Frankfurt University of Applied Sciences

– Faculty of computer science and engineering

Department of Information Technology–

Project Digitalization

-Text your IoT Device-

Documentation for

the development of an IoT application for capturing and remotely monitoring room climate data via Telegram

Group 2

Mohammadjavad Esmaeili (1323388)

Hani Rezaei (1265310)

Submission Deadline: 21.02.2025

Instructor: Prof. Dr. Oliver Hahm

This project focuses on bridging the gap between the physical and digital worlds by creating an interactive interface for IoT devices using instant messaging platforms like Telegram. Using the open-source IoT operating system RIOT, this project develops firmware for microcontroller-based Smart Objects, allowing smooth communication with users over IPv6.

The software solution integrates IoT devices with cloud-based backends using AWS IoT Core for data processing, device control, and management. Participants will collaborate in teams, using Git for version control, and work with technologies such as MQTT protocols and RIOT OS. For encryption and security, the project implements TLS encryption. The project aims to provide hands-on experience in IoT device programming, cloud integration, and real-time communication through modern messaging platforms.

Abbreviation List

IoT Internet of things

AWS Amazon web services

API Application Programming Interfaces

SNS Simple Notifications Service

RIOT Real-time Internet of Things

MQTT Message Queuing Telemetry Transport

TLS Transport Layer Security

IPv6 Internet Protocol version 6

IAM Identity and Access Management

SDK Software Development Kit

CLI Command Line Interface

MIT (License) Massachusetts Institute of Technology

SNS Simple Notification Service (already in the list)

FE Frontend

BE Backend

DK Development Kit

EC2 Amazon Elastic Compute Cloud

VPC Virtual Private Cloud

IGW Internet Gateway

ACM Abstract Control Model

List of Figures

List of Tables

Chapter 1

Introduction

Motivation

The Internet of Things (IoT) is revolutionizing the way physical devices interact with the digital world. As smart devices become more prevalent in daily life, there is a need for user-friendly and efficient methods to communicate with them. Instant messaging platforms like Telegram are widely used and offer an intuitive way for users to control IoT devices. This project explores integrating IoT with messaging apps to create an accessible and seamless user experience, eliminating the need for specialized applications or technical expertise.

Problem Statement

Despite advancements in IoT technology, interacting with IoT devices remains complex for many users. Existing solutions often require dedicated apps or technical knowledge, creating barriers for non-experts. Additionally, IoT devices typically rely on intricate communication protocols that do not easily integrate with simple messaging tools. This project addresses these challenges by enabling IoT devices to communicate over IPv6 and integrating them directly with Telegram, while utilizing AWS IoT services for device management and data processing.

Structure

This document is structured into three main sections:

- **Frameworks and Technologies:** This section discusses the core technologies and frameworks used in the project, including the RIOT operating system, AWS IoT Core, and MQTT for data communication. The Telegram Bot API is utilized for user interaction, and Microsoft Teams is used for project management and collaboration.
- **Hardware and Implementation:** This section provides a detailed overview of the hardware components, including the NRF52840 Dongle and NRF52840 DK, used as IoT devices. The Raspberry Pi 4 serves as a gateway, while the BME280 sensor is employed for environmental data collection. Additionally, the section covers firmware setup, flashing procedures, and network configurations necessary for seamless integration with the cloud.
- **Results, Conclusion, and Future Work:** This section presents the testing results, evaluates the project's achievements, and discusses potential improvements and future directions.

This introduction provides an overview of the project's motivation, the key challenges it addresses, and a structured guide to the document's contents.

Chapter 2

Frameworks and Technologies

Project Management

Effective project management is essential to ensure a well-structured workflow, timely completion, and successful collaboration among team members. By organizing tasks and responsibilities systematically, we can track progress efficiently and address challenges proactively.

Use of Microsoft Teams for Project Management:

For this project, we divided our work into four milestones to ensure an organized workflow and structured progress. To facilitate collaboration, we used Microsoft Teams for communication and project management. Microsoft Teams is a digital platform that integrates chat, video meetings, file storage, and collaborative tools, making it an ideal choice for teamwork. By utilizing Teams, we were able to share updates, assign tasks, and track progress efficiently [1].

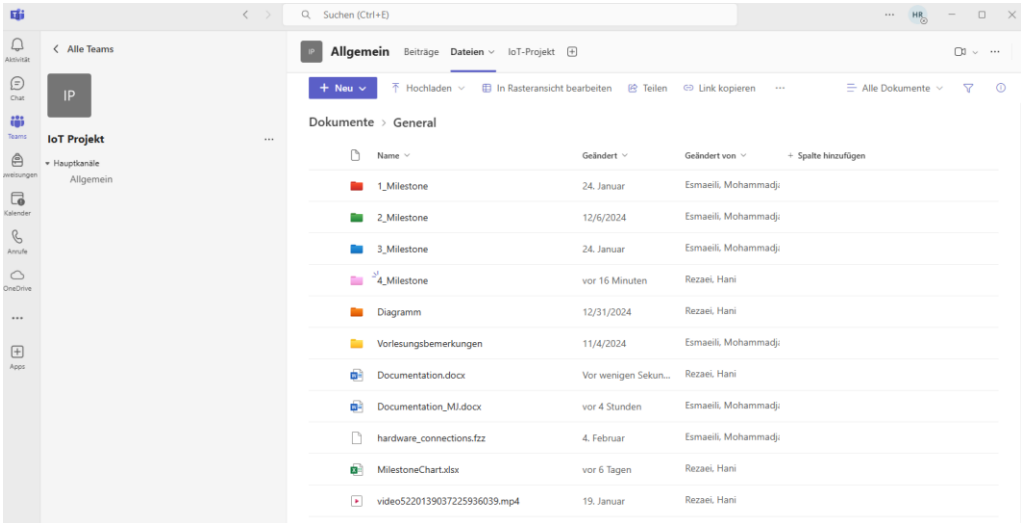

Figure 2.1: Microsoft Teams as Project management

Microsoft Teams allowed us to:

- Communicate effectively through instant messaging, scheduled meetings, and video calls.
- Share files and documents in a central location, ensuring that everyone had access to the latest versions.
- Manage tasks and deadlines using Planner and To-Do lists, which helped in tracking individual and group responsibilities.
- Maintain project transparency by keeping all discussions and updates in dedicated channels.

Project Milestones

To achieve our project goals, we followed a milestone-based approach. The key milestones are outlined below:

Milestone 1: Submission and Presentation of Architecture (November 25, 2024)

- Defined the overall system architecture and requirements.
- Identified the key technologies and frameworks to be used.
- Assigned responsibilities among team members.
- Created an initial design and documented the system's components.
- Presented the architecture to gather feedback and make necessary adjustments.

Milestone 2: Walking Skeleton Presentation (January 20, 2025)

- Developed a minimal but functional version of the system (a "walking skeleton").
- Integrated core components to establish basic connectivity.
- Conducted a preliminary demonstration to validate the approach.
- Addressed feedback from the first milestone and refined the design.

Milestone 3: Final Presentation (February 10, 2025)

- Completed the full implementation of the system.
- Ensured the system met all functional and performance requirements.
- Conducted extensive testing to verify stability and usability.
- Delivered a live demonstration showcasing the final product.
- Collected feedback for further improvements before final submission.

Milestone 4: Submission (February 21, 2025)

- Finalized and documented the entire project.
- Uploaded the completed code to the repository.
- Granted access to Prof. Dr. Oliver Hahm for review.
- Submitted all required reports and documentation.

Project Planning and Results Tracking

To effectively manage our timeline and ensure alignment with our milestones, we prepared detailed charts and schedules using Excel. These charts provided a clear visual representation of our progress, deadlines, and task distribution. We will include a screenshot of these planning charts to illustrate our structured approach.

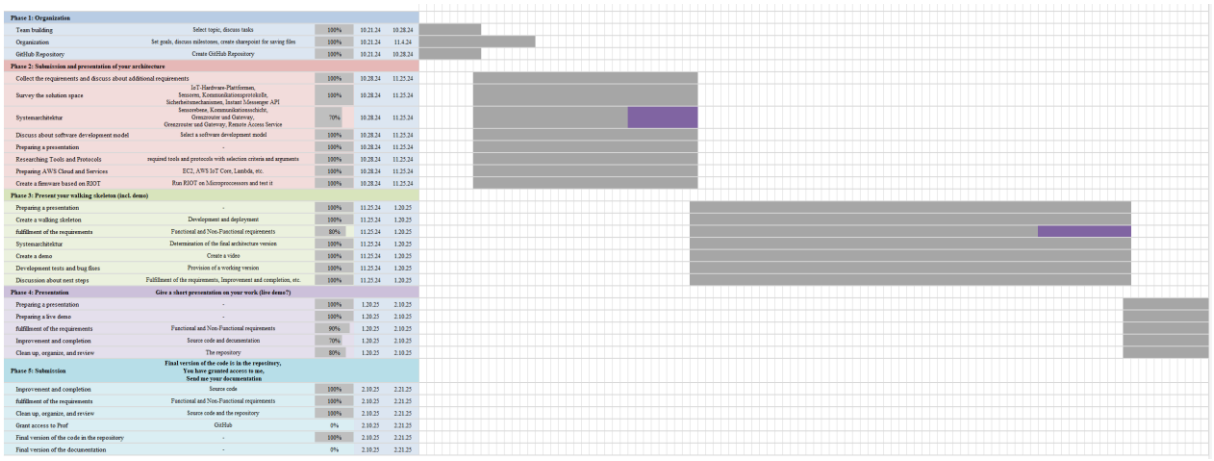

Figure 2.2: charts and schedules

By following this structured methodology, we ensured that our project was completed on time and met all required objectives.

Telegram Bot API

Telegram bots are automated programs that operate within the Telegram messaging app. These bots make it easy to offer services such as sending messages, providing information, or controlling external systems, like IoT devices. Telegram offers an open and developer-friendly platform for “bot” creation. The API is easy to use and well-documented, making it suitable for a wide range of applications, including IoT projects [2].

Why Telegram?

Telegram offers several advantages over other platforms like WhatsApp or Slack, especially for IoT projects [2]:

Ease of Integration:

Unlike WhatsApp bots, which require a complex Business API and additional requirements (like a phone number), Telegram is much more open. You can start developing a bot immediately.

Free to Use:

Telegram does not charge for using its Bot API. Messaging, webhooks, and features are unlimited and free.

Real-Time Interaction:

Telegram is highly compatible with webhook, allowing efficient real-time communication and control for a wide range of applications. With webhook, you can exchange notifications and control commands in real time.

Cross-Platform Availability

Telegram works on almost all devices (iOS, Android, Web), making it easily accessible for users worldwide.

Steps to Create a Telegram Bot

Telegram's BotFather is an official tool for generating and managing bots.

- Open Telegram and search for "@BotFather."
- Start the chat with /start and create a new bot using /newbot.
- After that, we need to choose a name and a username for our bot. The username should end with "bot." Once completed, our bot is created in a Telegram channel. We also receive the HTTP API token, which we need for our API calls.

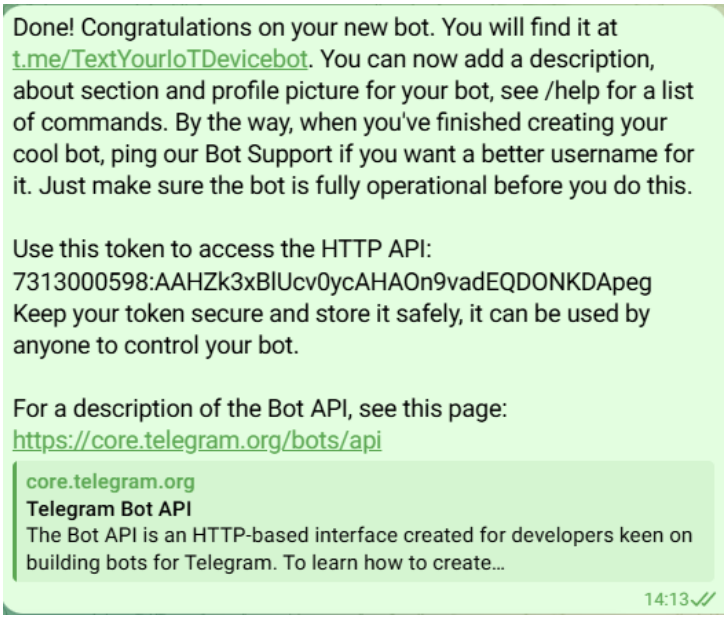

Figure 2.3: Create a telegram bot and received the API Token

Webhook

A webhook is a mechanism that allows one application to send real-time data to another application via a specific URL. Instead of polling for updates (actively requesting data), a webhook enables an application to push updates directly to a predefined endpoint whenever an event occurs [3].

In the case of Telegram bots, a webhook is used to send user messages or commands directly to your backend system in real time. This ensures instant processing of user interactions, making it ideal for IoT projects where timely responses are critical [3].

Setting Up a Webhook Using Postman

Postman provides a user-friendly interface to make HTTP requests for setting up the webhook [4].

To create a New POST Request, follow these steps:

- Click on the "New" button and select "Request".
- Name your request (e.g., “Set Telegram Webhook”) and save it in a collection if needed.
- Use the Telegram Bot API URL for setting the webhook. The URL should look like this: ***https://api.telegram.org/bot<BotToken>/setWebhook***

Configure the Body of the Request:

- Switch to the "Body" tab in Postman.
- Select "form-data" as the format.

Add a key-value pair with the following details:

- Key: url
- Value: ***https://<api-gateway-endpoint>***

Send the Request:

Click "Send" to make the POST request.

Telegram will respond with a confirmation JSON object if the webhook was set up successfully:

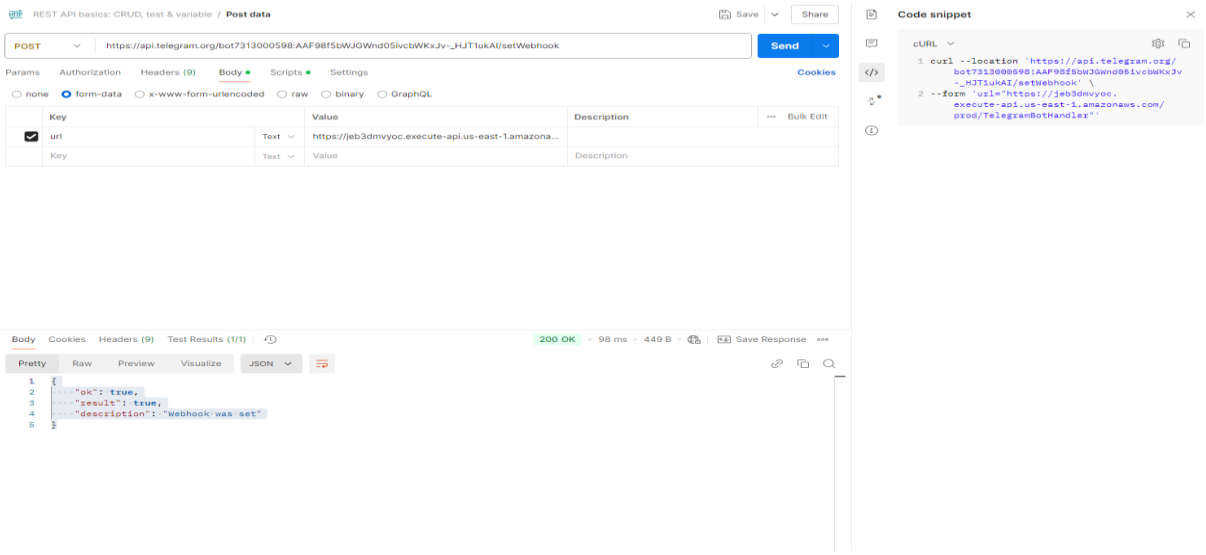

Figure 2.4: Setting up a Webhook using Postman

Amazon Web Services

Amazon Web Services (AWS) is a comprehensive cloud computing platform offering a wide range of on-demand services, such as computing power, storage, networking, databases, and more. These services are available through a pay-as-you-go model, enabling organizations to dynamically and efficiently scale their infrastructure without the need for physical hardware. AWS has been the backbone of this project, providing reliable, secure, and scalable solutions to build, deploy, and manage my application seamlessly [5].

API Gateway

Amazon API Gateway is a fully managed service for creating, deploying, and managing APIs. It acts as an interface between external clients and backend services, ensuring secure, efficient communication. API Gateway supports both RESTful and WebSocket APIs, making it suitable for a variety of use cases, such as data exchange, real-time updates, or integration with AWS services like Lambda and DynamoDB [6].

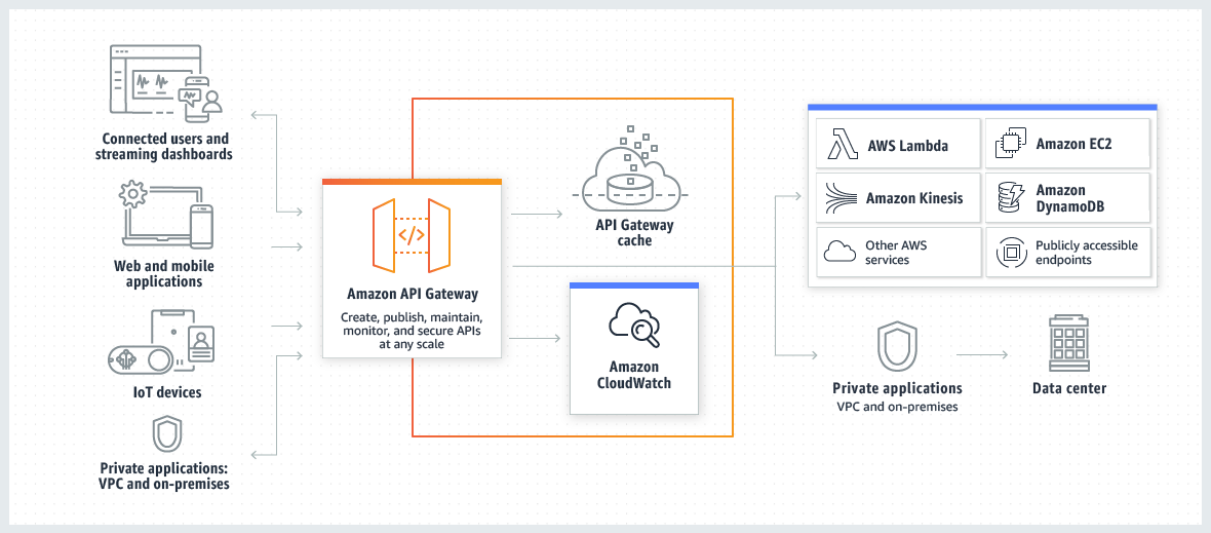

Figure 2.5: AWS API Gateway [6]

API Gateway ensures robust security by offering built-in authentication and authorization mechanisms, including AWS Identity and Access Management (IAM), API keys, and token-based authentication methods.

In this project, an API Gateway named **lambdaHandlerAPI** is implemented to act as the central entry point for Telegram webhooks, facilitating real-time interactions with the Telegram bot.

- **Endpoint:** The API Gateway is configured with a RESTful API endpoint: https://twhmeirf1a.execute-api.us-east-1.amazonaws.com. Telegram sends incoming messages and updates to this endpoint.
- **Integration:** The gateway integrates directly with AWS Lambda, forwarding these incoming requests to a Lambda function for processing. The Lambda function then performs actions like sending responses back to Telegram or triggering other backend services.
- **Security:** To ensure secure communication, we configured the API Gateway to require only HTTPS traffic and used an IAM role to grant Lambda access.

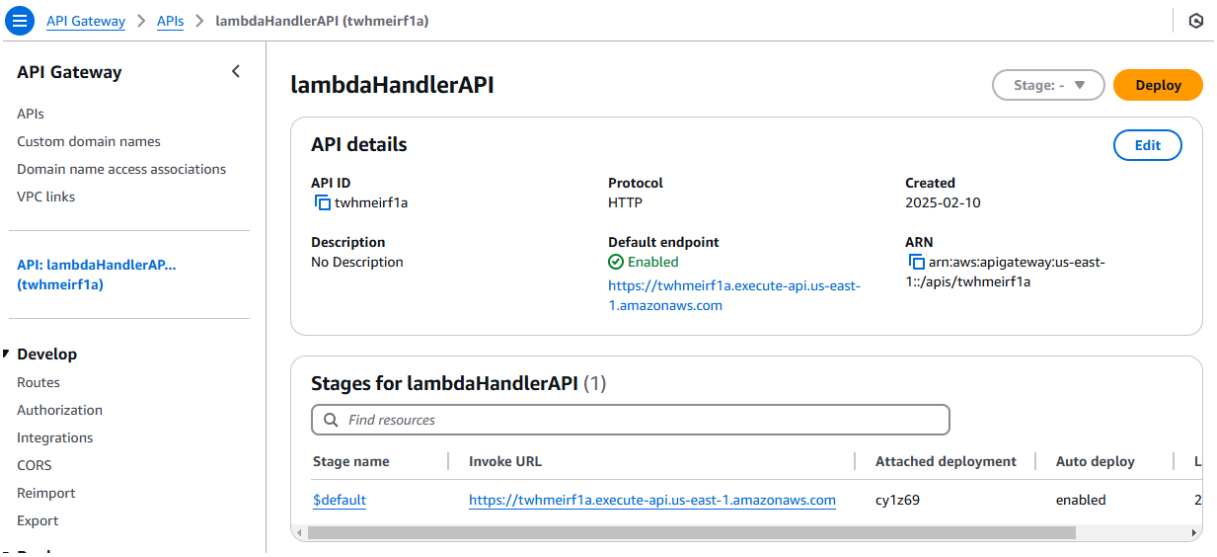

Figure 2.6: lambdaHandlerAPI

Lambda

AWS Lambda is a serverless computing service that allows you to run code in response to predefined events without provisioning or managing servers. It operates on an event-driven model where code execution is triggered by specific actions, such as an HTTP request or a database update [7].

The serverless computing service AWS Lambda plays a central role in this project, processing all communication between the Telegram bot and IoT Core. A Lambda function named **“lambdaHandler”** is implemented to handle incoming webhook requests and facilitate seamless interaction between the bot and IoT devices.

Functionality:

- **Processing User Messages:** Messages received from Telegram users are processed within the Lambda function. Based on the message content, the function determines the appropriate action to take.

- **Interaction with IoT Devices:** If required, the function sends commands or requests to IoT devices via IoT Core and retrieves the necessary information. This allows for real-time control and data exchange with connected devices.

- **Direct Responses:** If no interaction with IoT devices is necessary, the Lambda function generates a direct response to the Telegram bot, ensuring fast and efficient communication.

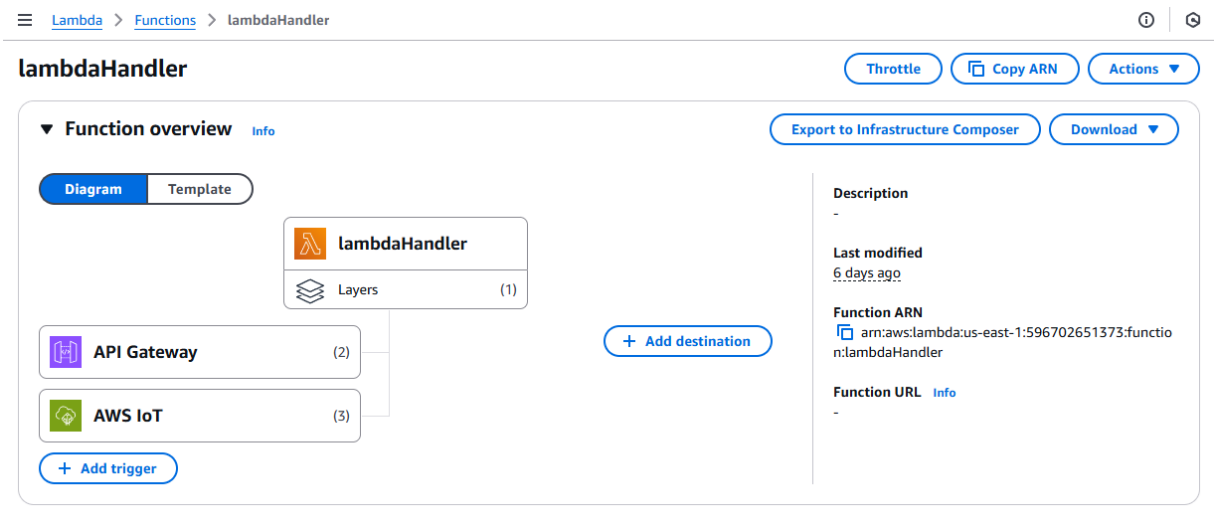

Figure 2.7: AWS Lambda

Specific comments have been added to the Lambda code to explain how requests are handled and how the function interacts with IoT Core and the bot. For more details on the implementation and logic, visit the project's GitHub repository and refer to the comments within the code.

IoT Core

AWS IoT Core is a managed cloud platform that facilitates the secure connection of Internet of Things (IoT) devices to AWS services. It provides tools for device management, message routing, and real-time data processing, enabling developers to build scalable IoT applications without managing the underlying infrastructure [8].

IoT Core Supports communication protocols like MQTT and HTTPS for secure data exchange between devices and the cloud.

AWS IoT Core is utilized in this project to enable secure and efficient communication between IoT devices and the Telegram bot. The platform handles both incoming and outgoing messages, facilitating seamless interaction with connected devices.

In AWS IoT Core, a Thing represents a device that connects to AWS IoT. It serves as a digital representation of a physical device, allowing interaction with cloud services securely. Each Thing has unique identifiers and can be managed using AWS IoT Core features like certificates, policies, and device shadows [9].

Protocols Used:

MQTT is the primary protocol used due to its lightweight nature and ability to handle low-latency communication efficiently. HTTPS is also used for secure interactions when required.

Creating a Thing in AWS IoT Core

A Thing can be created using the AWS IoT Console, AWS CLI, or AWS SDKs. When creating a Thing, you define its identity, attributes, and security credentials. AWS IoT Core automatically assigns a unique identifier and provides authentication mechanisms to establish secure communication.

AWS IoT Core uses certificates for secure device authentication. When a Thing is created, the following certificates are generated:

Certificate	Description
Device Certificate (. pem.crt)	Authenticates the IoT device
Private Key (. pem.key)	Used for secure authentication
Public Key (.pem)	Optional, used for encryption
Amazon Root CA Certificate	Ensures trusted communication

Table 2.1: Summary of Certificates Generated

These certificates enable secure, encrypted communication between your IoT device and AWS IoT Core.

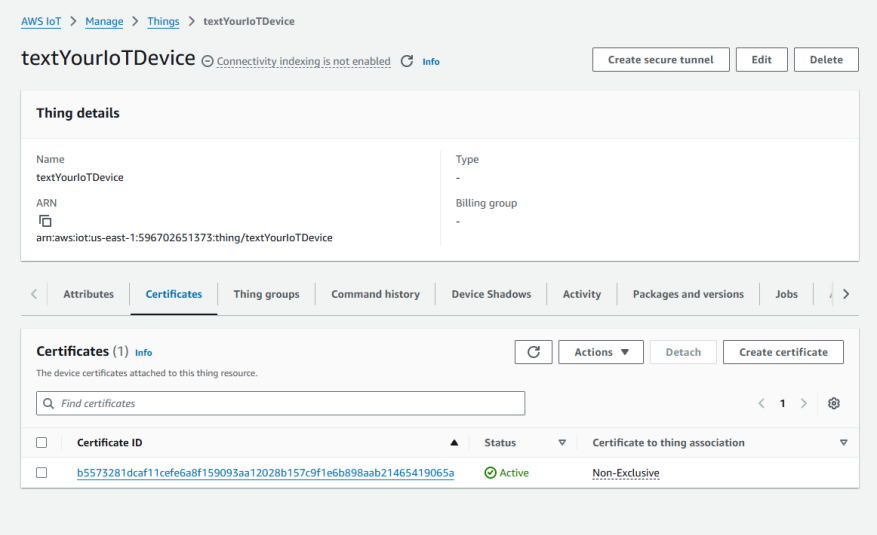

Figure 2.8: AWS IoT Core, Thing

Policies

Policies define the permissions granted to an IoT device in AWS IoT Core. The policy associated with **textYourIoTDevice** is **Policy_TextYourIoTDevice**. It has the following configuration:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
```

Figure 2.9: AWS IoT Core, Thing

This policy grants full access to all AWS IoT actions and resources. Once created, a Thing can securely exchange data with AWS IoT Core, allowing for real-time monitoring and control.

Message Routing: Rules

AWS IoT Core allows message routing using rules, which process and route messages from IoT devices based on defined topics. The following rules are configured in the system:

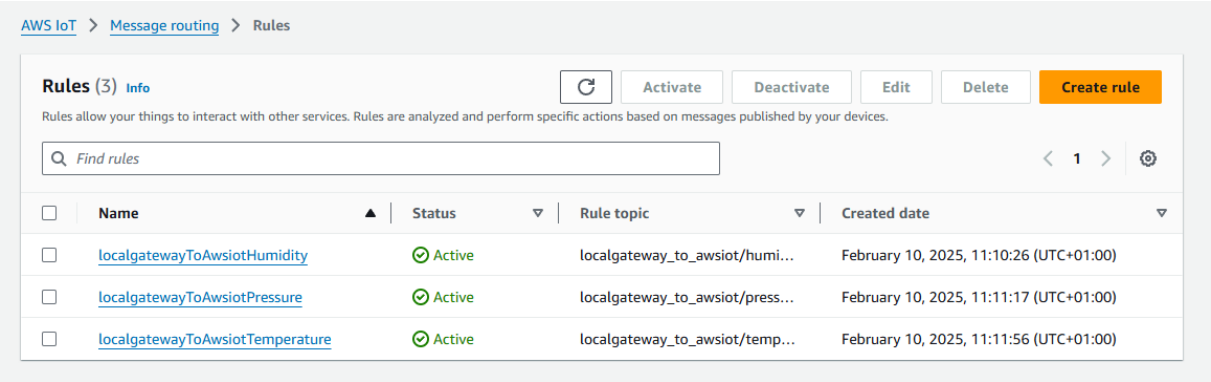

Figure 2.8: AWS IoT Core, Rules

Each rule listens to a specific MQTT topic and processes incoming messages using the following SQL queries:

```
SELECT * FROM 'localgateway_to_awsiot/humidity';
SELECT * FROM 'localgateway_to_awsiot/pressure';
SELECT * FROM 'localgateway_to_awsiot/temperature';
```

Figure 2.10: AWS IoT Core, Rules SQL queries

These rules are connected to an AWS Lambda function (xxx). When an IoT device publishes a message to any of the specified topics, the rule triggers the Lambda function. The Lambda function processes the incoming data and prepares a response, which is then sent to a Telegram bot for further interaction with users.

Implementation:

- **MQTT Topics for Communication:**

The communication between IoT devices and the Lambda function is organized using the following six MQTT topics:

- **For sending requests to IoT devices from the Telegram bot:**
  - awsiot_to_localgateway/humidity
  - awsiot_to_localgateway/pressure
  - awsiot_to_localgateway/temperature

These topics are used when the Telegram bot sends messages to the Cloud. The Lambda function processes the request and publishes messages to these topics on the MQTT broker. These topics are used to request specific data (humidity, pressure, temperature) from the IoT devices.

Here is the part of the Lambda function that processes the publishing data to the topics above:

```
def fetch_data_from_iot(chat_id, function_name, preferences):
    topic_mapping = {
        "get_temperature": "awsiot_to_localgateway/temperature",
        "get_humidity": "awsiot_to_localgateway/humidity",
        "get_air_pressure": "awsiot_to_localgateway/pressure"
    }
    topic = topic_mapping.get(function_name)
    if not topic:
        return "❌ Invalid topic for the requested function."
    preference_value = next(iter(preferences.values()))
    payload = {
        "u": preference_value,  # Assign the unit value to "u"
        "chat_id": str(chat_id)  # Mapp chat_id as string
    }
    try:
        print(f"Publishing to topic: {topic}, Payload: {json.dumps(payload)}")
        response = iot_client.publish(
            topic=topic,
            qos=1,
            payload=json.dumps(payload)
        )
        print(f"Successfully published to {topic}: {response}")
        return f"☑ Fetching {function_name.replace('get_', ' ')} ..."
    except Exception as e:
        print(f"Failed to publish to MQTT: {e}")
        return "❌ Failed to fetch data from IoT Core."
```

Figure 2.11: Description for fetch_data_from_iot() function

The ***fetch_data_from_iot*** function publishes a message to an IoT Core topic based on the provided function_name (e.g., "get_temperature", "get_humidity"). It constructs a payload using a value from the preferences dictionary and attempts to send it to the correct topic via MQTT. If successful, it returns a success message, otherwise, it returns an error message if the topic is invalid or if publishing fails.

- **For receiving responses from IoT devices back to the Telegram bot:**
  - localgateway_to_awsiot/humidity
  - localgateway_to_awsiot/pressure
  - localgateway_to_awsiot/temperature

Once the IoT devices process the request and send the data back, the MQTT broker publishes the response to these topics. The Lambda function subscribes to these topics to retrieve the response data as soon as it's available.

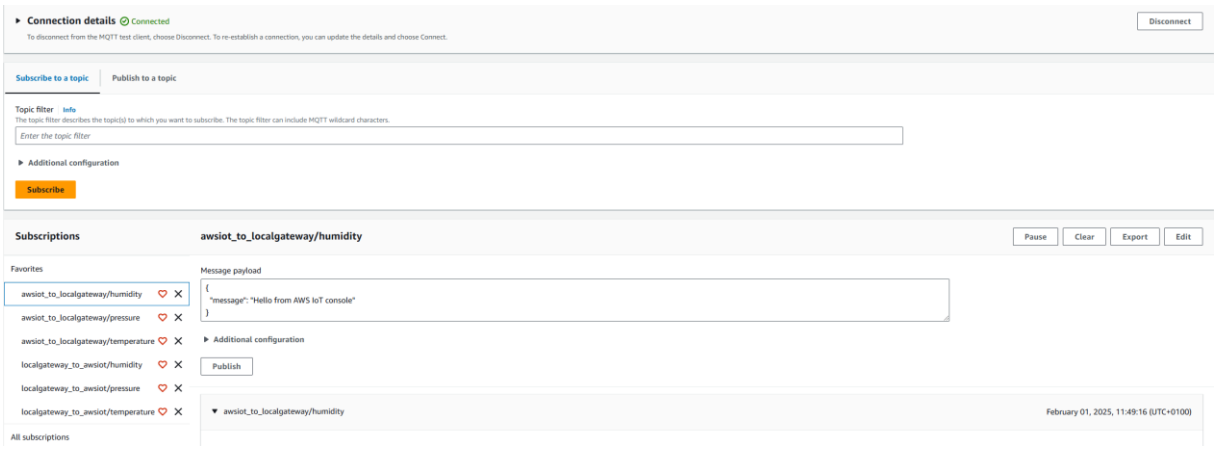

Figure 2.12: AWS IoT Core, MQTT Topics for Communication

This setup allows for real-time, bidirectional communication between the IoT devices and the Telegram bot.

Integration with SNS:

The communication between the Lambda function and IoT Core is further enhanced using Amazon Simple Notification Service (SNS). SNS is used to publish messages that notify IoT Core or Lambda about new events or updates. This integration ensures real-time data flow and simplifies the management of asynchronous communication.

CloudWatch

Amazon CloudWatch is a monitoring and observability service that provides real-time data and insights into AWS resources, applications, and services. It collects and tracks metrics, logs, and events, allowing you to monitor system performance, troubleshoot issues, and set alarms for specific conditions [10].

Amazon CloudWatch is used in this project primarily for capturing and analyzing logs generated by various AWS services, such as the API Gateway and Lambda functions.

CloudWatch Logs are configured to collect detailed information about the execution of the lambdaHandler function. These logs include:

- Incoming requests from the API Gateway.
- Processing details, such as decisions made by the Lambda function (e.g., whether to interact with IoT Core or respond directly to the Telegram bot).
- Outgoing responses sent back to the Telegram bot or IoT Core.

If any errors occur during the execution, they will be logged and can be traced in the CloudWatch logs. This allows for efficient debugging and troubleshooting. The logs provide valuable insights into where the issue occurred, helping the team address problems quickly.

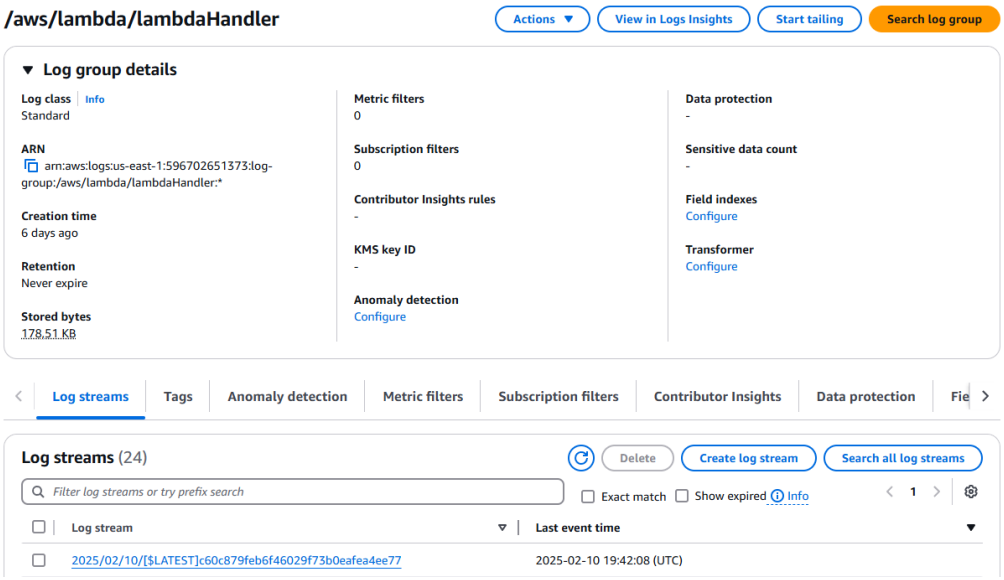

Figure 2.13: AWS CloudWatch, Log groups

By analyzing these logs, patterns can be identified to improve system performance and reliability.

Below is a screenshot of CloudWatch Logs, showing sample outputs from the Lambda function:

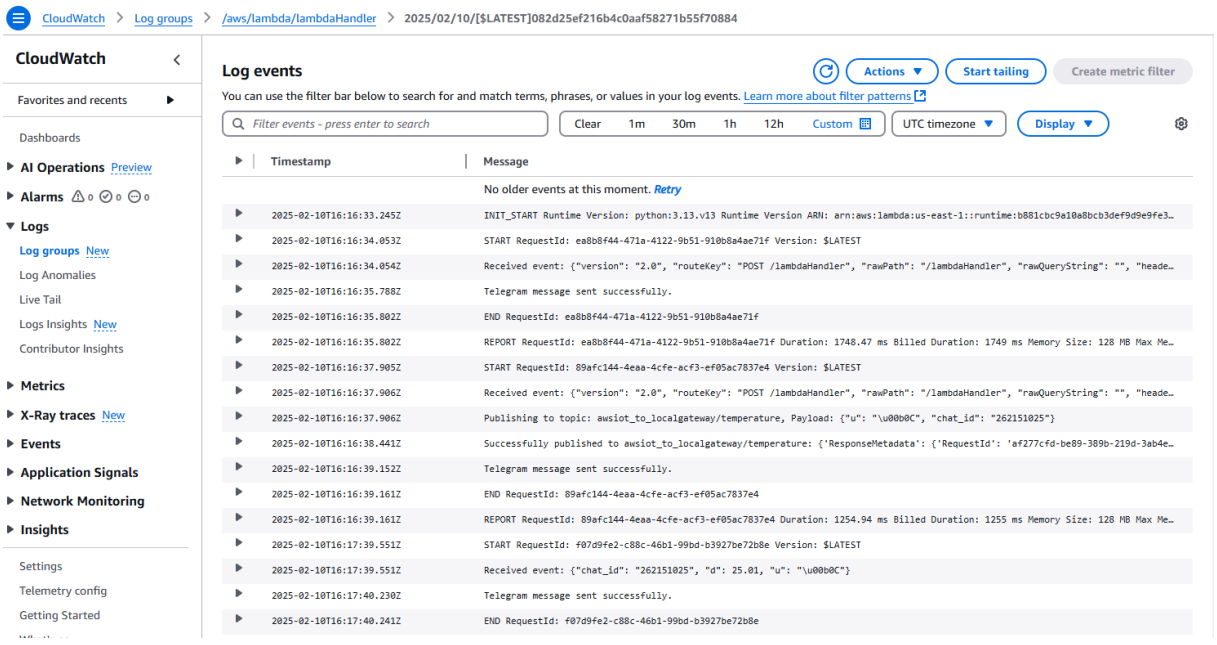

Figure 2.14: AWS CloudWatch, TelegramBotHandler Logs

Chapter 3

Hardware and implementation

NRF52840 Dongle

Figure 3.1: NRF52840 Dongle [11]

The *nRF52840 Dongle* (dongle) is a small, low-cost USB dongle that supports Bluetooth 5.4, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT and 2.4 GHz proprietary protocols [11].

Connecting and powering the NRF52840 Dongle

The dongle connects directly to the host computer via a USB port. Unlike the *nRF52840 DK* (DK), which has a dedicated debug interface, the dongle uses the USB interface for both power and data communication. An LED on the dongle indicates that the device is receiving power. The dongle communicates via a virtual serial interface USB *Communication Device Class – Abstract Control Model* (CDC-ACM). The dongle is delivered with a bootloader as standard. If new firmware needs to be flashed, it is best to use *nRF Connect for Desktop* application [12].

Firmware Setup for NRF52840 Dongle

The *gnrc_border_router* example from RIOT OS is used to configure the dongle as a border router.

The makefile of *gnrc_border_router* on dongle should also be adapted. At first, the name of the board that supports RIOT should be specified. If the board is unknown, the default value *"native"* can be used:

```
# If no BOARD is found in the environment, use this default:
BOARD ?= nrf52840dongle
```

The next step is to adjust the RIOT paths on the local computer. Depending on where the RIOT OS was cloned, the path must be adjusted:

```
# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= $(CURDIR)/../../../RIOT
```

The type of network connection between the IoT device and a host (e.g. PC or router) must also be determined by UPLINK. USB *Communication Device Class – Ethernet Control Model* (CDC-ECM) Simulates an Ethernet connection via USB. It is used when the device appears as a network interface via USB.

```
# Default to using ethos for providing the uplink when not on native
UPLINK ?= cdc-ecm
```

The standard radio channel for wireless communication must be set. This applies to IEEE 802.15.4 protocols such as: Thread (mesh networks), Zigbee and 6LoWPAN. The channel value 22 corresponds to a specific frequency in the 2.4 GHz band (usually 2420 MHz).

```
# Default channel of nRF52840DK
DEFAULT_CHANNEL = 22
```

Integration of default-radio-settings.inc.mk provides central configuration of radio parameters.

```
# Set a custom channel if needed
include $(RIOTMAKE)/default-radio-settings.inc.mk
```

Flashing Firmware on NRF52840 Dongle

To compile and flash, the command is first used:

sudo make -C gnrc_border_router/ all

"sudo" runs the command with administrator privileges. *"make"* calls the build system to compile the source code. *"-C gnrc_border_router/"* means that make will first go to the *"gnrc_border_router/"* directory. *"all"* is a standard target in the makefile that performs all necessary steps to compile.

A *.hex* file is generated which is used for programming dongle, as Dongle requires special binary formats to be flashed. Intel HEX is a widely used format for bootloaders and flash tools.

Here is a detailed guide on how to upload a .hex file using *nRF Connect for Desktop* application:

1. After starting the application, select *"Programmer"*:

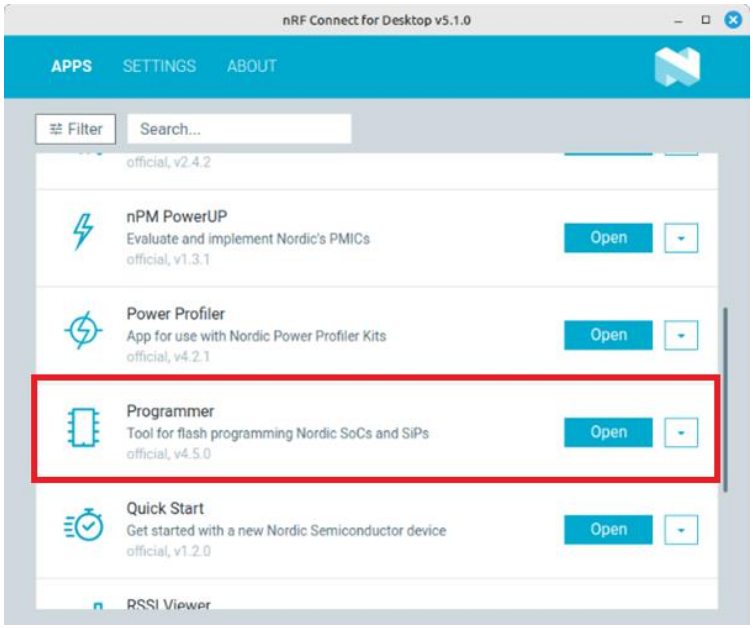

Figure 3.2: Main window nRF Connect for Desktop

2. Make sure the dongle is connected to the computer via USB. Click *"Select Device"* and select the connected device from the list:

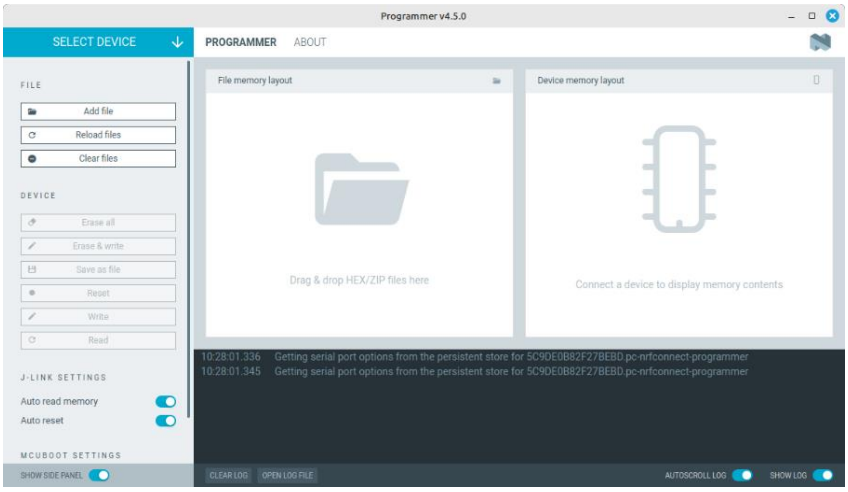

Figure 3.3: Select Device nRF Connect for Desktop

The dongle is recognized as a *bootloader*:

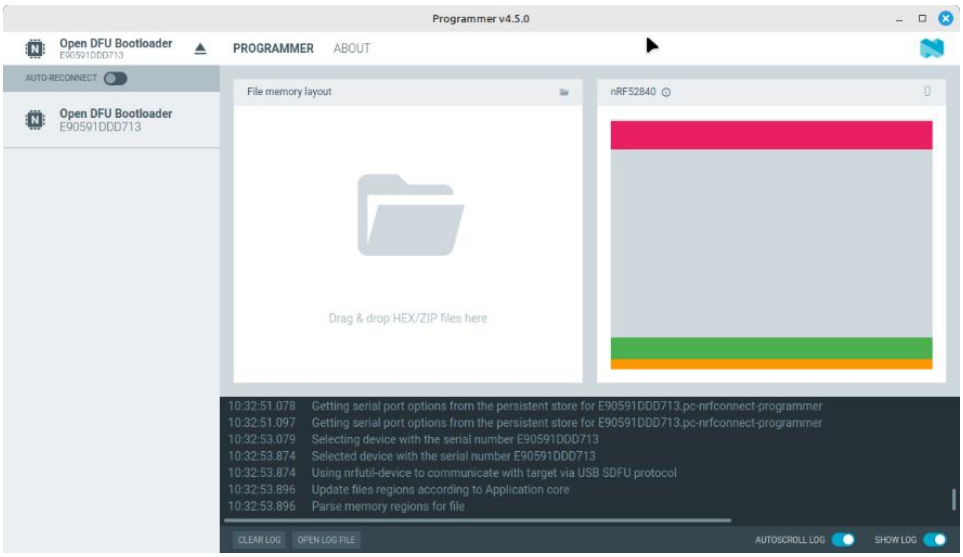

Figure 3.4: Bootloader in nRF Connect for Desktop

3. Click on *"Add file"* in the left sidebar. Select the .hex file that you want to flash to the device. Alternatively, you can drag and drop the file directly into the *"File memory layout"* area:

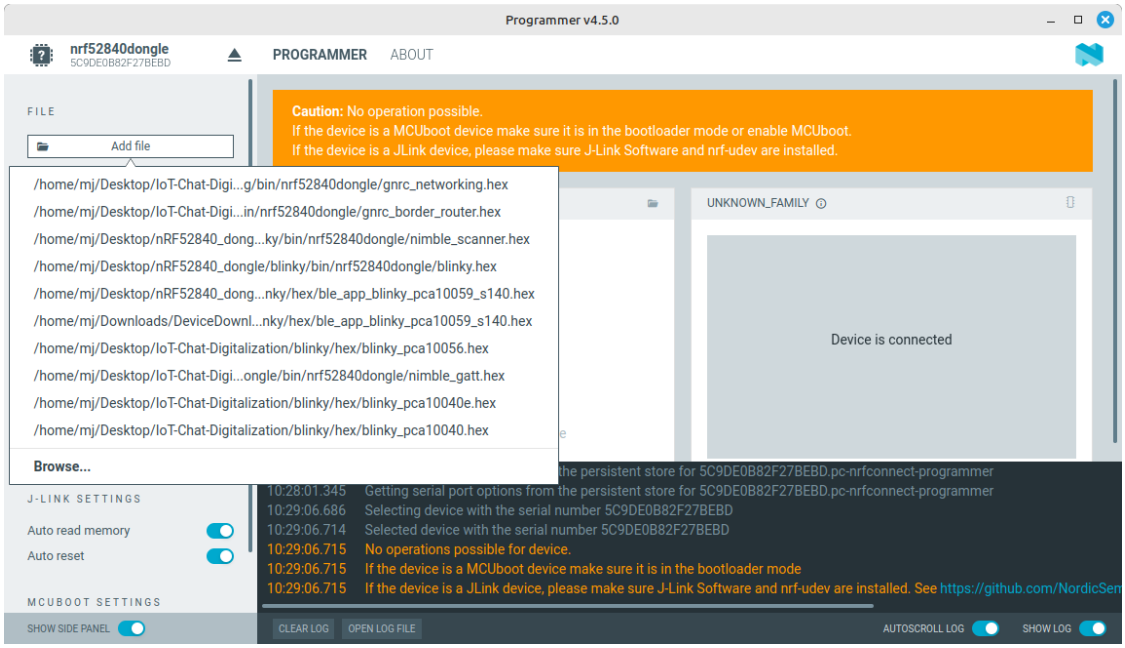

Figure 3.5: Add file nRF Connect for Desktop

Select the generated *.hex* file:

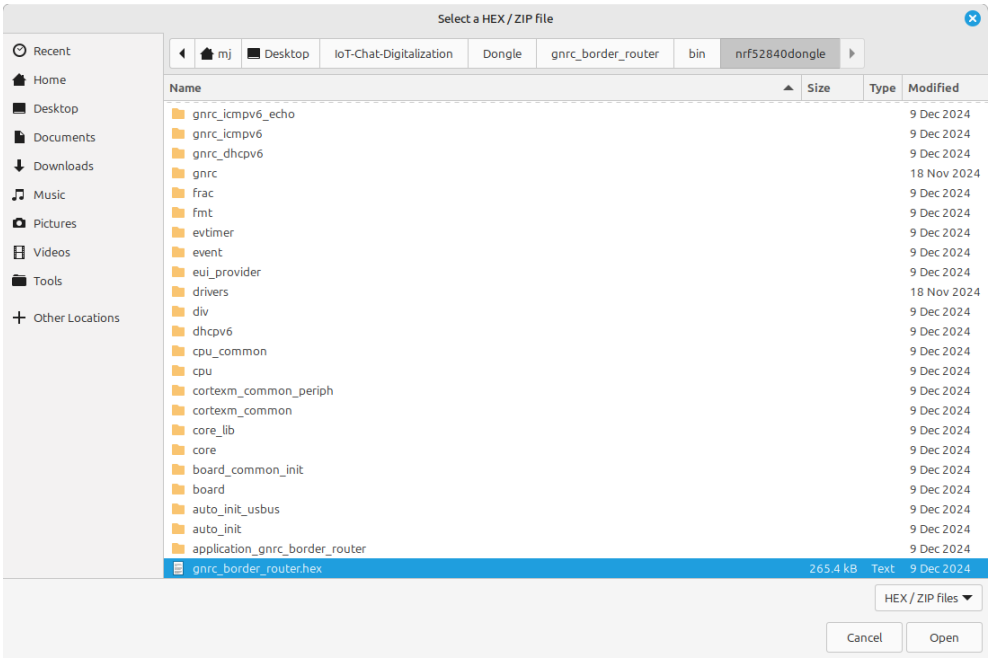

Figure 3.6: .hext file in nRF Connect for Desktop

4. After loading the file, the *"File memory layout"* area will be updated with the memory map. If the device is connected, the current device memory content can be viewed in the right pane. To start the flashing process, click *"Write"*. If necessary, you can erase the memory beforehand using *"Erase all"* or *"Erase & write"*:

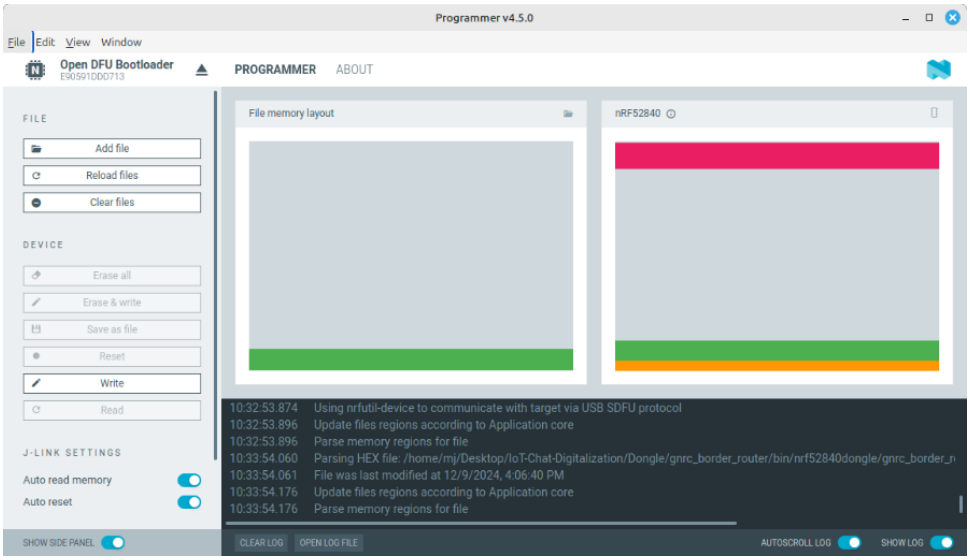

Figure 3.7: Write file in nRF Connect for Desktop

5. In the lower log section, you can track the status of the flashing process in real time. Successful operations will be confirmed accordingly. If the firmware does not start automatically, a reboot may be necessary. To do this, you can manually restart the device or enable the *"Auto reset"* option. After these steps, the new firmware should be successfully flashed onto the dongle board:

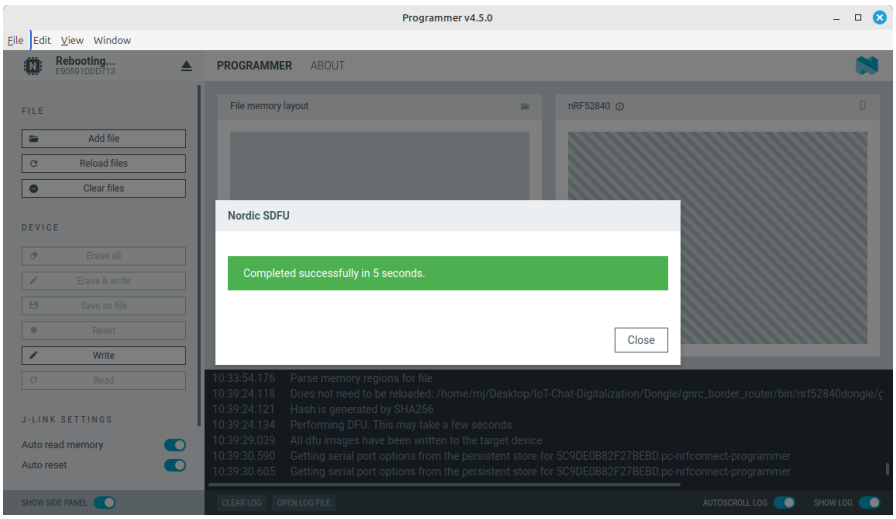

Figure 3.8: Successful flashing process in nRF Connect for Desktop

Testing of NRF52840 Dongle

After the *gnrc_border_router* firmware is running on the dongle, it should provide a 6LoWPAN network interface. This can be checked with the following commands:

Ifconfig

The output should contain a 6LoWPAN interface, e.g.:

```
ifconfig
Iface  6  HWaddr: EE:4F:25:C3:4C:C2
          L2-PDU:1500  MTU:1500  HL:64  RTR
          Source address length: 6
          Link type: wired
          inet6 addr: fe80::ec4f:25ff:fec3:4cc2  scope: link  VAL
          inet6 group: ff02::2
          inet6 group: ff02::1
          inet6 group: ff02::1:ffc3:4cc2

Iface  7  HWaddr: 36:BD  Channel: 22  NID: 0x23  PHY: O-QPSK
          Long HWaddr: 5E:9D:E0:B8:2F:27:B6:BD
           State: IDLE
          ACK_REQ  L2-PDU:102  MTU:1280  HL:64  RTR
          6LO  IPHC
          Source address length: 8
          Link type: wireless
          inet6 addr: fe80::5c9d:e0b8:2f27:b6bd  scope: link  VAL
          inet6 group: ff02::2
          inet6 group: ff02::1
          inet6 group: ff02::1:ff27:b6bd
```

Figure 3.9: Terminal output of dongle

If no interface is displayed, the firmware may not be flashed correctly, or the dongle may not be connected correctly.

NRF52840 DK

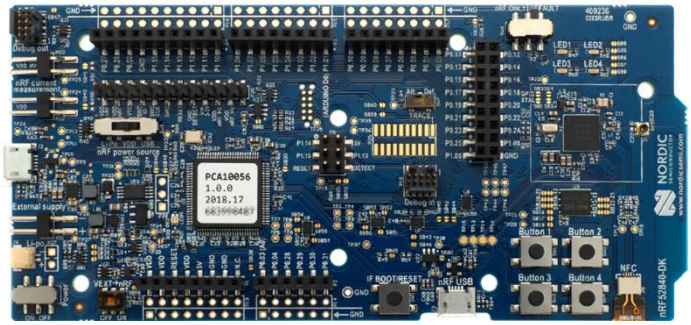

Figure 3.10: NRF52840 DK [13]

The *nRF52840 Development Kit* (DK) is a versatile single-board development kit for Bluetooth Low Energy, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT and 2.4 GHz

proprietary applications. This development kit is based on the nRF52840 SoC and provides developers with an accessible platform for embedded IoT applications.

For more details, refer to the official documentation [14].

Connecting and powering the NRF52840 DK

The DK is connected to a host system using a J-Link interface for power, programming, and debugging. It provides a built-in *Segger J-Link* debugger, enabling serial communication and firmware flashing.

Power can be supplied via USB (micro-USB cable to PC or power adapter) or an external 3V–5V source. When the microcontroller is powered on, the LED in the middle lights up. It should be ensured that power switch (SW6) is set to VDD.

The J-Link interface provides a virtual COM port for serial communication with the nRF52840 DK. To identify the correct serial port on Linux:

ls /dev/ttyACM*

Because RIOT OS is used as firmware in this project, which is explained in the next chapter, the J-LINK interface from RIOT is used for this purpose.

Firmware Setup for NRF52840 DK

Because we are using the RIOT OS as firmware in this project, the RIOT must first be cloned:

git clone https://github.com/RIOT-OS/RIOT.git

The examples *pahu_mqtt*, *saul* and *blinky* are used here as a basis for the various purposes. The *pahu_mqtt* example is used for MQTT communication, as well as the *saul* example for reading the sensor data and the *blinky* example for the delay implementation of the LED lighting.

In order to get the whole thing running and on DK, the Make files must first be adjusted. At the beginning only the *pahu_mqtt* is used. The other parts are included in the main makefile in the next steps and are used.

We enter an application name, which in this case is *"TextYourIoTDevice"*:

```
# Name der Anwendung
APPLICATION = TextYourIoTDevice
```

After that, the name of the board that supports RIOT should be specified. If the board is unknown, the default value *"native"* can be used:

```
# Standard-Board setzen, falls keine spezifische Auswahl getroffen wurde
BOARD ?= nrf52840dk
```

The next step is to adjust the RIOT paths on the local computer. Depending on where the RIOT OS was cloned, the path must be adjusted:

```
# Absoluter Pfad zum RIOT-Basisverzeichnis
RIOTBASE ?= $(CURDIR)/../../RIOT
```

The next setting is to specify the default radio channel for the nRF52840DK. It specifies that the nRF52840DK uses radio channel 22 by default. This setting is relevant for wireless communication, e.g. IEEE 802.15.4 (Zigbee, Thread) or BLE (Bluetooth Low Energy). This is set to 22 by agreement so that the DK and dongle are in the same network in the same channel.

```
# Default channel of nRF52840DK
DEFAULT_CHANNEL = 22
```

Last configuration step in *pahu_mqtt* is to define default values for wireless communication in RIOT OS. Presumably parameters like radio channel, frequency, modulation type or MAC address for different radio modules (e.g. IEEE 802.15.4, LoRa, BLE) will be set.

```
# Set a custom channel if needed
#include $(RIOTMAKE)/default-radio-settings.inc.mk
```

Flashing Firmware on NRF52840 DK

The RIOT build system is used to compile and flash firmware:

Sudo make -C TextYourIoTDevice/ flash term

"sudo" runs the command with administrator privileges.

"make" is the build system that controls the compilation and installation of software.

"-C TextYourIoTDevice/" means that make changes to the *TextYourIoTDevice/* directory before executing further commands. *"flash"* is a makefile target that typically compiles the project and pushes the firmware to the IoT device (via J-Link). *"term"* is another makefile target that starts a serial connection to the IoT device after flashing. Typically, *pyterm* or *minicom* is used to see the device's output in a terminal.

Sometimes it can happen that DK is locked by J-LINK and needs to be unlocked [14]:

***openocd -c 'interface jlink; transport select swd; ***

source [find target/nrf52.cfg]' -c 'init' -c 'nrf52_recover'

or

nrfjprog –recover

Testing of NRF52840 DK

The dongle should now act as a 6LoWPAN border router. This can be checked with the following commands:

Ifconfig

If the DK is successfully connected to the Border Router, it should have an IPv6 address. The output should contain an IPv6 address, e.g.:

```
ifconfig
Iface  6  HWaddr: 51:BA  Channel: 22  NID: 0x23  PHY: O-QPSK
          Long HWaddr: BA:70:1F:42:D0:3F:D1:BA
           State: IDLE
          ACK_REQ  L2-PDU:102  MTU:1280  HL:64  6LO
          IPHC
          Source address length: 8
          Link type: wireless
          inet6 addr: fe80::b870:1f42:d03f:d1ba  scope: link  VAL
          inet6 addr: 2001:470:7347:c218:b870:1f42:d03f:d1ba  scope: global  VAL
          inet6 group: ff02::1
```

Figure 3.11: ifconfig output of DK

The following command is used to establish a serial connection to this connected DK. before executing this command, you must change to the directory:

cd ~/RIOT/dist/tools/pyterm/

. /pyterm runs the pyterm script (local file in the current directory). -p /dev/ttyACM0 specifies the serial device to connect to. /dev/ttyACM0 is the typical device name for USB-connected microcontrollers with a serial interface under Linux.

. /pyterm -p /dev/ttyACM0

Raspberry Pi 4 Model B

The *Raspberry Pi* (Pi) requires a 5V power supply via USB-C.

The dongle is plugged directly into a USB port on the Pi.

Figure 3.12: Raspberry Pi 4 Model B [15]

In this setup, the Raspberry Pi takes on the role of a gateway and an infrastructure bridge between the dongle and the rest of the network. Its main tasks are:

- Providing a stable interface for the dongle: The dongle is connected to the Pi via USB. The Pi communicates with the dongle via a serial interface (UART or USB).
- Border router management: The dongle runs the RIOT OS *gnrc_border_router* example and acts as a 6LoWPAN border router. The Pi ensures the control and configuration of the border router.
- Data forwarding and protocol translation: Received sensor data from 6LoWPAN nodes (e.g. via the DK) is received by the dongle. The Raspberry Pi processes and forwards this data to the MQTT broker system (e.g. Mosquitto on EC2).

Sensor BME280

Figure 3.13: Sensor BME280 [16]

The *BME280* sensor, developed by Bosch, is designed for environmental monitoring, measuring humidity, barometric pressure, and temperature. It supports both I2C and SPI communication interfaces.

Key specifications [17]:

- Humidity: ±3% accuracy
- Pressure: ±1 hPa absolute accuracy
- Temperature: ±1.0°C accuracy
- Altitude estimation: ±1 meter

Firmware Setup for the sensor BME280

To read the sensor data from the BME280, configurations for Inter-integrated circuit (I2C) communication must be included in the makefile of *"saul"* [18]:

```
# BME280 connected via I2C
USEMODULE += bme280_i2c
# When using I2C, specify the default I2C device to use,
# and the BME280's address (see datasheet).
# The values below are the defaults:

CFLAGS += -DBMX280_PARAM_I2C_DEV="I2C_DEV(0)"
CFLAGS += -DBMX280_PARAM_I2C_ADDR=0x77
```

Figure 3.14: BME280 Settings in makefile of saul

The given configuration ensures that the BME280 sensor can be addressed correctly via the I2C interface in RIOT OS. First, the corresponding driver module is integrated using *USEMODULE += bme280_i2c* so that the necessary functions for communication with the sensor are available. Then *CFLAGS += -DBMX280_PARAM_I2C_DEV="I2C_DEV(0)"* is

used to specify that the microcontroller uses the first I2C device *(I2C_DEV(0))* for communication. In addition, the I2C address of the BME280 sensor is set to *0x77*, which corresponds to the default address when the SDO pin is set to high (VCC). If the sensor was configured to address *0x76* instead, this value would have to be adjusted accordingly. These settings allow the microcontroller to address the BME280 sensor via the correct I2C bus and retrieve the sensor data for temperature, air pressure and humidity.

Connecting the sensor BME280 with nRF52840 DK

After the makefile configuration, the wiring can be done properly before you perform tests to collect sensor data. The hardware connection diagram below shows how the sensor is connected to the DK via I2C:

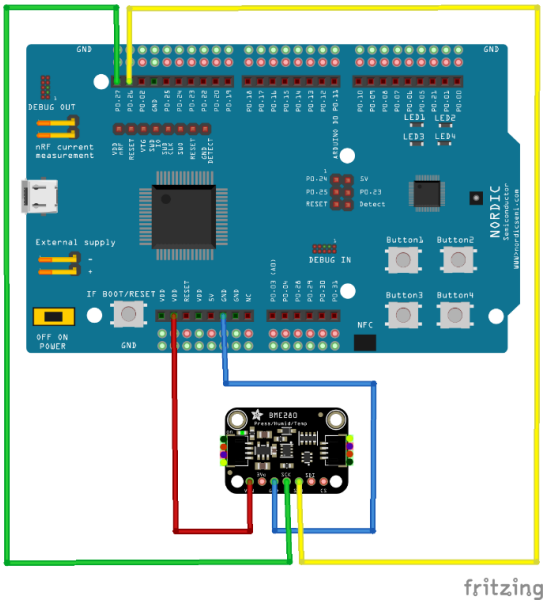

Figure 3.15: Connecting the sensor BME280 with DK

Connection via I2C:

- VCC (BME280) → 3.3V (DK)
- GND (BME280) → GND (DK)
- SDA (BME280) → SDA (Pin 26 on DK)
- SCL (BME280) → SCL (Pin 27 on DK)
- SDO (BME280) → GND (for I2C address 0x76) or 3.3V (for I2C address 0x77)

MQTT Broker on AWS EC2

Amazon Elastic Compute Cloud (EC2) provides scalable computing resources in the cloud, allowing users to run virtual machines with flexible configurations. EC2 instances can host applications such as MQTT brokers, facilitating IoT communication between devices and cloud services.

Create an instance of AWS EC2

The steps for creating an EC2 instance are explained here. After the Sign in to AWS Console navigate to AWS EC2 und launch an instance:

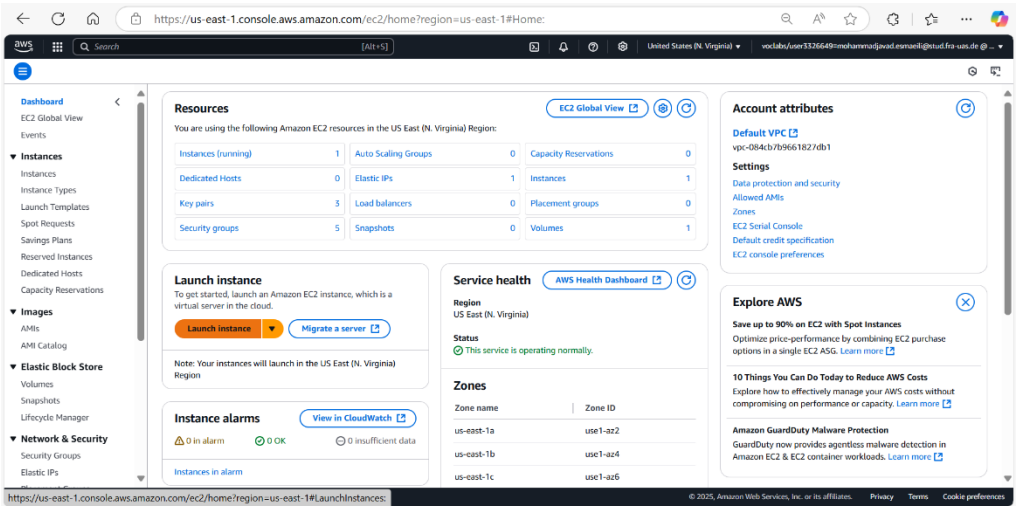

Figure 3.16: Main window of AWS EC2 instance

To create an instance, an important settings must be made. The instance name is *“VM_TextYourIoTDevice”*. For the next step, an Amazon machine image must be selected, in this case *“Ubuntu Server 24.04 LTS (HVM), SSD Volume Type”*.

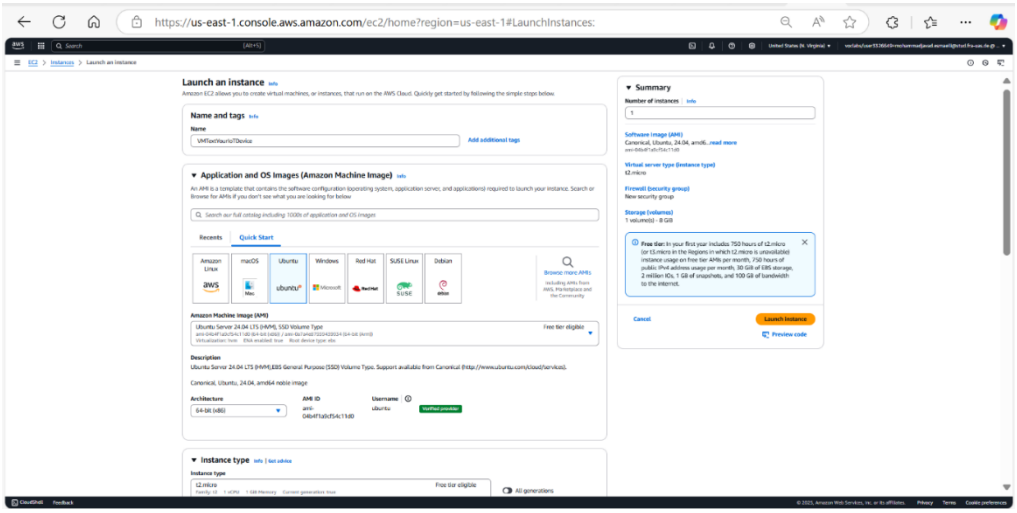

Figure 3.17: Settings of AWS EC2 instance

Next, a key pair is required. We need it to connect securely to the instance. Depending on your needs, the key pair type can be selected between *Rivest–Shamir–Adleman* (RSA) and ED25519.

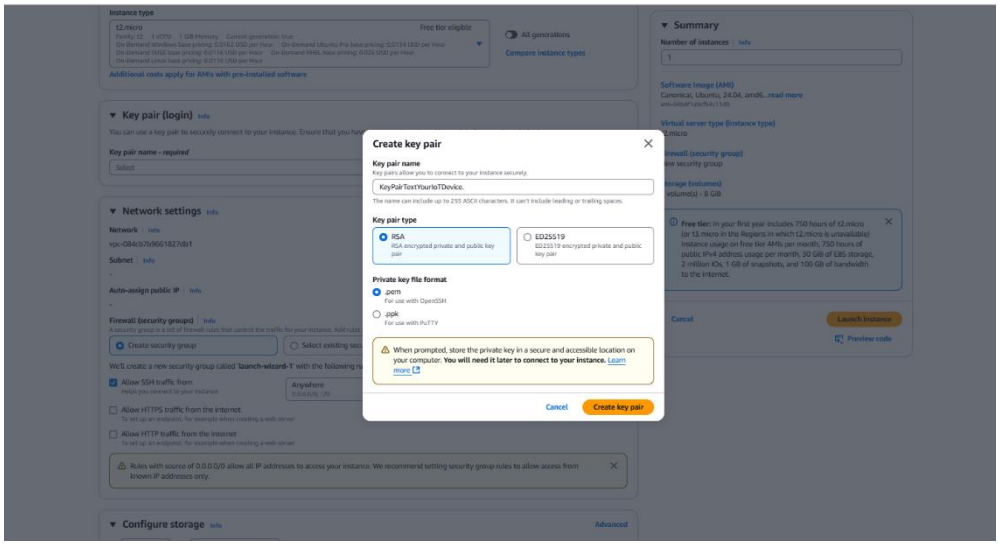

Figure 3.18: Key pair settings of AWS EC2 instance

An important step is the network and security settings:

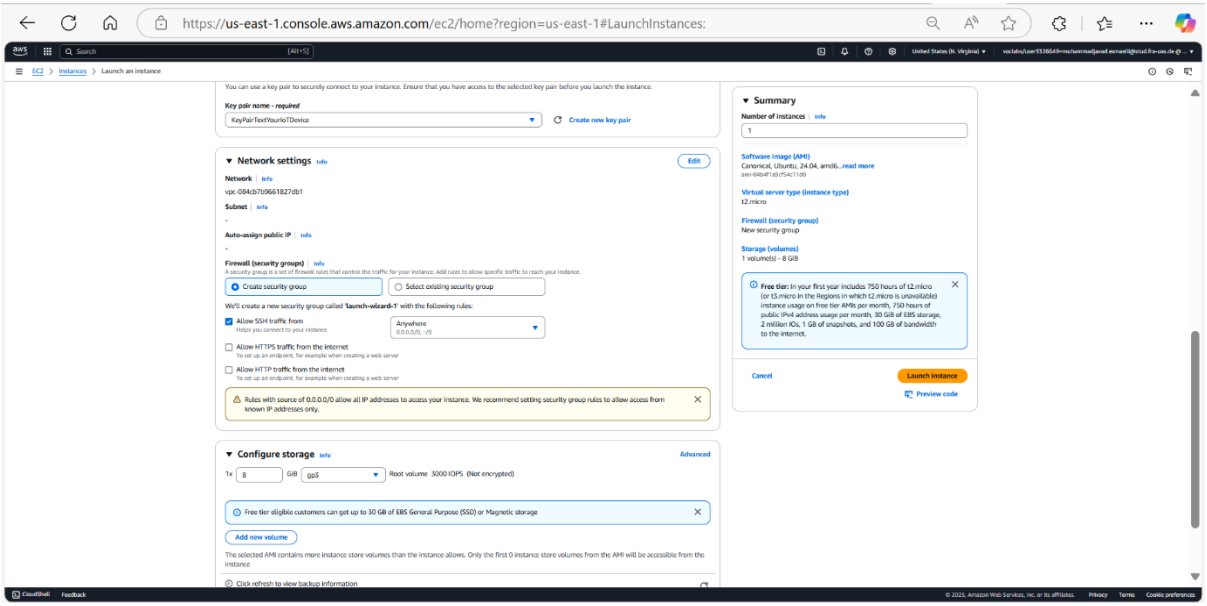

Figure 3.19: Network settings of AWS EC2 instance

Virtual Private Cloud (VPC)

A *Virtual Private Cloud* (VPC) is an isolated network within AWS that allows resources to be managed securely and in a controlled manner. The EC2 instance runs within a VPC so that it can be reached via private and public IP addresses. By default, AWS creates a default VPC, but it may be useful to create a separate VPC for the IoT system to better manage network security and access control. VPC Controls traffic via firewall rules (security groups and network ACLs) and enables secure access to the instance through subnets and routing.

IPv4 settings:

- IPv4 CIDR block: 10.0.0.0/24

IPv6 settings:

- Amazon-provided IPv6 CIDR block: AWS automatically assigns a /56 prefix from the global IPv6 range.

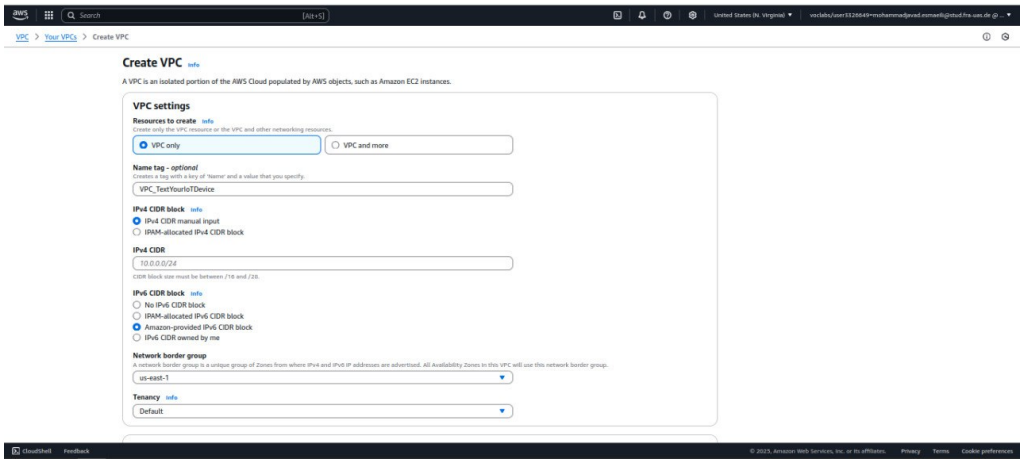

Figure 3.20: VPC settings

Subnet

A *subnet* is a sub-area of the VPC that defines specific IP address ranges for instances. In AWS, there are public and private subnets:

- Public subnet has access to the Internet via an Internet gateway.
- Private subnet has no direct Internet access; communication runs via NAT or other internal services.

The EC2 instance with the MQTT broker should be in a public subnet so that external IoT devices can access it via the Internet. Subnet defines which IP address ranges are used by the EC2 instance, allows the separation of public and private network areas, and controls the accessibility of the EC2 instance for external clients.

- Settings for IPv4:
  - IPv4 CIDR block: 10.0.1.0/24 (subnet from the VPC)
- Settings for IPv6:
  - IPv6 CIDR block: AWS automatically assigns a /64 subnet from the IPv6 prefix of the VPC.
- Routing:
  - The subnet must be defined as public, i.e. it needs a route to the Internet gateway.
- Automatic IP assignment:
  - Activate "Enable auto-assign public IPv4 and IPv6 addresses".

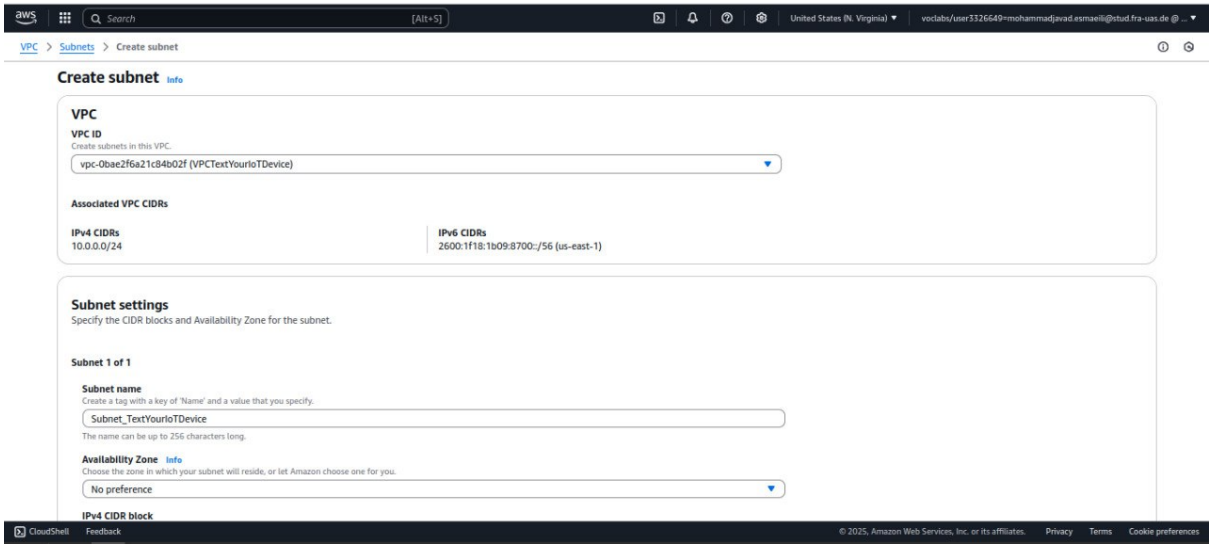

Figure 3.21: Subnet settings

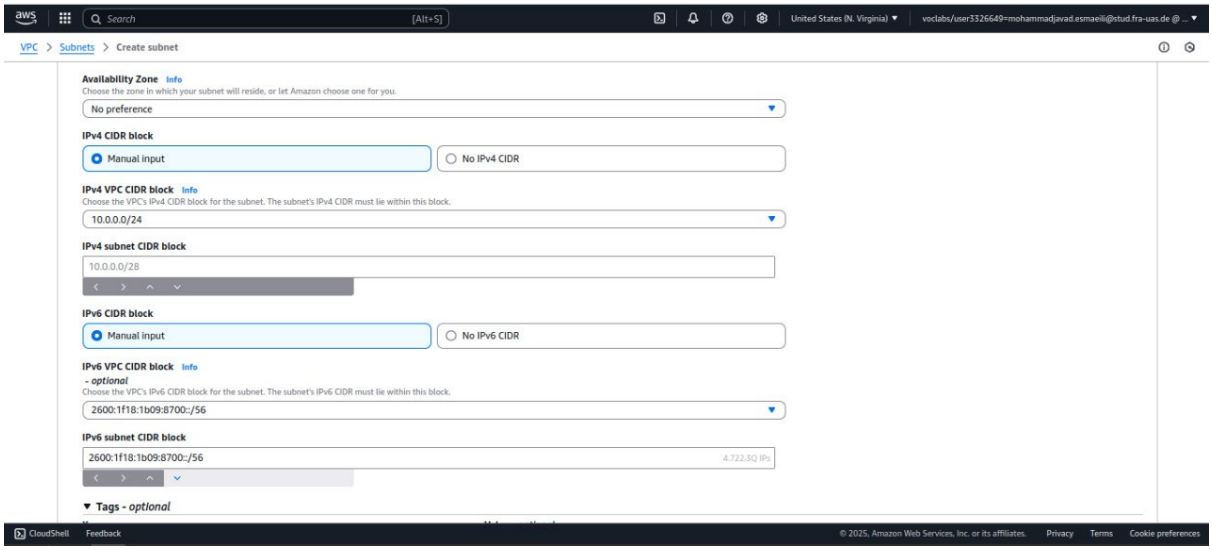

Figure 3.22: Subnet IP settings

Internet Gateway (IGW)

An *Internet Gateway* (IGW) enables resources in the VPC (e.g. the EC2 instance) to communicate with the Internet. Without an IGW, the EC2 instance would not be able to establish connections to external MQTT clients or cloud services. IGW allows outbound and inbound connections over the public Internet, makes the EC2 instance reachable in a public subnet, and is essential for running an MQTT broker that accepts connections from IoT devices.

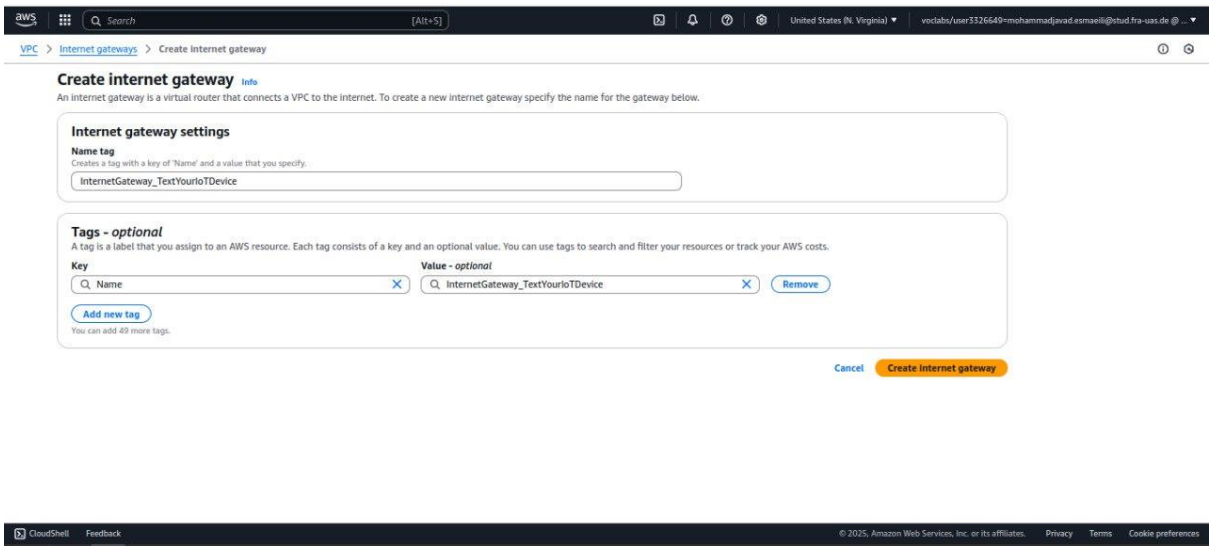

Figure 3.23: Internet gateway settings

It is important to remember to connect the IGW to the VPC by attaching it to VPC after creation:

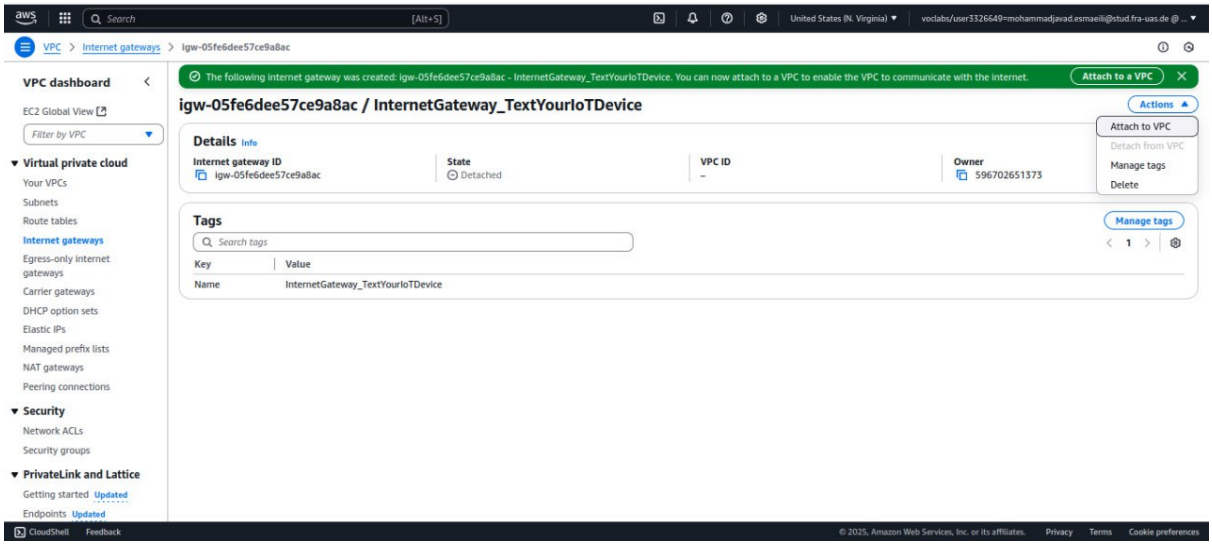

Figure 3.24: Attach internet gateway

Route table

A *route table* defines how network traffic is routed within the VPC. By default, there is a main route table, but different routes can be configured for public and private subnets. For the public subnet where the EC2 instance is located, a route to the Internet Gateway (0.0.0.0/0 → IGW ID) must be added. Route Table ensures that all outbound Internet traffic is routed through the Internet Gateway, defines the network flow within the VPC (e.g. communication between MQTT broker and other AWS services). Without the route, the EC2 instance would not be reachable from the outside.

IPv4 route:

- Destination: 0.0.0.0/0
- Destination type: Internet Gateway
- Destination: Select the Internet Gateway created previously

IPv6 route:

- Destination: ::/0
- Destination type: Internet Gateway
- Destination: Select the Internet Gateway created previously

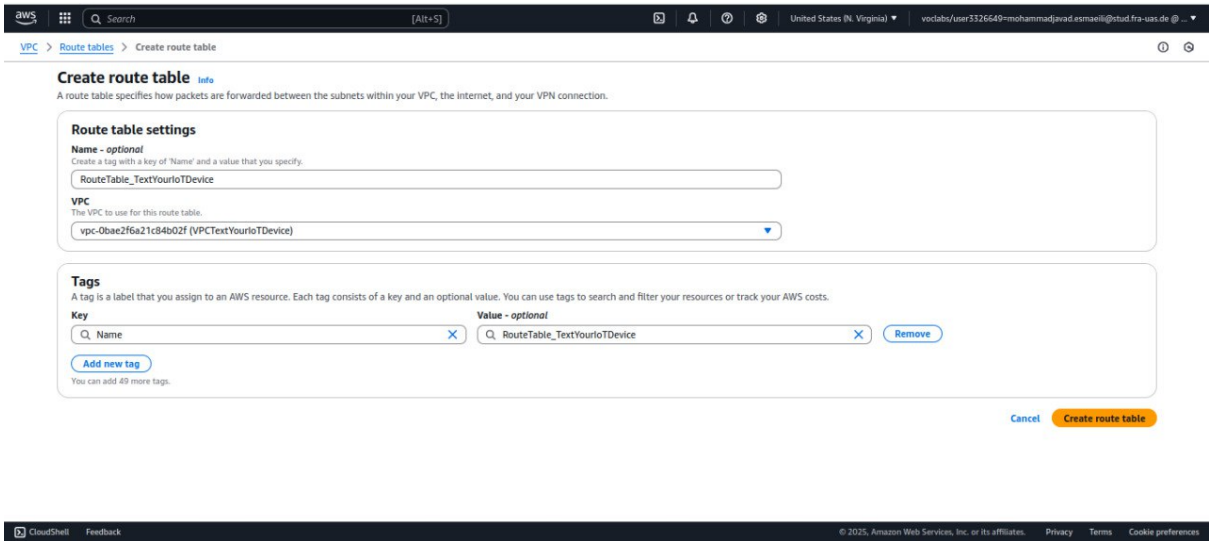

Figure 3.25: Route table settings

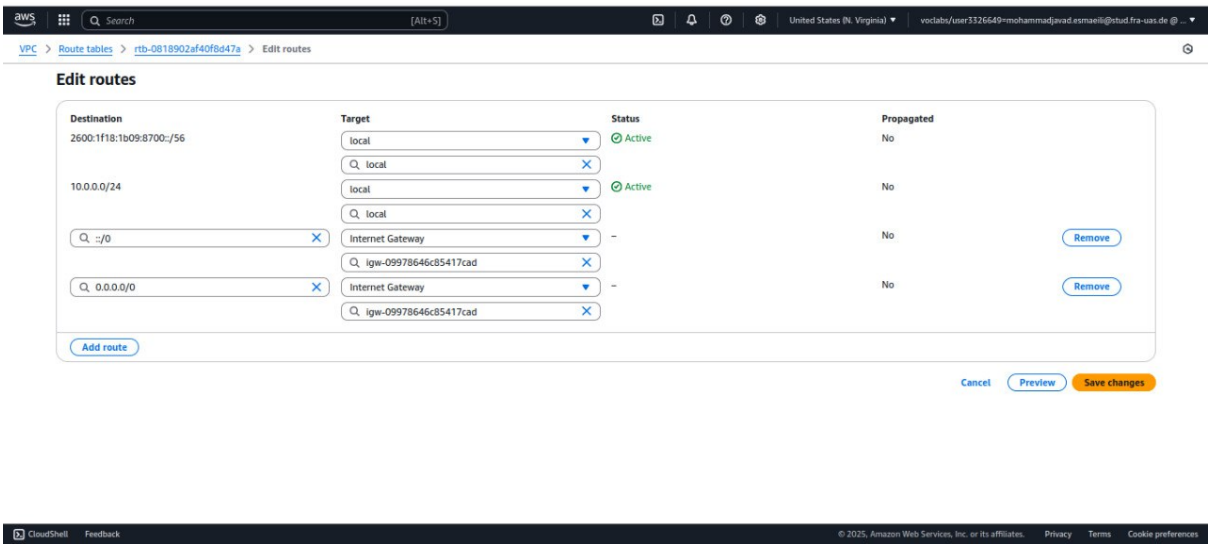

Figure 3.26: edit routes settings

Security Group

An AWS *Security Group* (SG) is a virtual firewall for EC2 instances that controls inbound and outbound network traffic. It works at the instance level and determines which connections are allowed or blocked.

The SG ensures that only IoT devices (DK & Pi) and AWS IoT Core can access the broker. Unauthorized connections are blocked. The Security Group limits connections to trusted IPs or IPv6 prefixes. Since the IoT project is based on IPv6, the Security Group must be configured accordingly.

The Inbound Rules for the Security Group:

Port	Protocol	Source	Purpose
8883	Custom TCP	::/0 (IPv6)	MQTT over TLS for secure communication
8080	Custom TCP	::/0 (IPv6)	Access to RIOT IoT service without TLS
22	SSH	0.0.0.0/0 (IPv4)	SSH access to the EC2 instance
All	All traffic	::/0 (IPv6)	Default configuration for the EC2 connection

Table 3.1: Inbound Rules for the Security Group

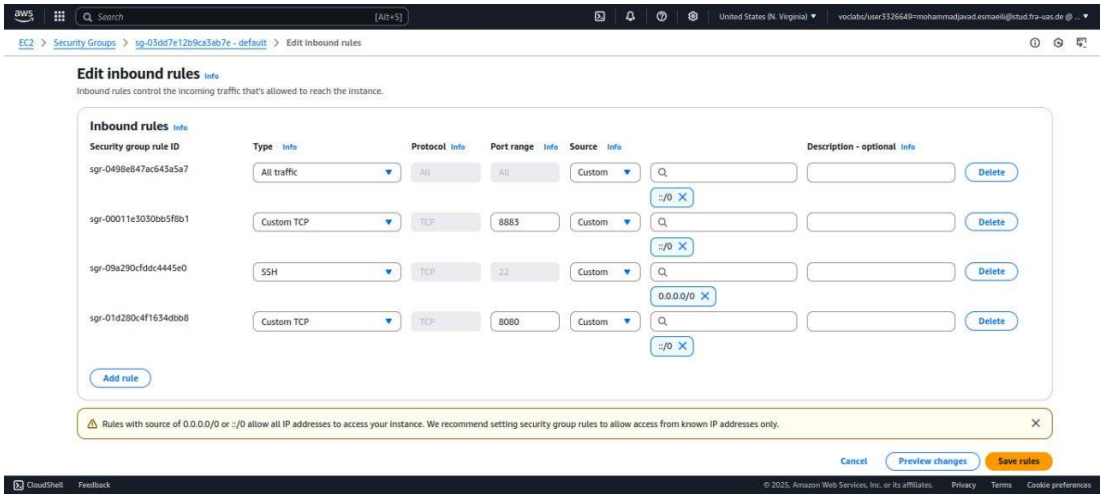

Figure 3.27: Inbound rules settings

Outbound Rules for the Security Group:

Port	Protocol	Source	Purpose
8883	Custom TCP	::/0 (IPv6)	MQTT over TLS for secure communication
8080	Custom TCP	::/0 (IPv6)	Access to RIOT IoT service without TLS
22	SSH	0.0.0.0/0 (IPv4)	SSH access to the EC2 instance
All	All traffic	::/0 (IPv6)	Default configuration for the EC2 connection

Table 3.2: Outbound Rules for the Security Group

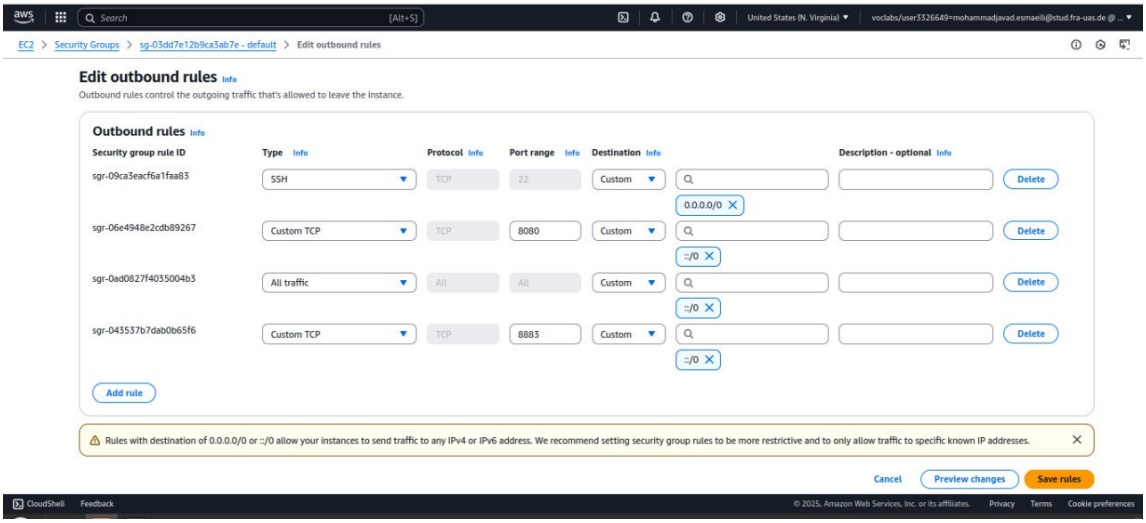

Figure 3.28: Outbound rules settings

After the Network settings you can see IPV6 addresses in the Networking tab on the main page

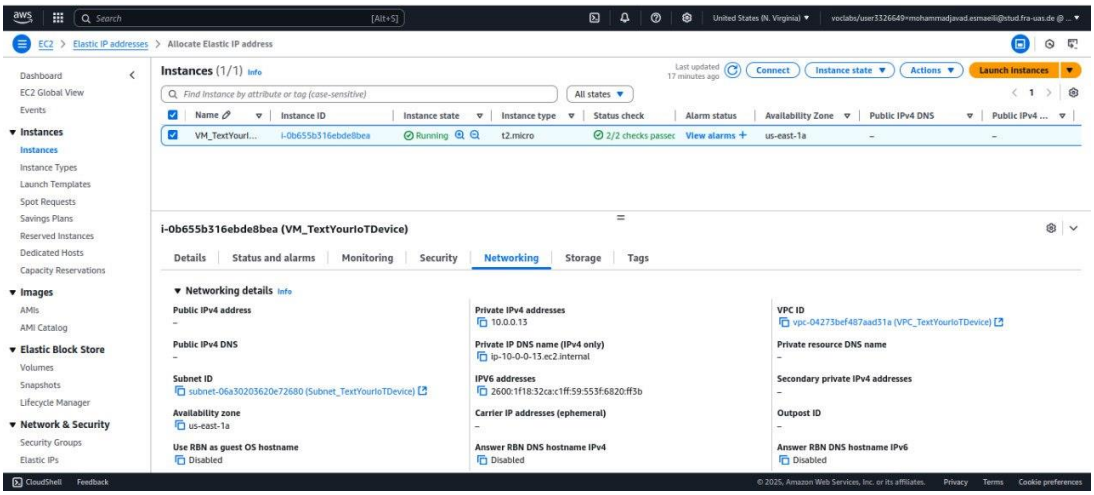

Figure 3.29: Network settings in AWS EC2 instance

IPV6 is used to connect to the instance. The username is *"ubuntu"* by default. The SSH client connection will be used later when uploading the security certificates to the EC2 instance.

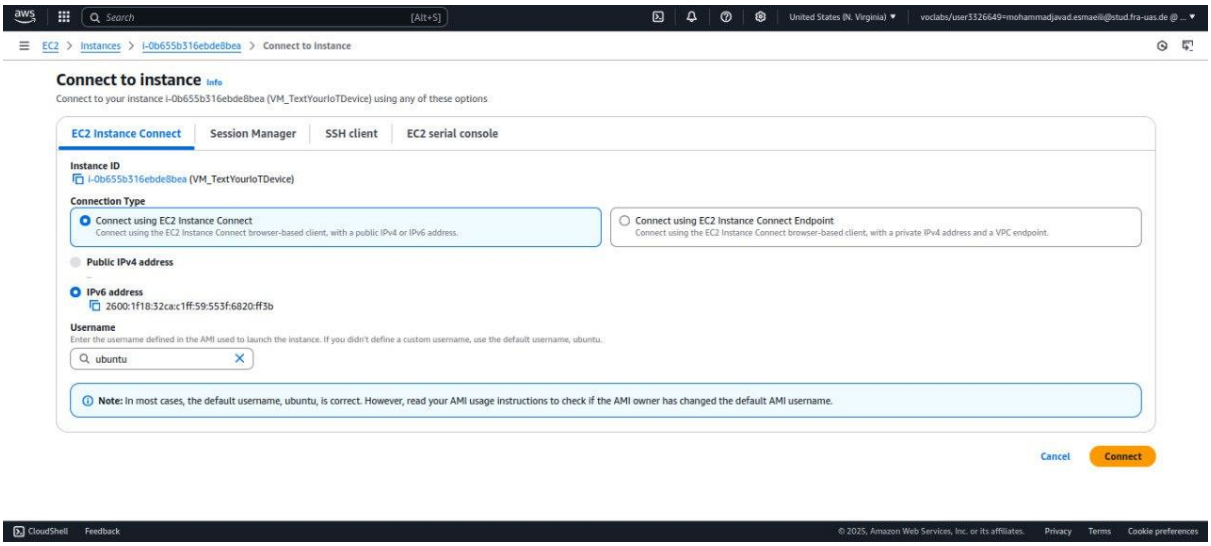

Figure 3.30: Connect to EC2 instance

Configurations of the MQTT Broker (Mosquitto) on the instance of AWS EC2

To install a *mosquitto* MQTT broker on the EC2 instance, the following commands must be executed:

sudo apt update

sudo apt install -y mosquitto

sudo apt install -y mosquitto-clients

The next step is to ensure that mosquitto is accessible as a services:

sudo systemctl enable mosquitto

Next, the three certificates for authentication and encryption generated in AWS IoT Core must be transferred to EC2. For this purpose, *FileZilla Client* [19] is used as SSH client. The following steps will show the instructions for the transfer:

1. After opening FileZilla, a connection must be established:
   - Protocol: SFTP – SSH File Transfer Protocol
   - Server: The public IPv4 or IPv6 address of your EC2 instance

- Port: 22
- Username: ubuntu (or another depending on the AMI you are using)
- Key file: Choose the *".pem"* key file for access.

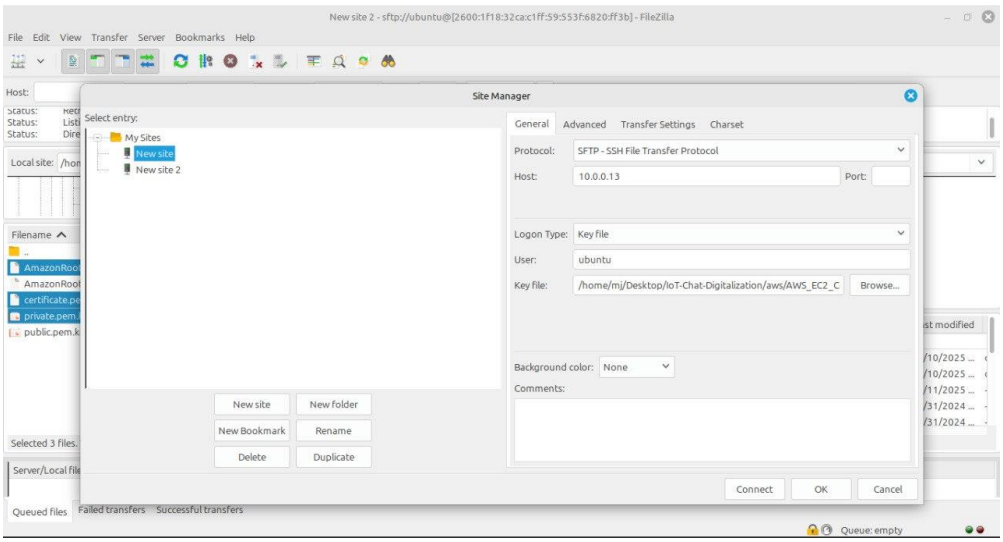

Figure 3.31: FileZilla settings

2. After the connection, you will see the file structure of the EC2 instance in the right window. In the left window, you will see the local files on your computer. Look for the certificate files on the left side (local computer):
    - certificate.pem.crt
    - private.pem.key
    - AmazonRootCA1.pem

Because you only have access to the *"ubuntu"* directory with an SSH connection at this point, the files can only be uploaded there. Drag and drop these files to the right side (server) in the user directory (ubuntu).

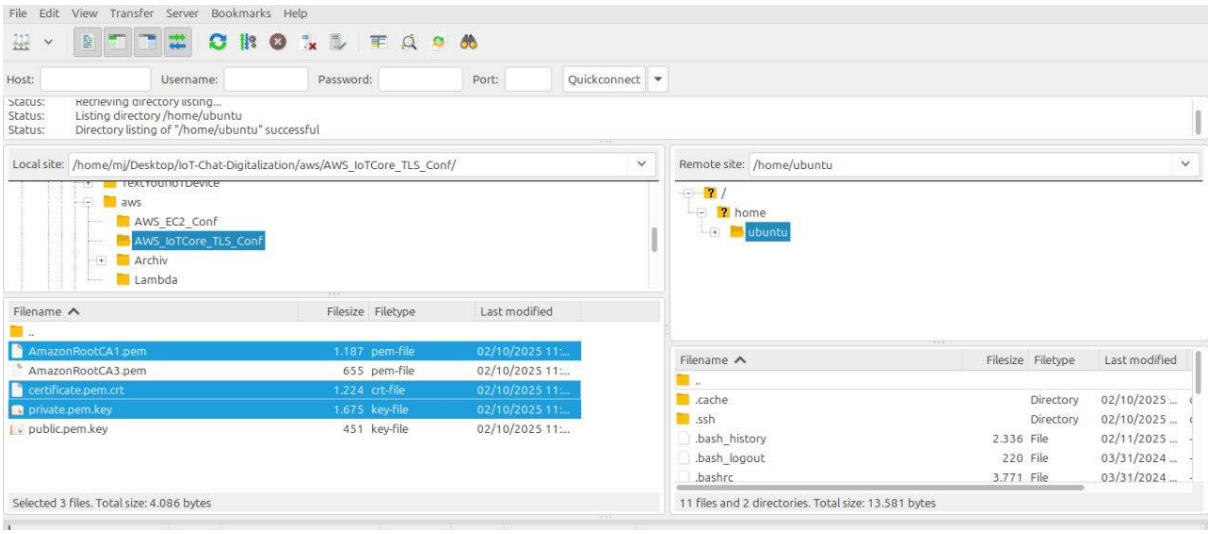

Figure 3.32: Uploading files in server

3. As soon as the upload is complete, you will see the files in the server directory.

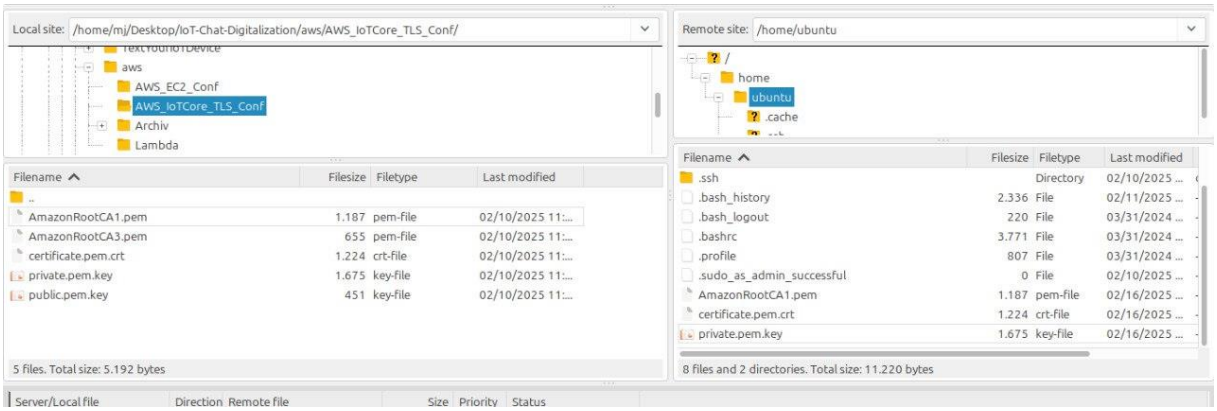

Figure 3.33: Uploaded files in the server

For the next step, the files should be transferred to the corresponding directories in mosquitto through the 3 commands:

```
ubuntu@ip-10-0-0-13:~$ sudo mv AmazonRootCA1.pem /etc/mosquitto/ca_certificates/
ubuntu@ip-10-0-0-13:~$ sudo mv certificate.pem.crt /etc/mosquitto/certs/
ubuntu@ip-10-0-0-13:~$ sudo mv private.pem.key /etc/mosquitto/certs/
ubuntu@ip-10-0-0-13:~$
```

Figure 3.34: Transfer files in mosquitto directories

With the command we can open and edit mosquitto.conf:

sudo vim /etc/mosquitto/mosquitto.conf

The next step is to configure mosquitto. *mosquitto.conf* configures mosquitto as an MQTT broker and enables the bridging feature to transfer messages between local MQTT devices and AWS IoT Core.

- *Listener 8080*: Used to receive MQTT messages over the port.
- *Listener 8883*: This is the standardized MQTT port for secure connections (MQTTS). It uses SSL/TLS encryption to transfer data securely.
- *allow_anonymous true*: This setting allows all connections without authentication.
- *persistence true*: Enables the storage of MQTT messages so that they are not lost after a broker restart.
- *persistence_location /var/lib/mosquitto/*: Specifies the storage location for persistent MQTT data.
- *log_dest file /var/log/mosquitto/mosquitto.log*: saves log data to a file. This Path contains connection events, errors and MQTT messages.
- *include_dir /etc/mosquitto/conf.d*: specifies that Mosquitto reads additional configuration files from the /etc/mosquitto/conf.d/ directory.
- *connection awsiot*: Definition of an MQTT bridge named awsiot.
- *address a1wr6xs1y6lp2l-ats.iot.us-east-1.amazonaws.com:8883*: specifies the address of the AWS IoT Core MQTT broker. Port 8883 is the standard MQTTS port for encrypted communication.
- MQTT topics determine what data is exchanged between the local Mosquitto broker and AWS IoT Core.
  - *topic awsiot_to_localgateway/# both 1*: allows subscribing to all messages from AWS IoT Core to the local gateway. both 1 means messages are synchronized in both directions.
  - *topic localgateway_to_awsiot/# both 1*: Allows publishing messages from the local broker to AWS IoT Core. *both 1* means messages are synchronized in both directions.
- *bridge_protocol_version mqttv311*: Uses MQTT v3.1.1, which is more stable than v3.1.
- *bridge_insecure false*: Enforces TLS encryption for secure communication.
- *cleansession true*: Clears saved MQTT sessions on each connection.
- *clientid bridgeawsiot*: The broker uses this ID when connecting to AWS IoT Core.

- *start_type automatic*: MQTT bridge starts automatically with Mosquitto.
- *notifications false*: Disables MQTT notifications for the bridge.
- *log_type all*: Enables comprehensive logs for troubleshooting and debugging.
- AWS IoT Core uses X.509 certificates for authentication and encryption. These must be configured here.
    - *bridge_cafile /etc/mosquitto/ca_certificates/AmazonRootCA1.pem*: Contains the AWS root certificate (AmazonRootCA1.pem).
    - *bridge_certfile /etc/mosquitto/certs/certificate.pem.crt*: Contains the client certificate (certificate.pem.crt) to authenticate to AWS IoT Core.
    - *bridge_keyfile /etc/mosquitto/certs/private.pem.key*: The private key (private.pem.key) associated with the certificate.

```
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

#pid_file /run/mosquitto/mosquitto.pid
listener 8080
listener 8883
allow_anonymous true


persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d


# =================================================================
# Bridge to AWS IOT
# =================================================================

connection awsiot

#<Paste your AWS IoT Core ATS endpoint retrieved from the AWS CLI in the form of xxxxxxxxxxxxxx-ats.iot.<region>.amazonaws.com:8883
# address xxxxxxxxxxxxxx-ats.iot.<region>.amazonaws.com:8883
address a1wr6xs1y6lp2l-ats.iot.us-east-1.amazonaws.com:8883

# Specifying which topics are bridged and in what fashion
topic awsiot_to_localgateway/# both 1
topic localgateway_to_awsiot/# both 1
# topic both_directions both 1

# Setting protocol version explicitly
bridge_protocol_version mqttv311
bridge_insecure false

# Bridge connection name and MQTT client Id, enabling the connection automatically when the broker starts.
cleansession true
clientid bridgeawsiot
start_type automatic
notifications false
log_type all

# =================================================================
# Certificate based SSL/TLS support
# =================================================================

#Path to the rootCA
bridge_cafile /etc/mosquitto/ca_certificates/AmazonRootCA1.pem

# Path to the PEM encoded client certificate
bridge_certfile /etc/mosquitto/certs/certificate.pem.crt

# Path to the PEM encoded client private key
bridge_keyfile /etc/mosquitto/certs/private.pem.key

#END of bridge.conf
```

Figure 3.35: mosquitto.conf

Run the MQTT Broker (Mosquitto) on AWS EC2

To start the MQTT broker mosquitto on the AWS EC2 instance, first run the following command:

sudo systemctl start mosquito

This command starts the mosquitto service so that it can process incoming and outgoing MQTT messages. To check if the service started successfully, then query the status with:

sudo systemctl status mosquito

This command displays information about the current state of the service, including its status (active or inactive) and any errors. If the service is not running, you can view and resolve any error messages here.

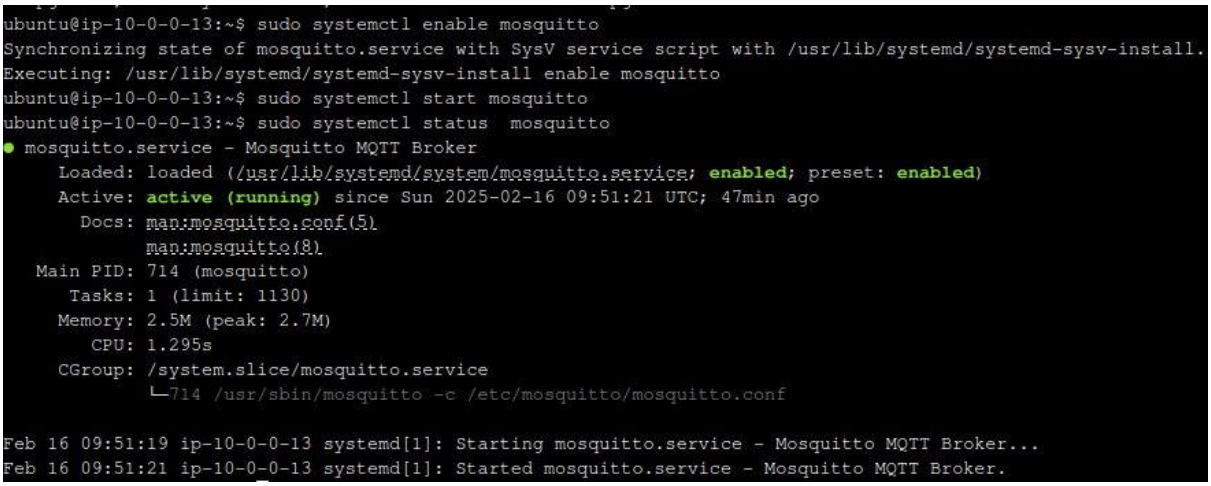

Figure 3.36: start with mosquitto

If changes have been made to the mosquitto configuration or the service needs to be restarted for any other reason, run the following command:

sudo systemctl restart mosquito

Test the MQTT Broker (Mosquitto) on AWS EC2

To test mosquitto you can use mosquitto_sub and mosquitto_pub:

- *mosquitto_sub*:

mosquitto_sub -h <broker-ip> -p 8883

--cafile /etc/mosquitto/ca_certificates/AmazonRootCA.pem

--cert /etc/mosquitto/certs/certificate.pem.crt

--key /etc/mosquitto/certs/private.pem.key

-t "<topic>" -d

The *mosquitto_sub* command is used to connect to an MQTT broker and subscribe to messages from a specific topic. The *-h* option specifies the IP address or hostname of the broker, while *-p 8883* specifies the port for communication. Since port 8883 is used here, the connection is made via SSL/TLS to ensure secure data transmission.

To ensure the trustworthiness of the broker, *--cafile* specifies the root certificate of the certificate authority. Additionally, *--cert* and *--key* are required to authenticate the client with a certificate and the associated private key, which is especially important for cloud services such as AWS IoT Core.

The *-t* option defines the topic to which the message is sent. Finally, the -d option activates debug mode, which outputs detailed information about the connection setup and data transmission.

Figure 3.37: mosquitto_sub

- *mosquitto_pub*

***mosquitto_pub -h <broker-ip> -p 8883 ***

***--cafile /etc/mosquitto/ca_certificates/AmazonRootCA.pem ***

***--cert /etc/mosquitto/certs/certificate.pem.crt ***

***--key /etc/mosquitto/certs/private.pem.key ***

-t "<topic>" -m "<Message>" -d

The *mosquitto_pub* command is used to send a message to an MQTT broker. -m is used to transmit the actual message conte.

Chapter 4

Results, Conclusion and Future Work

Results

The IoT system developed in this project successfully integrates a Telegram bot with an IoT device to retrieve and display environmental data. The system allows users to request humidity, temperature, and pressure data from the IoT device through a simple and intuitive interface.

- At the beginning, the Telegram bot presents three buttons to the user: Humidity, Temperature, and Pressure.
- If the user selects Humidity, the bot directly retrieves the humidity value from the IoT device and displays it in the chat.
- If the user selects Temperature, two additional buttons appear, allowing the user to choose between Celsius or Fahrenheit. After selecting the unit, the bot fetches the temperature value from the IoT device.
- If the user selects Pressure, the bot provides options to choose between Bar or Pascal, and after the user's selection, the bot retrieves the pressure value.

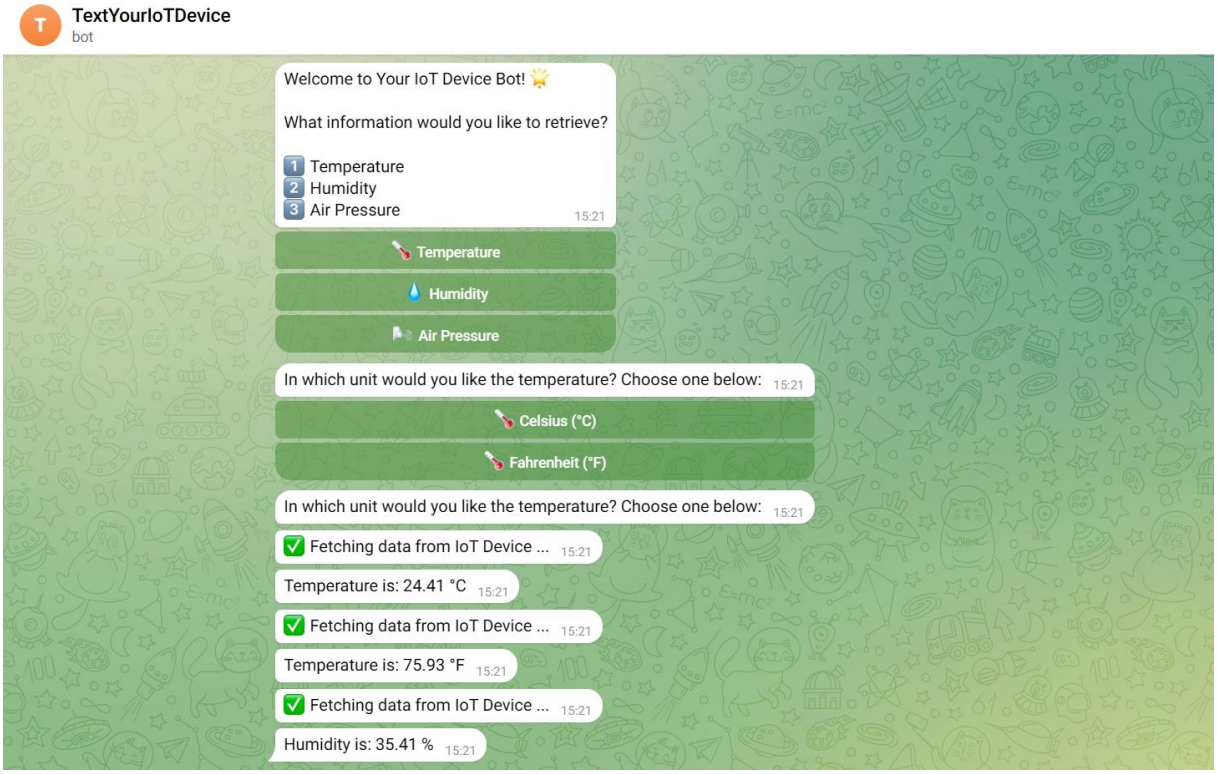

Figure 4.1: Screenshot from the Telegram bot and communication

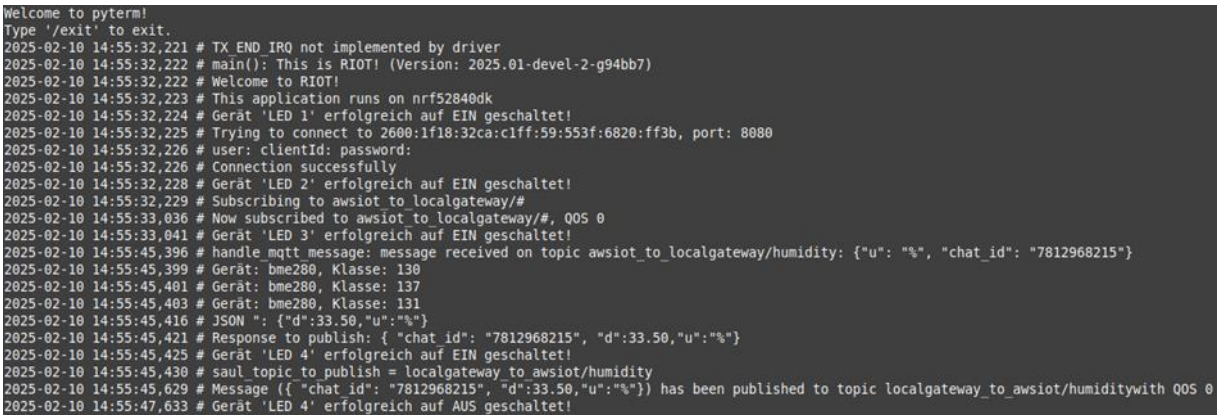

Figure 4.2: Result of humidity request in terminal

Conclusion

The project successfully demonstrates a seamless interaction between an IoT device and a messaging platform. By integrating AWS IoT Core, MQTT, and Telegram, real-time environmental data retrieval was made accessible through an easy-to-use chat interface. The system provides a user-friendly way to interact with IoT devices without the need for specialized applications.

Future work

While the system is functional, certain areas can be improved:

- Security Enhancement: The communication between the IoT device and the first broker is currently not secure. Implementing TLS encryption for this communication step would improve security and protect data integrity.
- Scalability: Extending the system to support multiple IoT devices and different types of sensors could enhance functionality.
- User Interface Improvements: Adding visualization features, such as charts for historical data, could make the bot more informative.
- Integration with Additional Cloud Services: Expanding the system to store and analyze historical data in cloud-based databases for trend analysis.

By addressing these areas, the system could evolve into a more robust and secure IoT monitoring solution.

Chapter 5

Appendices

Der gesamte Quellcode für das Projekt ist im GitHub-Repository unter https://github.com/Hani-Rezaei/IoT-Chat-Digitalization

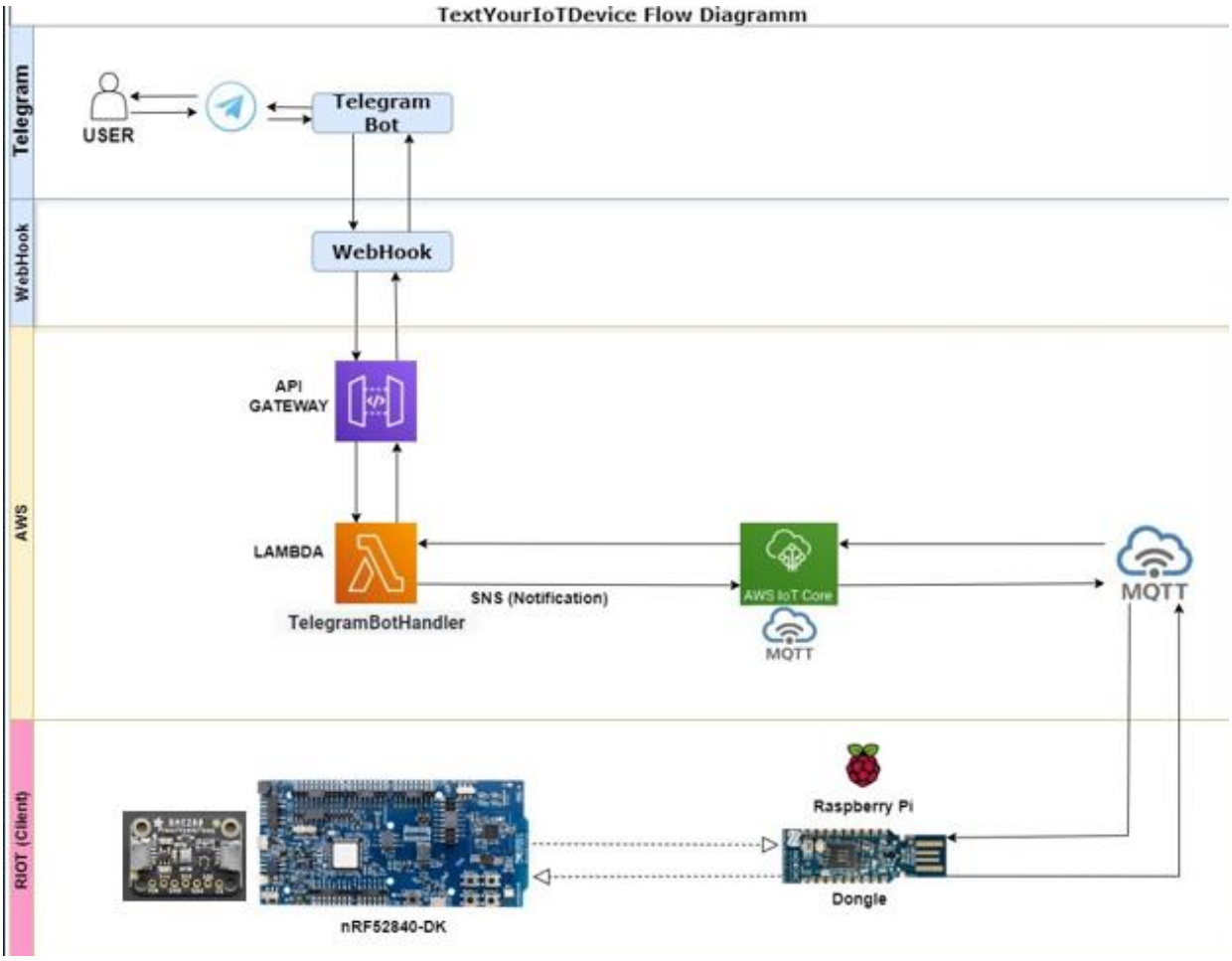

Figure 5.1: TextYourIoTDevice-Flow-Diagramm

Bibliography

[1] S. Pandey, "BrightWork," 19 06 2024. [Online]. Available: https://www.brightwork.com/microsoft-teams-project-management.

[2] Telegram, "Telegram Bot API Documentation," [Online]. Available: https://core.telegram.org/bots/api.

[3] Webhooks, "Wikipedia," [Online]. Available: https://de.wikipedia.org/wiki/Webhooks.

[4] Postman, "Postman," API platform, 2012. [Online]. Available: https://www.postman.com/product/what-is-postman/. [Accessed 29 12 2024].

[5] AWS, "AWS Cloud Services," [Online]. Available: https://aws.amazon.com/de/.

[6] AWS, "AWS API Gateway," [Online]. Available: https://aws.amazon.com/api-gateway/.

[7] AWS, "AWS Lambda," [Online]. Available: https://aws.amazon.com/de/lambda/.

[8] AWS, "AWS iot-core," [Online]. Available: https://aws.amazon.com/iot-core/.

[9] "AWS Things," [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/thing-types.html.

[10] AWS, "AWS CloudWatch," [Online].

[11] Dongle, [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle.

[12] Dongle, [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop/Download?lang=en#infotabs.

[13] Nordic, [Online]. Available: https://de.farnell.com/nordic-semiconductor/nrf52840-dk/dev-kit-bluetooth-low-energy-soc/dp/2842321.

[14] Nordic, [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK.

[15] Raspberry Picture, [Online]. Available: https://www.reichelt.de/de/de/shop/produkt/gehaeuse_fuer_raspberry_pi_4_schwarz_grau-260039?PROVID=2788&gad_source=1&gclid=CjwKCAiA2cu9BhBhEiwAft6IxF1eEhKoY78oc0RY3z4DExEHWi-MPvuiwZtNVHZ-UpB1LmEKwNCc0xoCdbUQAvD_BwE.

[16] BME280 Picture, [Online]. Available: https://www.adafruit.com/product/2652.

[17] BME280 Documentation, [Online]. Available: (https://cdn-shop.adafruit.com/product-files/2652/2652.pdf.

[18] RIOT, [Online]. Available: (https://doc.riot-os.org/group__drivers__bmx280.html.

[19] FileZilla, [Online]. Available: https://filezilla-project.org/download.php?type=server.

[20] Dongle, [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle.