

Amazon Web Services 2023q. What is IoT (Internet of Things)? Accessed on

15 May 2023 <https://aws.amazon.com/what-is/iot/>

Amazon Web Services 2023r. X.509 client certificates. Accessed on 1

November 2023 <https://docs.aws.amazon.com/iot/latest/developerguide/x509-client-certs.html>

Bangalore, S., Bhan, A., Miglio, A. D., Sachdeva, P., Sarma, V., Sharma, R. & Srivathsan, B, 2023. Investing in the rising data center economy. McKinsey & Company 17.01.2023. Accessed on 15 January 2024

<https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/investing-in-the-rising-data-center-economy>

Bluetooth SIG 2011. Heart Rate Service. Accessed on 31 August 2023

<https://www.bluetooth.com/specifications/specs/heart-rate-service-1-0/>

Bluetooth SIG 2021. Bluetooth Core Specification Version 5.3. Accessed on 31

August 2023 <https://www.bluetooth.com/specifications/specs/core-specification-5-3/>

Bluetooth SIG 2023. Assigned Numbers. Accessed on 31 August 2023

<https://www.bluetooth.com/specifications/assigned-numbers/>

Cloudflare 2023a. What is mutual TLS (mTLS)? Accessed on 30 August 2023

<https://www.cloudflare.com/en-gb/learning/access-management/what-is-mutual-tls/>

Cloudflare 2023b. What is TLS (Transport Layer Security)? Accessed on 30

August 2023 <https://www.cloudflare.com/en-gb/learning/ssl/transport-layer-security-tls/>

Dart 2023. Dart overview. Accessed on 30 August 2023

<https://dart.dev/overview>

Ebrahimi, K., Jones, G. F. & Fleischer, A. S. 2014. A review of data center cooling technology, operating conditions and corresponding low-grade waste heat recovery opportunities. *Renewable and Sustainable Energy Reviews*, Volume 31, March 2014, 622-638. Accessed on 15 January 2024

<https://doi.org/10.1016/j.rser.2013.12.007>

Espressif Systems 2023a. ESP32. Accessed on 31 August 2023

<https://www.espressif.com/en/products/socs/esp32>

Espressif Systems 2023b. ESP-IDF. Accessed on 31 August 2023
<https://www.espressif.com/en/products/sdks/esp-idf>

George Brown College 2023. The Rise of IoT. Accessed on 15 May 2023
<https://www.gbctechtraining.com/blog/rise-iot>

Green Revolution Cooling 2022. Mishandling Your Data Center's Cooling System Costs More Than You Think. Accessed on 15 January 2024
<https://www.grcooling.com/blog/data-center-cooling-system-mismanagement/>

HiveMQ 2015a. Advanced Authentication Mechanisms - MQTT Security Fundamentals. Accessed on 30 August 2023 <https://www.hivemq.com/blog/mqtt-security-fundamentals-advanced-authentication-mechanisms/>

HiveMQ 2015b. Authentication with Username and Password - MQTT Security Fundamentals. Accessed on 30 August 2023
<https://www.hivemq.com/blog/mqtt-security-fundamentals-authentication-username-password>

HiveMQ 2019. MQTT Topics, Wildcards, & Best Practices – MQTT Essentials: Part 5. Accessed on 27 March 2024 <https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/>

HiveMQ 2023a. Introducing the MQTT Protocol – MQTT Essentials: Part 1. Accessed on 30 August 2023 <https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/>

HiveMQ 2023b. MQTT Client, MQTT Broker, and MQTT Server Connection Establishment Explained – MQTT Essentials: Part 3. Accessed on 30 August 2023 <https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/>

ISO/IEC 20922:2016. Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1. Edition 1. International Organization for Standardization.

MQTT 2022. Who invented MQTT? Accessed on 30 August 2023
<https://mqtt.org/faq/>

Oracle 2023. What is IoT? Accessed on 15 May 2023
<https://www.oracle.com/internet-of-things/what-is-iot/>

Polonelli, T., Brunelli, D., Bartolini, A. & Benini, L. 2019. A LoRaWAN Wireless Sensor Network for Data Center Temperature Monitoring. Application in Electronics Pervading Industry, Environment and Society, 2018, 169-177. Accessed on 16 January 2024 https://doi.org/10.1007/978-3-030-11973-7_20

Rodriguez, M. G., Ortiz Uriarte, L. E., Jia, Y., Yoshii, K., Ross, R. & Beckman, P. H. 2011. Wireless sensor network for data-center environmental monitoring. Fifth International Conference on Sensing Technology, 2011. Accessed on 29 January 2024 <https://doi.org/10.1109/ICSensT.2011.6137036>

Rozite, V., Bertoli, E. & Reidenbach, B. 2023. Data Centres and Data Transmission Networks. International Energy Agency 11.07.2023. Accessed on 15 January 2024 <https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks>

Scherer, T., Lombriser, C., Schott, W., Truong, H. L. & Weiss, B. 2012. Wireless Sensor Network for Continuous Temperature Monitoring in Air-Cooled Data Centers. IBM Research Report, RZ 3807, 2011. Accessed on 31 January 2024 <https://dominoweb.draco.res.ibm.com/11d5db9b654e35ad852578c4004d6ec1.html>

Woolley, M. 2023. The Bluetooth® Low Energy Primer. Kirkland: Bluetooth SIG, Inc. Accessed on 31 August 2023 <https://www.bluetooth.com/bluetooth-resources/the-bluetooth-low-energy-primer/>

APPENDICES

Appendix 1. Guide to setting up the ESP32 temperature sensor

Appendix 1 1(37). Guide to setting up the ESP32 temperature sensor



Guide to setting up the ESP32 temperature sensor

Court Onni

2024

Appendix 1 2(37). Guide to setting up the ESP32 temperature sensor

CONTENTS

1	INTRODUCTION	3
2	PREREQUISITES.....	4
3	SETUP THE TEMPERATURE SENSOR'S CLOUD SIDE	6
3.1.1	Setup fleet provisioning by claim	6
3.1.2	Setup AWS IoT Rule	14
4	SETUP THE ESP32 DEVICE	18
5	SETUP THE FLUTTER APP	23
5.1.1	Setup AWS Amplify	29
5.1.2	Deploy the API.....	32
6	TESTING THE PROGRAMS	33
6.1.1	Testing the Flutter app.....	33
6.1.2	Testing the temperature sensor.....	34
	REFERENCES	36

Appendix 1 3(37). Guide to setting up the ESP32 temperature sensor

1 INTRODUCTION

This guide was written to accompany the thesis *Creating an ESP32 temperature sensor with AWS IoT Core and Flutter*. This guide goes through the steps to setup the ESP32 temperature sensor, and its accompanying Flutter app. All source code is publicly available on GitHub and under the MIT licence.

Appendix 1 4(37). Guide to setting up the ESP32 temperature sensor

2 PREREQUISITES

Here is a list of prerequisites, needed to follow along this guide:

- Any ESP32 device
- Android or iOS mobile phone
- AWS account
- Flutter
- Amplify CLI
- ESP-IDF

The temperature sensor program has only been tested with an esp32c3 device. An emulator cannot be used as a substitute for a mobile phone as Bluetooth is not supported by an emulator (Advanced emulator usage 2023). As the example programs uses AWS Cloud, you will naturally need an AWS account. All AWS services used have free tiers. While optional, it is recommended to install either Android Studio or Visual Studio Code with Flutter. Amplify CLI is needed to setup AWS Amplify. For instructions on how to set it up, see the following link: (<https://docs.amplify.aws/start/q/integration/flutter/>).

Follow the official instructions to install ESP-IDF. You can use any of the official methods; just make sure that the IDF_PATH environment variable points to your ESP-IDF folder. The installers should do this automatically for you. Test that this is the case by running one of the following commands:

```
PowerShell: '$env:IDF_PATH'
```

```
Cmd.exe: 'echo %IDF_PATH%'
```

If this variable is set, it will print out its value, which should be the location of your ESP-IDF installation. By default, this is 'C:\Users\<<username>\esp\esp-idf'. If you do not get the desired output or no output, the env variable is not set. See the following guide on how to set it up: (https://docs.espressif.com/projects/esp-idf/en/v3.3.1/get-started/add-idf_path-to-profile.html).

Appendix 1 5(37). Guide to setting up the ESP32 temperature sensor

3 SETUP THE TEMPERATURE SENSOR'S CLOUD SIDE

Let us start by setting-up the AWS cloud for our temperature sensor. We'll first setup the fleet provisioning by claim, this will give us the ability to provision devices with claim certificates, we will later flash these certificates onto our device. After setting-up fleet provisioning, we will create an AWS IoT Rule which will insert temperature data to a DynamoDB table.

3.1.1 Setup fleet provisioning by claim

To use fleet provisioning, we need to create a fleet provisioning template. Fleet provisioning templates are JSON documents that describes the resources that devices use to interact with AWS IoT.

- Start by signing into your AWS account. Once you've signed in, navigate to AWS IoT Console by selecting the IoT Core service from services.
- From there navigate to Connect -> Connect many devices -> Create provisioning template.
- Next select Provisioning device with claim certificate as the Provisioning scenario, proceed further with the Next button.
- Select Active as the Provisioning template status.
- Set Provisioning template name as 'temperature-sensor-template', note this name down, we will use it later.
- (Optional) Set a description.
- For provisioning role select Create new role, give the role a name of 'temperature-sensor-claim-role'.
- Click the View button which will open the role in the IAM console.
- Select Add permissions and Attach policies, search for 'AWSIoTThingsRegistration', select it and attach it with Add permissions as show in Figure 1.

Appendix 1 6(37). Guide to setting up the ESP32 temperature sensor

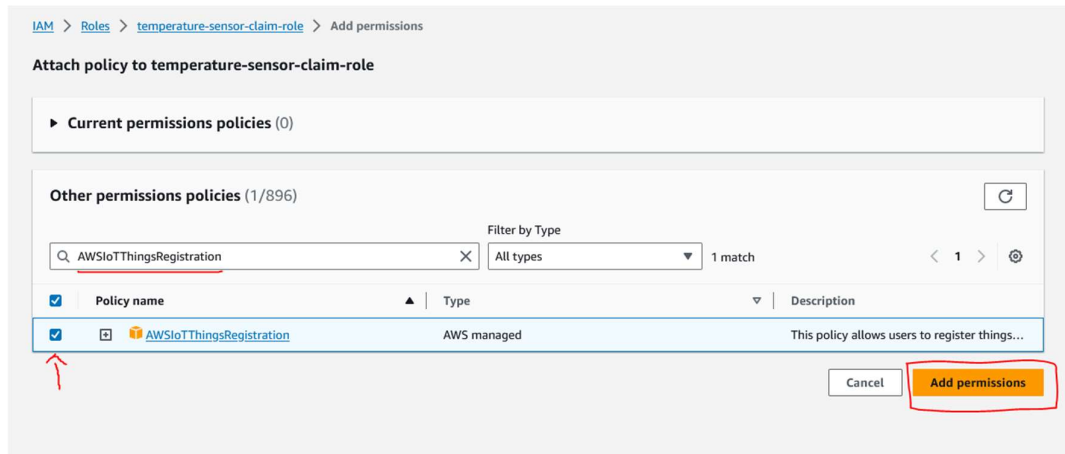


Figure 1. Attach policy to role

- Once back at the template creation page and make sure the previously created role is selected (Figure 2). This policy will provide the template the permissions needed to register a thing.

Appendix 1 7(37). Guide to setting up the ESP32 temperature sensor

Provisioning template properties [Info](#)

Provisioning template status
The provisioning template status determines whether the template can be used to provision a new device. Only active templates can provision devices.

Inactive
Inactive templates can't provision any devices that are configured to use it. You can create an inactive template to prevent devices from being provisioned until you're ready.

Active
An active template can provision the devices that are configured to use it.

Provisioning template name

The name can have up to 36 characters and must not contain spaces. Valid characters: A-Z, a-z, 0-9, and _ (underscore) and - (hyphen).

Description - optional

500 character remaining

Provisioning role
The provisioning role uses an IAM role that authorizes AWS IoT to access resources on your behalf.

Attach managed policy to IAM role

► Tags - optional

Claim certificate policy [Info](#)
The claim certificate requires a policy that authorizes it to connect to AWS IoT and perform the actions that provision the device. This policy doesn't apply to the device certificate that will be provisioned. You'll configure the policies for the provisioned device certificate later.

Claim certificate provisioning policy

Figure 2. Select policy for provisioning role

- For Claim certificate provisioning policy select Create IoT policy.
- Give the policy the name of 'temperature-sensor-claim-policy'.
- For Policy document select JSON and paste in the following json.

Appendix 1 8(37). Guide to setting up the ESP32 temperature sensor

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:<region>:<ACCOUNT-ID>:topic/$aws/certificates/create/*",
        "arn:aws:iot:<region>:<ACCOUNT-ID>:topic/$aws/provisioning-templates/temperature-sensor-template/provision/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": [
        "arn:aws:iot:<region>:<ACCOUNT-ID>:topicfilter/$aws/certificates/create/*",
        "arn:aws:iot:<region>:<ACCOUNT-ID>:topicfilter/$aws/provisioning-templates/temperature-sensor-template/provision/*"
      ]
    }
  ]
}

```

Appendix 1 9(37). Guide to setting up the ESP32 temperature sensor

- Next make the following modifications to the json.

Change all <region>'s to your region. This can be found out by selecting your regions name in the top right corner. For example, for region Europe (Stockholm) replace <region> with eu-north-1 (see Figure 3).

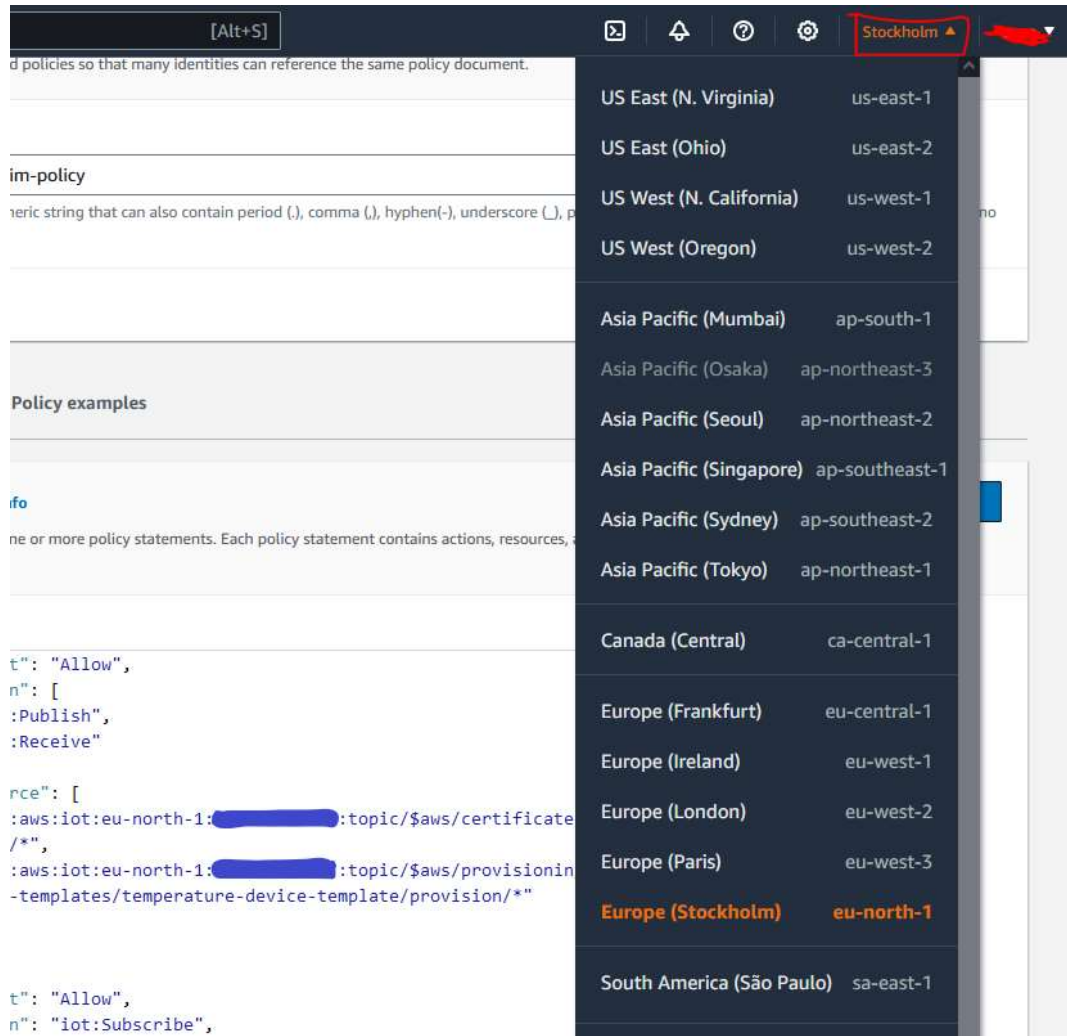


Figure 3. Finding AWS region

Change all <ACCOUNT-ID>'s with your accounts id. Your account id can be found out by selecting your accounts name in the top right corner. For example, for an account id of 4123-0432-1234, replace <ACCOUNT-ID> with 412304321234. See the following json for an example of these modifications.

Once all modifications are done, select Create.

Appendix 1 10(37). Guide to setting up the ESP32 temperature sensor

```
{
  "arn:aws:iot:eu-north-1:412304321234:topic/$aws/certificates/create/*",
  "arn:aws:iot:eu-north-1:412304321234:topic/$aws/provisioning-templates/tem-
perature-sensor-template/provision/*"
}
```

- Once back at the template creation page, make sure that the policy is selected. This policy will restrict the access of a device that is registering itself to only the essentials that it needs. Once it has registered itself, it will get access to another set of certificates that it will use to publish data with.
- Let us now create the claim certificates. Open a new browser tab and navigate to IoT Core -> Security -> Certificates and select Add certificate -> Create certificate.
- Select Auto-generate new certificates (recommended) and set Certificate status to Active, create the certificates by selecting Create.
- Download all files with the exception being the Amazon Root CA 3 file. These files will be used by our sensor device to make the initial connection to AWS IoT Core. Make sure to download these files here as once you click Continue you won't be able to download them again.
- Back in provisioning template creation page, for Claim certificates, select the certificate we just created (If you cannot see it in the list, refresh the list). Once done select Next.
- Select Don't use a pre-provisioning action under Pre-provisioning action. AWS recommends using a pre-provisioning action as they provide extra security, but in our demo, we won't use one.
- Leave everything else empty and proceed further with Next.
- Next, we will create the policy that a registered thing will have. This policy will define the actions and topics that the thing has access to. Select Create policy, give the policy the name of `temperature-sensor-policy`.
- For Policy document select JSON and paste in the following json.

Appendix 1 11(37). Guide to setting up the ESP32 temperature sensor

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:<region>:<ACCOUNT-ID>:client/${iot:Connection.Thing.ThingName}"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": "arn:aws:iot:<region>:<ACCOUNT-ID>:topic/device/${iot:Connection.Thing.ThingName}/temperature/data"
    }
  ]
}

```

- Like previously, replace the <region>'s and <ACCOUNT-ID>'s.

"arn:aws:iot:<region>:<ACCOUNT-ID>:client/\${iot:Connection.Thing.ThingName}" will allow only a registered thing to connect to AWS IoT Core, '\${iot:Connection.Thing.ThingName}' will be replaced by AWS with the things name on creation.

"arn:aws:iot:<region>:<ACCOUNT-ID>:topic/device/\${iot:Connection.Thing.ThingName}/temperature/data" will allow a registered thing to publish messages to their own topics. For example, a thing with a name of thing123, can publish messages to the following topic 'device/thing123/temperature/data'.

Appendix 1 12(37). Guide to setting up the ESP32 temperature sensor

- Select Create, next select the policy that you just created and select Next (If you cannot see the policy refresh the list). Review that all fields are correct and select Create template.
- Once the template is created, select it again, scroll down and select Edit JSON.
- Replace the contents with the following json. With this new document AWS will set the name of the thing to be registered to the ThingName parameter provided by the device.

```
{
  "Parameters": {
    "ThingName": {
      "Type": "String"
    },
    "AWS::IoT::Certificate::Id": {
      "Type": "String"
    }
  },
  "Resources": {
    "policy_temperature-sensor-policy": {
      "Type": "AWS::IoT::Policy",
      "Properties": {
        "PolicyName": "temperature-sensor-policy"
      }
    },
    "certificate": {
      "Type": "AWS::IoT::Certificate",
      "Properties": {
        "CertificateId": {
          "Ref": "AWS::IoT::Certificate::Id"
        }
      },
      "Status": "Active"
    }
  }
}
```

Appendix 1 13(37). Guide to setting up the ESP32 temperature sensor

```
}  
},  
"thing": {  
  "Type": "AWS::IoT::Thing",  
  "OverrideSettings": {  
    "AttributePayload": "MERGE",  
    "ThingGroups": "DO_NOTHING",  
    "ThingTypeName": "REPLACE"  
  },  
  "Properties": {  
    "AttributePayload": {},  
    "ThingGroups": [],  
    "ThingName": {  
      "Ref": "ThingName"  
    }  
  }  
}  
}  
}
```

- Save the changes with Save as new version. And make sure that the newly created version (most likely named Version 2) is active as show in Figure 4. With this we have created a fleet provisioning template for our project.

Appendix 1 14(37). Guide to setting up the ESP32 temperature sensor

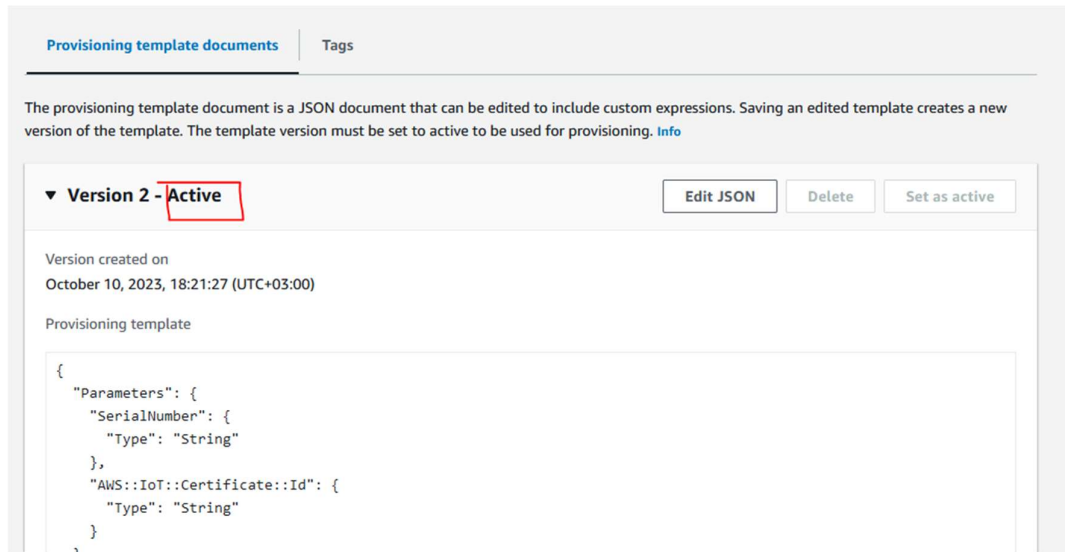


Figure 4. Find active template

3.1.2 Setup AWS IoT Rule

Next, we will create an AWS IoT Rule that will allow us to insert temperature data into a DynamoDB table. AWS IoT Rule provides two actions that can be used to insert data into a DynamoDB table. We will use the DynamoDBv2 action rather than the DynamoDB action, as the former will insert the data into its own columns. Before we create the rule, let's first create a DynamoDB table that will store our temperature data.

Next, we will create a DynamoDB table with 3 columns. These columns are: thing-name, timestamp and temperature. With thingname being the primary key and timestamp being the sort key, both will be of type String.

The following will describe the steps to create such a table:

- First, navigate to DynamoDB by selecting it from AWS services
- Select Tables and Create table
- Give the table the name 'temperature_data'
- Set the Partition key as 'thingname' with the type of String

Appendix 1 15(37). Guide to setting up the ESP32 temperature sensor

- Set the Sort key as 'timestamp' with the type of String
- For Table settings, leave the default selected
- Select Create table to finish creating the table

Unlike a relational database such as SQL, DynamoDB does not require separately defining a column for each item. Meaning that if we want to add temperature data to our table, we can just do so.

Now that we have created a table to store our temperature data, let's go ahead and create an AWS IoT Rule that will insert data into it. The following will describe the steps to create an AWS IoT Rule that will insert temperature data into our table:

- Navigate to the AWS IoT Console, once there, navigate to Message routing -> Rules. Start the rule creation process by selecting Create rule.
- Give the rule the name 'temperature_data_ddb' and an optional description. Proceed further with Next.
- Select SQL version '2016-03-23' for the SQL version. In the SQL statement box, enter the following:

```
SELECT temperature, cast(timestamp() as String) as timestamp, cast(topic(2) as String) as thingname, FROM 'device+/temperature/data'
```

- Proceed further with the Next button.
- Under Rule actions, set Action 1 as DynamoDBv2, and for Table name, select the table we created previously (temperature_data).
- For the IAM role, choose Create new role. Give the role the name 'temperature_data_ddb_role'. This IAM policy will give our rule the required permissions to insert data into our DynamoDB table. Make sure the policy is selected.
- Finish the rule creation with Next and Create on the next page.

The SQL statement previously defined will do the following:

Appendix 1 16(37). Guide to setting up the ESP32 temperature sensor

- Listen to MQTT messages with the topic of 'device/+temperature/data'. For example, the topics 'device/thing1/temperature/data' and 'device/thing2/temperature/data' will match this topic filter.
- 'SELECT temperature' will extract the 'temperature' field from the published message payload.
- 'cast(timestamp() as String) as timestamp' will get the current timestamp with the timestamp() and convert it to the type of String with the cast() function. Afterwards the timestamp value will be given the alias of 'timestamp'. Both functions are provided by AWS IoT Core.
- 'cast(topic(2) as String) as thingname' will extract the second topic segment from the topic and cast it to a type of String with the alias of 'thingname'. For example, with a topic of 'device/thing1/temperature/data' this would equal to 'thing1'. With this function we can extract the name of the thing that published this message.

Let's test whether this rule works. Start by navigating to the AWS IoT Console -> Test -> MQTT test client. Subscribe to the 'device/+temperature/data' topic. Next, publish a message to the 'device/test123/temperature/data' topic with a message payload of:

```
{  
  "temperature": 30  
}
```

Next, open the DynamoDB table that we created before in a separate tab by navigating to DynamoDB console -> Tables and selecting the 'temperature_data' table. Select "Explore table items" to see the table's items. You should now see an entry in this table's items, if not, make sure that you have done the previous steps correctly. You can edit the created AWS Rule by navigating to the Rules page, and selecting the rule that we just created, and selecting the "Edit" button.

Appendix 1 17(37). Guide to setting up the ESP32 temperature sensor

4 SETUP THE ESP32 DEVICE

Next, we will set up our ESP32 device. This part will consist of configuring, building, and flashing our program and the claim certificates to an ESP32 development board.

Start by cloning the repository with git. This can be done with the following command:

- git clone <https://github.com/ocour/esp32-temperature-sensor.git>
- Navigate into the cloned directory with the command 'cd esp32-temperature-sensor'.
- Connect the ESP32 device to your computer with a USB cable.
- Run one of the following commands, these commands will add ESP-IDF tools to the PATH.

Powershell: '\$env:IDF_PATH/export.ps1'

Cmd.exe: '%IDF_PATH%/export.bat'

- Next, set the target board with the 'idf.py set-target <board>' command. Replace <board> with your board. All supported boards can be found by executing the next command: 'idf.py set-target --help'.
- Run: 'idf.py menuconfig', which will open a terminal-based configuration menu. Navigation is done using the 'h', 'j', 'k' and 'l' keys.
- Navigate to Partition Table and set it to 'custom partition table CSV'.
- Navigate to CUSTOM CONFIG and,
- Set 'ble device name' to anything, for example 'temperature sensor'.
- Set 'broker url' to your AWS MQTT endpoint. The endpoint can be found by navigating to AWS IoT Console in your browser and selecting Settings. The Endpoint field will be under Device data endpoints. Copy and paste it with 'CTRL + SHIFT+ v'.

Appendix 1 18(37). Guide to setting up the ESP32 temperature sensor

- Set port to '8883'.
- Set 'aws template name' to fleet provisioning by claim template name that we created previously, which is 'temperature-sensor-template'.
- Set 'Claim ClientId' to anything, note that only one device at a time can connect with the same client ID. For example, 'claimer1'.
- (Optional) Change 'Wifi scan mode auth threshold'.
- Next navigate to Component config -> Bluetooth and enable (tick) Bluetooth. Set Host to 'Nimble – BLE only'.
- Navigate to Component config -> Wi-Fi and disable (untick) 'Wifi NVS flash'. This will prevent ESP-IDF from saving Wi-Fi connection data to NVS, we don't need this as we do this ourselves already.
- Save and exit with S & Q.
- Build the program with the 'idf.py build' command. Make sure there are no errors.
- Build and flash the partition table with 'idf.py -p <COM> partition-table partition-table-flash'. Change '<COM>' to your device's COM port.
- On Windows, the COM port can be found by first connecting your ESP32 device to your computer and searching for Device Manager in the search bar. Opening it and navigating to Ports (COM & LPT) (Figure 5). If you see more than one COM port, disconnect the ESP32 device, note down the current COM ports, connect your ESP32 device again, and see which one was added.

Appendix 1 19(37). Guide to setting up the ESP32 temperature sensor

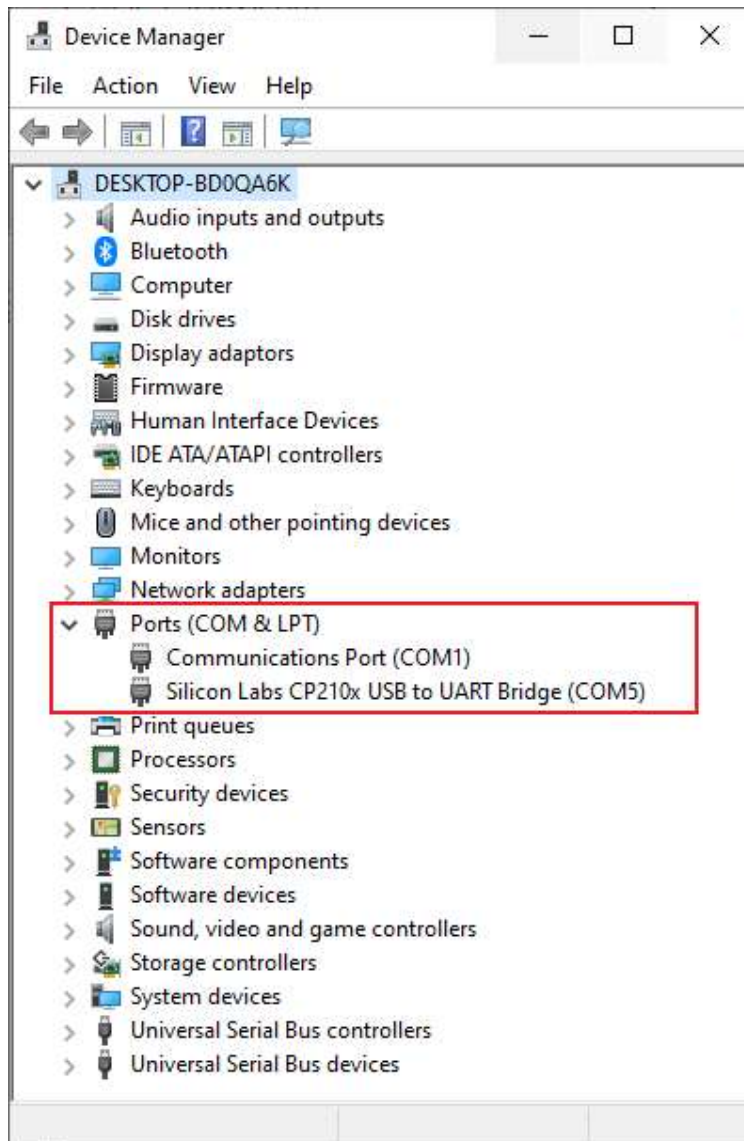
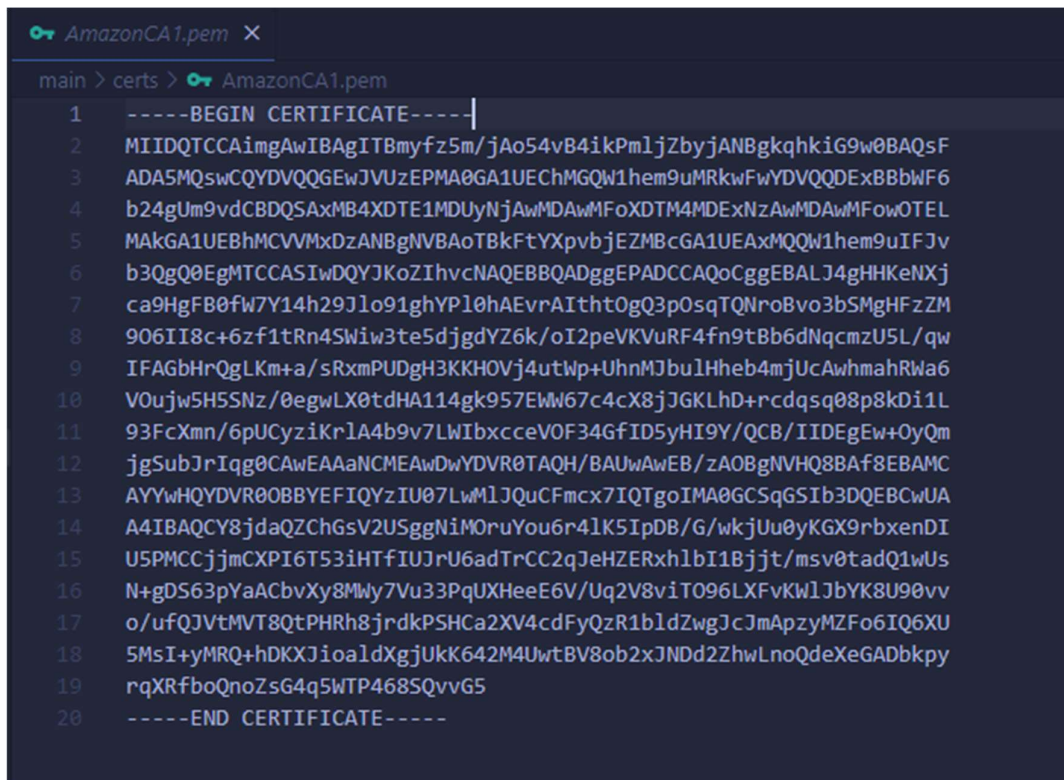


Figure 5. Finding the COM port

- Next, navigate to the 'esp32-temperature-sensor/main/certs' directory and create 3 files inside it.
 - AmazonCA1.pem
 - client.pem.crt
 - client.pem.key
- Locate the X.509 certificate files that you downloaded in Chapter 3.1.1 and copy their contents to these files.

Appendix 1 20(37). Guide to setting up the ESP32 temperature sensor

- AmazonRootCA1.pem -> AmazonCA1.pem
 - xxxxxxx-certificate.pem.crt -> client.pem.crt
 - xxxxxxx-private.pem.key -> client.pem.key
- Note: Make sure to copy the entire file's content into its respective file. This includes the '-----BEGIN CERTIFICATE-----' and '-----END CERTIFICATE-----'.
 - Figure 6 shows what the AmazonCA1.pem file should look like.



```

AmazonCA1.pem X
main > certs > AmazonCA1.pem
1  -----BEGIN CERTIFICATE-----|
2  MIIDQTCCAimgAwIBAgITBmyfz5m/jAo54vB4ikPmljZbyjANBgkqhkiG9w0BAQsF
3  ADA5MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6
4  b24gUm9vdCBDQSAxMB4XDTE1MDUyNjAwMDAwMFoXDTM4MDEwNzAwMDAwMFowOTEL
5  MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvcjEzMDEwNAkGA1UEAxMQQW1hem9uIFJv
6  b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
7  ca9HgFB0fw7Y14h29Jl091ghYp10hAEvrAItht0gQ3p0sqTQNroBvo3bSMgHFzZM
8  906II8c+6zf1tRn4Swiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/qw
9  IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+UhnMJbulHheb4mjUcAwhmahRWa6
10 V0ujw5H5SNz/0egwLX0tdHA114gk957EwW67c4cX8jJGKLhD+rcdqsq08p8kD11L
11 93FcXmn/6pUCyzIKr1A4b9v7LWIbxcceV0F34GfID5yHI9Y/QCB/IIDEgEw+OyQm
12 jgSubJrIqg0CAwEAaAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zA0BgNVHQ8BAf8EBAMC
13 AYYwHQYDVr00BBYEFIQYzIU07LwMlJQuCFmcx7IQTgoIMA0GCSqGSIb3DQEBcWUA
14 A4IBAQC8jdaQZChGsV2USggNiM0ruYou6r4lK5IpDB/G/wkjUu0yKGX9rbxenDI
15 U5PMCCjJmCXPI6T53iHTfIUJrU6adTrCC2qJeHZERxh1bI1BjJt/msv0tadQ1wUs
16 N+gDS63pYaACbvXy8Mwy7Vu33PqUXHeeE6V/Uq2V8viT096LXFvKWlJbYK8U90vv
17 o/ufQJvtMVT8QtPHRh8jrdkPSHca2XV4cdFyQzR1bldZwgJcJmApzyMZFo6IQ6XU
18 5MsI+yMRQ+hDKXJioaldXgJukK642M4UwtBV8ob2xJNDd2ZhwLnoQdeXeGADbkpy
19 rQXRfboQnoZsG4q5WTP468SQvvG5
20 -----END CERTIFICATE-----

```

Figure 6. AmazonCA1.pem files contents

- Generate NVS binary file from a CSV file with one of the following commands.

Powershell: 'python \$env:IDF_PATH/components/nvs_flash/nvs_partition_generator/nvs_partition_gen.py generate "nvs.csv" "nvs.bin" 0x10000'

Cmd.exe: 'python %IDF_PATH%/components/nvs_flash/nvs_partition_generator/nvs_partition_gen.py generate "nvs.csv" "nvs.bin" 0x10000'

Appendix 1 21(37). Guide to setting up the ESP32 temperature sensor

- Flash this NVS binary file to the board. Change the '<COM>' to your device's COM port.

Powershell: 'python \$env:IDF_PATH/components/partition_table/part-tool.py -p <COM> write_partition --partition-name=nvs --input "nvs.bin"'

Cmd.exe: 'python %IDF_PATH%/components/partition_table/parttool.py -p <COM> write_partition --partition-name=nvs --input "nvs.bin"'

- Lastly, build, flash, and monitor the program on the device with the following command: 'idf.py -p <COM> build flash monitor'.

If the program was successfully flashed, you should see 'I (XXX) NimBLE: Started advertising.' as the last message printed to the console.

Appendix 1 22(37). Guide to setting up the ESP32 temperature sensor

5 SETUP THE FLUTTER APP

Next, we will create the API that our Flutter app will use to fetch temperature data. This API will have two endpoints, which are GET methods.

The first endpoint, '/temperature/{thingname}' will fetch the temperature data of the specific device, the name of the device is specified in the URL. For example, the following URL, '/temperature/thing1' will fetch the temperature data of the device 'thing1'.

We will secure this API with an Amazon Cognito user pool. To invoke the API, the user must pass an identity token inside the 'Authorization' HTTP header. To get this token, the user must be signed in. (Amazon Web Services 2023a).

Let's begin by creating the IAM role that our API will use. This role will provide the API with the permissions that it needs to read data from our DynamoDB table.

The following describes the steps to create the IAM role:

- Start by navigating to the Identity and Access management (IAM) console -> Access management -> Roles and selecting Create role.
- Set Trusted entity type as AWS service. Under Use case, set Service or use case as API Gateway. Select Next, and on the next page, select Next again.
- Give the role the name 'temperature-sensor-table-role'. Create the role with Create role.
- Next, locate the role (under Roles) and select it by clicking its name.
- Select Add permissions and Attach policies.
- Type 'AmazonDynamoDBReadOnlyAccess' into the search field and select (tick) it. Attach the policy with Add permissions.
- Copy the ARN of this IAM role and save it to a notepad, we will use it later. See Figure 7.

Appendix 1 23(37). Guide to setting up the ESP32 temperature sensor

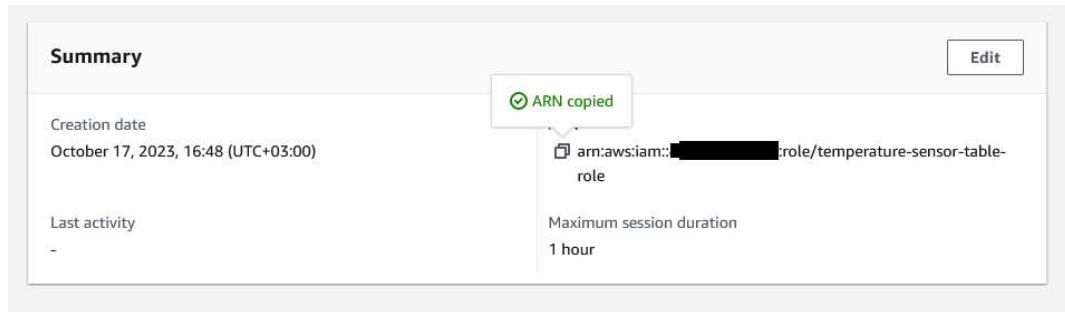


Figure 7. Locate and copy IAM Role's ARN

The 'AmazonDynamoDBReadOnlyAccess' policy will provide read-only access to any DynamoDB table. This policy will allow us to use the 'Query' and 'Scan' operations. 'Query' will be used to fetch the data of a specific device, 'Scan' will be used to fetch all devices.

Now that we have created the IAM role for our API, let's create the API.

- Start by navigating to the API Gateway console. Select APIs and Create API.
- Choose the Rest API by clicking its Build button.
- Select New API and give it the name 'temperature-sensor-api'. For API endpoint type, select Regional. Proceed further with Create API.
- Next, we will create the methods. Start by selecting Create resource and setting its name (Resource name) as 'temperature'. Proceed further with Create resource.
- Select '/temperature' and create another resource with Create resource. Set its name as '{thingName}'. Continue with Create resource.
- Next, select '/{thingName}' and click Create method. This method will fetch the temperature records of a device.
- Set Method type as GET.
- Set Integration type as AWS service.
- Set AWS Region to your region.

Appendix 1 24(37). Guide to setting up the ESP32 temperature sensor

- Set AWS Service as DynamoDB. Leave AWS subdomain empty.
- Select POST for the HTTP method.
- Set Action name as 'Query'.
- For Execution role, set the IAM policy ARN that we created previously. See Figure 8 for an example of this method.
- Create the method with Create method.

Appendix 1 25(37). Guide to setting up the ESP32 temperature sensor

Create method

Method details

Method type
GET

Integration type

Lambda function
Integrate your API with a Lambda function.

HTTP
Integrate with an existing HTTP endpoint.

Mock
Generate a response based on API Gateway mappings and transformations.

AWS service
Integrate with an AWS Service.

VPC link
Integrate with a resource that isn't accessible over the public internet.

AWS Region
eu-north-1

AWS service
DynamoDB

AWS subdomain

HTTP method
POST

Action type

Use action name

Use path override

Action name - *optional*
Query

Execution role
arn:aws:iam::[redacted]:role/temperature-sensor-table-role

Credential cache
Do not add caller credentials to cache key

Default timeout
The default timeout is 29 seconds.

Cancel **Create method**

Figure 8. Create DynamoDB query method

Appendix 1 26(37). Guide to setting up the ESP32 temperature sensor

- Next, proceed to Integration request (see Figure 9.) and select Create template. Make sure that the GET method that we just created is selected.
- Set Content type to 'application/json', leave Generate template empty.
- For the Template body, paste the following json.

```
{
  "TableName": "temperature_data",
  "PrimaryKey": "thingname",
  "KeyConditionExpression": "thingname = :val",
  "ExpressionAttributeValues": {
    ":val": {
      "S": "$input.params('thingName')"
    }
  }
}
```

“TableName” will correspond to our temperature data DynamoDB table name. “PrimaryKey” will be this table’s Partition key, in our case, it is ‘thingname’. With “KeyConditionExpression” we will retrieve only entries that match the expression. In this case, only entries that match the device name that we supply as a parameter. “\$input.params(‘thingName’)” will fetch the thingName parameter from the http request and assign it to the “:val” variable. (Amazon Web Services 2023b).

- Finish creating this method with Create template.
- Next, let’s test that this method works. You should already have some data inside the DynamoDB table from the testing that we did in Chapter 3.1.2. Remember that we tested that our AWS IoT rule works. Let’s fetch the temperature data that we published in that chapter.

Appendix 1 27(37). Guide to setting up the ESP32 temperature sensor

- Navigate to the method's Test tab (located next to Integration request). To do so, set thingName to 'test123', this will fetch the temperature data that the device with a name of 'test123' has published.

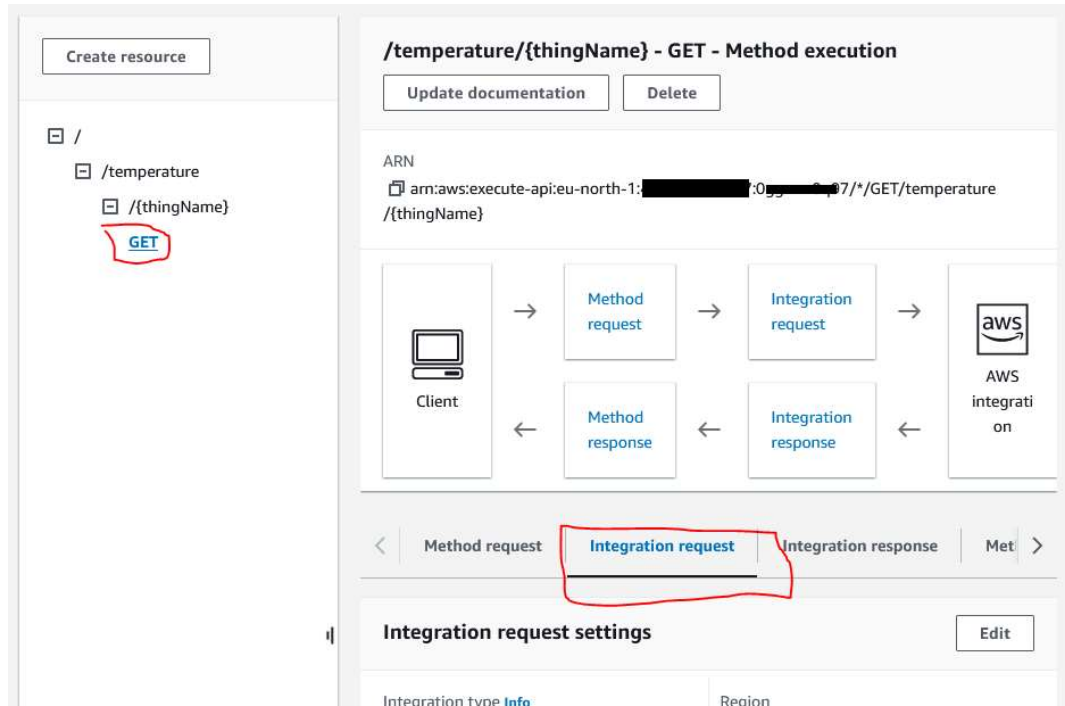


Figure 9. Locate Integration request

- You should get a response with some temperature data inside the "Items" array and a status of 200 to indicate success.

If you did not get a status of 200 but instead an error status, you have done something wrong. Go back and make sure that you did everything as instructed

- Once the method works, navigate back out of the Test tab.
- Select '/' on the left inside resources and create another resource with Create resource.
- Give this resource the name of 'things' and select Create resource.
- Create a method for this resource with Create method.

Appendix 1 28(37). Guide to setting up the ESP32 temperature sensor

- Set the Method type to GET, and just like previously, select AWS service, region, DynamoDB and POST.
- For Action name type 'Scan'.
- Set Execution role as the IAM policy ARN that we used previously.
- Proceed with Create method.
- Just like before, navigate to Integration request and select Create template.
- Set Content type to 'application/json' and Template body to the following json.

```
{  
  "TableName": "temperature_data",  
  "ProjectionExpression": "thingname"  
}
```

"ProjectionExpression" will allow us to select which columns to return. With a value of "thingname" our response will consist only of the "thingname" columns, no temperature data will be returned, as we don't need it (Amazon Web Services 2023c).

- Test this method as well. You should get a json response and a status of 200 to indicate success.

Before we deploy our API to the internet so that it can be used by our app, let's secure our API with an Amazon Cognito user pool. Rather than manually creating an Amazon Cognito user pool, we will leverage AWS Amplify. AWS Amplify Auth lets us quickly and securely set up authentication for our app. AWS Amplify will also provide us with libraries that we can use to communicate with Amazon Cognito and other AWS Cloud services. We will use a Flutter library called 'amplify_auth_cognito' provided by AWS Amplify to interface with Amazon Cognito.

Appendix 1 29(37). Guide to setting up the ESP32 temperature sensor

5.1.1 Setup AWS Amplify

In this subchapter, we will set up AWS Amplify for our Flutter app. Of the features provided by AWS Amplify, we will use Authentication.

Before we go any further, let's make sure that Amplify CLI is installed. Do so with the following command: `amplify -v`. If Amplify CLI is installed, you should see the version of your Amplify CLI program. If you get an error, install Amplify CLI (see 1. PREREQUISITES).

Once Amplify CLI is installed, we can proceed to setup AWS Amplify Auth.

- Start by cloning the Flutter app with: `git clone https://github.com/ocour/flutter_temperature_app`
- Next, navigate into the cloned directory and run: `'flutter pub get'`
This will fetch the app's dependencies from pub.dev.
- Initialise Amplify with: `'amplify init'`, proceed through by pressing enter when prompted, this will leave all values as their defaults.
- Run: `amplify push`, this will create our Cognito user pool. Enter 'yes' when prompted. This will push the local Amplify project to AWS cloud.
- Next, let's add a user to the Cognito user pool. Do this by navigating to AWS Amplify console in your browser.
- Select the app just created in All apps -> temperatureapp.
- Under Backend environments tab, select Action and View details.
- Note down the User pool name, it will be something like `'temperatureappb12345ab_userpool_b12345ab-dev'`.
- Under the Users tab create a new user with Create user.
- Give the user a name, email address and a password. Do not give them a phone number.

Appendix 1 30(37). Guide to setting up the ESP32 temperature sensor

Make sure to tick the box under Email address - optional that says Mark email address as verified. If this box is not ticked you will not be able to proceed into the app, as the Flutter app does not currently implement email verification.

- Finish with Create user.

Now that we have created the Cognito user pool and added a user, let's make our API use the user pool.

Let's create the authorizer for our API. To do so, follow the following steps:

- Start by navigating back to API Gateway console.
- Select the 'temperature-sensor-api' API.
- Select Authorizers from the menu on the left and proceed to create an authorizer by selecting Create authorizer.
- Give the authorizer a name, such as 'Cognito_authorizer'.
- For Authorizer type select Cognito.
- Set the region of the user pool that we just create and select the user pool.
- For Token source type 'Authorization'. This will be the name of the http header that will contain our identity token. Leave Token validation empty. See Figure 10 for an example authorizer.

Appendix 1 31(37). Guide to setting up the ESP32 temperature sensor

Authorizer details

Authorizer name

Authorizer type [Info](#)
 Choose to authorize your API calls using one of your Lambda functions or a Cognito User Pool.

Lambda
 Cognito

Cognito user pool
 Select the Cognito user pool that will authenticate requests to your API.

Token source
 Enter the header that contains the authorization token.

Token validation - optional
 Enter a regular expression to validate tokens.

Figure 10. Example API authorizer

- Finish creation with Create authorizer.
- Next, we need to attach this authorizer to our APIs methods. To do so, select Resources from the menu and select the first GET method.
- Under the Method request tab select Edit.
- For Authorization, select the authorizer 'Cognito_authorizer'.
- Save changes with Save. Next, same for the other GET method.

Once both GET methods have the authorizer attached to them, proceed to deploying our API.

5.1.2 Deploy the API

Next, we will deploy our API to the internet.

- Start by navigating to API Gateway console -> APIs -> 'temperature-sensor-api'.
- Select Deploy API located in the upper right corner.
- Set Stage to *New stage* and give the stage the name 'API'.

Appendix 1 32(37). Guide to setting up the ESP32 temperature sensor

- Finish deploying with Deploy.
- Next, copy the Invoke URL.

Congratulations you have created and deployed your API. All that's left to do is provide our Flutter app the invoke URL and test that it works.

This is done by creating a 'backend.dart' file inside 'flutter_temperature_app/lib' with the following contents:

```
/// AWS API GATEWAY endpoint without the https start or stage name
const ENDPOINT = "<INVOKE_URL>";
/// AWS API GATEWAY stage name
const STAGE = "<STAGE_NAME>";
```

Set ENDPOINT to the invoke URL of our API. Trim off the 'https://' and '/API' parts. Set STAGE to the stage name of our API, this is 'API'.

After modification the contents should look something like so:

```
/// AWS API GATEWAY endpoint without the https start or stage name
const ENDPOINT = "123456abc.execute-api.eu-north-1.amazonaws.com";
/// AWS API GATEWAY stage name
const STAGE = "API";
```

Save the file and proceed to the next chapter where we test the app.

Appendix 1 33(37). Guide to setting up the ESP32 temperature sensor

6 TESTING THE PROGRAMS

Now that we have set up our AWS cloud resources, Flutter app and ESP32 temperature sensor, let's test that everything works.

6.1.1 Testing the Flutter app

Let us test that our app can successfully sign in and call APIs. To do this, you need to build and run the app. This can be done using Android Studio or Visual Studio Code. Since we are just testing API and login calls, we can use an emulator; later, when testing device registration, we need to use a physical phone since emulators don't support Bluetooth (Advanced emulator usage 2023).

First, open our Flutter app with Android Studio or Visual Studio Code. Next, build and run it using the IDE's controls. Run the app on an emulator or a physical device.

Once the app starts up, you should see a sign-in screen. Go ahead and sign in with the user that we created in previously. Next, you should see a new password screen asking you to set a new password for the account. To proceed, set a new password.

After successfully setting a new password, you should be taken to the app's home screen, which displays all registered devices. You should see one device in the list.

Make sure that you have not received any errors. Errors can be seen from the console, if you did get errors, you did something wrong when following the instructions. Go back and make sure everything is done correctly.

You could alternatively post dummy data to the DynamoDB table to see that everything is working correctly. To do this, navigate to DynamoDB console -> Tables and select the temperature_data table. Next, select Explore table items and Create item. Set thingname as 'dummything' and timestamp as '123456789'. To add

Appendix 1 34(37). Guide to setting up the ESP32 temperature sensor

a temperature field, select Add new attribute and Number, giving the field the name 'temperature' and a value of 30. Finish by clicking Create item.

Next, go back into the app and refresh the list by dragging the content down. You should now see an item with the name of 'dummything'. Tap the item, and you will be taken to another screen that lists all the data that this device has published.

6.1.2 Testing the temperature sensor

The following will describe the steps to test our temperature sensor.

- Connect your device to your computer with an USB cable, this will start your device. Next, we will monitor its output.
- Run one of the commands depending on your shell.
 - Powershell: `.$env:IDF_PATH/export.ps1`
 - Cmd.exe: `%IDF_PATH%/export.bat`
- Next run the command: `'idf.py monitor -p <COM>'`. This will start monitoring your device. You see an output like the one shown in Figure 11. With `'NimBLE: Started advertising.'` being the last command. This tells us that our device has started advertising itself as a BLE server and that it can be connected to.
- Next, we will connect to it with our Flutter app and send it data. This data will consist of a Wi-Fi SSID, Wi-Fi password and a thingName. ThingName will be the name that we want to register our device with.
- Open the Flutter app, sign-in if needed and press the large floating action button with the '+' icon located on the home screen.
- Follow the onscreen instructions to enable Bluetooth and if also required the GPS.
- Start scanning for device and connect to the temperature sensor.
- Bond with the device using the popup. Next fill in the required fields and send the data.

Appendix 1 35(37). Guide to setting up the ESP32 temperature sensor

And that's it! The temperature sensor should start sending temperature data after connecting to the Wi-Fi, registering itself with AWS IoT Core.

Depending on the Wi-Fi stations signal strength, it may take some time for the device to connect to it, but give it sometime and it should successfully connect to it. If not try another Wi-Fi station.

In the case of the temperature sensor failing to connect to Wi-Fi with a message of: 'BLE_PROV: Connection to wifi failed.', make sure that you have sent the correct Wi-Fi SSID and password. And that the network bandwidth of the access point is/supports 2.4GHz (some ESP32 device don't support 5GHz), and the Wi-Fi access points security protocol corresponds to the device's security protocol, by default this is WPA2, but you can change it in menuconfig (see Chapter 4 SETUP ESP32 DEVICE).

Appendix 1 36(37). Guide to setting up the ESP32 temperature sensor

```

Windows PowerShell
I (258) esp_image: segment 4: paddr=00115378 vaddr=403845fc size=0fdb0h ( 64944) load
I (276) boot: Loaded app from partition at offset 0x20000
I (277) boot: Disabling RNG early entropy source...
I (288) cpu_start: Pro cpu up.
I (297) cpu_start: Pro cpu start user code
I (297) cpu_start: cpu freq: 160000000 Hz
I (297) cpu_start: Application information:
I (300) cpu_start: Project name:      ble_provisioning
I (306) cpu_start: App version:      692fdaf-dirty
I (311) cpu_start: Compile time:     Oct 31 2023 16:22:19
I (317) cpu_start: ELF file SHA256:  b7713258e0eeb76d...
I (323) cpu_start: ESP-IDF:         v5.0.2
I (328) cpu_start: Min chip rev:    v0.3
I (333) cpu_start: Max chip rev:    v0.99
I (338) cpu_start: Chip rev:       v0.3
I (343) heap_init: Initializing. RAM available for dynamic allocation:
I (350) heap_init: At 3FCA4880 len 00037E90 (223 KiB): DRAM
I (356) heap_init: At 3FCDC710 len 00002950 (10 KiB): STACK/DRAM
I (363) heap_init: At 50000020 len 00001FE0 (7 KiB): RTCRAM
I (370) spi_flash: detected chip: generic
I (374) spi_flash: flash io: dio
W (378) spi_flash: Detected size(4096k) larger than the size in the binary image header(2048k). Using the size in the binary image header.
I (391) coexist: coexist rom version 9387209
I (396) cpu_start: Starting scheduler.
Opening Non-Volatile Storage (NVS) handle storage... Done
Getting my_wifi_ssid... Error (ESP_ERR_NVS_NOT_FOUND) getting my_wifi_ssid!
Wifi has NOT been provisioned...
Starting ble.
I (426) BTDM_INIT: BT controller compile version [80abacd]
I (436) phy_init: phy_version 950,11a46e9,Oct 21 2022,08:56:12
I (466) system_api: Base MAC address is not set
I (466) system_api: read default base MAC address from EFUSE
I (466) BTDM_INIT: Bluetooth MAC: f4:12:fa:18:b6:ee

I (476) BLE_PROV: BLE Host Task Started
I (486) NimBLE: GAP procedure initiated: stop advertising.

I (486) NimBLE: Device Address:
I (496) NimBLE: ee:fd:47:b0:8d:51
I (496) NimBLE:

I (496) NimBLE: GAP procedure initiated: advertise;
I (506) NimBLE: disc_mode=2
I (506) NimBLE: adv_channel_map=0 own_addr_type=1 adv_filter_policy=0 adv_itvl_min=0
adv_itvl_max=0
I (516) NimBLE:

I (516) NimBLE: Started advertising.

```

Figure 11. IDF monitor command output

Appendix 1 37(37). Guide to setting up the ESP32 temperature sensor

REFERENCES

Amazon Web Services 2023a. Control access to a REST API using Amazon Cognito user pools as authorizer. Referenced 10.11.2023

<https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-integrate-with-cognito.html>

Amazon Web Services 2023b. Query. Referenced 14.11.2023

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html

Amazon Web Services 2023c. Scan. Referenced 14.11.2023

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Scan.html

Android Developers 2023. Advanced emulator usage. Referenced 06.11.2023

<https://developer.android.com/studio/run/advanced-emulator-usage>