CasaGrama - The Telegram Smart Home Bot

Raphael Heer, Maurin Donat Thalmann
HSLU, Informatik Department, Switzerland
raphael.heer@hslu.ch, maurindonat.thalmann@stud.hslu.ch

June 15, 2019

Contents

1 Abstract

The Internet of Things plays a huge role in the market for Smart Home devices. Nowadays and in the future, people want their homes to behave as intelligent as possible, whenever and wherever they are. But while it's an expensive task to fully automate your home, there's smaller and cheaper solutions to partially make your house smarter, so everyone should be able to "smart up" his house.
The task of this paper is to develop a Telegram Bot, running on a dedicated server or cloud solution, which is able to control a number of IoT Smart Home devices. Through the Telegram messenger application, one should be able to add, configure and control a handful of IoT devices. So, for example, an LED or door lock can be (de-)activated or data sent from a sensor can be requested through the bot interface inside of the Telegram application. Users should also be able to subscribe to specific devices, so the bot can notify them on a specified event.

Telegram Telegram is a cloud-based instant messaging service, available on a wide range of platforms as Android, iOS, Windows, etc. Besides communication between real-life persons, it supports the ability to communicate with Telegram bots. These are running on dedicated servers and usually offer a range of commands to perform certain tasks within or outside of the Telegram application.

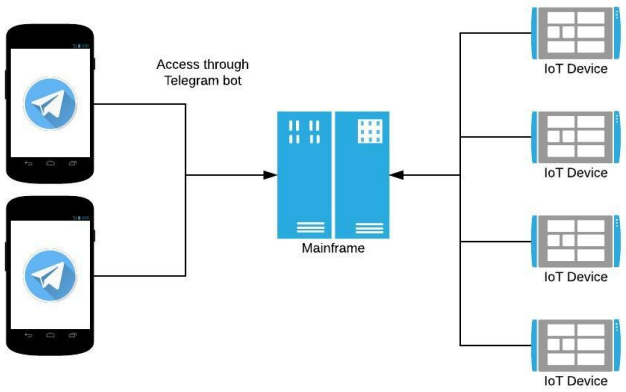

Figure 1: Project Infrastructure Model

2 Introduction

During this project, two main targets should be fulfilled for an ideal user experience.
First target is to think of an intuitive way in which the user interacts with the bot. How do the commands of the bot need to be set up, so that the user easily understands how he's able to control or read a certain device. The second target is the functional range of the bot itself, which functions need to be covered and which ones can be marked as optional. So in the following, the bot should be able to:

- Select a certain device from the range of connected devices
- Notify the user about certain device's events
- Perform a task on the selected device
  - Read data from a device
  - Change the state of a device (ON/OFF)
  - Change the values of a device
- Add / remove devices from the bot interface *(optional)*

We consider the project successful if we're able to access different types of devices or sensors through the bot interface. The bot itself runs on an Azure cloud instance or at least uses their services. The first version does not need to be capable of adding or removing other devices than the ones used during this project.

3 Related Work / Research

Throughout our research, we sought projects that would help us in the development of our Telegram bot and IoT devices we would want to be able to connect to. Although we found several projects, most of them were in a foreign language we don't understand or just slightly helpful. For example, there are several smaller DIY projects on the internet, for example:

How to Set Up Home Automation Through the Telegram Messenger App
https://blog.hackster.io/how-to-set-up-home-automation-through-the-telegram-messenger-app-8551d6f493a1
Home Automation With Telegram Bot
http://www.lucadentella.it/en/2015/10/13/automazione-con-bot-telegram/
Telegram bot for chatting with arduino - Smart Home
https://www.youtube.com/watch?v=UrwlOUQ1JMc

These are simple examples of Telegram bots to control a light chain or to read data from sensors. Unfortunately, they aren't very well documented and are very basic in functionality. But only the fact to see something working, similiar to what we want to achieve, was kind of impressive. The last link leads to a YouTube video of a Telegram bot, receiving input from a button and controlling a LED light, which is apparently connected through Azure (as far as the description tells us). Sadly this is even less documented that the other examples, therefore it wasn't of help for our project as well.
For the development of the Telegram bot, we've consulted the official Telegram bot API and several tutorials to create first Telegram bots. While working through some of these tutorials, we've discovered a git repository for an (apparently) well-known wrapper for the Telegram bot API:

GitHub : python-telegram-bot
https://github.com/python-telegram-bot/python-telegram-bot

The *python-telegram-bot* is a package that wraps the Telegram bot API into easily understandable classes. It has a huge community, a well-documented API and a wiki with tutorials, code snippets and also bot examples, created by other users. We've also found other wrappers like this, but in the end we decided to use this one for its simple usage and big community.
The documentation for the Azure Python SDK contains a lot of tutorials, well-documented APIs as well as examples of real-life use cases, from which we were able to derivate several code snippets into our own project.

4 System Design and Implementation

As cloud infrastructure platform, we chose Microsoft Azure services, for which we were granted access by the HSLU. The main application of the Telegram Bot is written in Python. The IoT devices are connected to the Azure Cloud through WLAN.

4.1 System Overview

A diagram of the whole system overview can be found on the next page.
The system of the prototype is divided into multiple sections, which will be described in the following:

- **User Side:** An end user is able to access the Telegram application on a device (smartphone, laptop etc.). Through this application, he can start a conversation with our Telegram Bot to access its features.
- **Telegram Servers:** Every action performed in the Telegram application is processed by the Telegram servers. So every request by a user to our Bot is first received by the Telegram servers and processed through to our Bot application.
- **Telegram Bot**: This is the main entry point for user requests. It is a Python script running on a dedicated server or computer. The user invokes a command over Telegram which is finally received by the Telegram bot. In here, the request is forwarded to a subclass of our Bot, which then communicates with the Azure Cloud. This defines the Telegram Bot as the main node between the Telegram servers and the Azure Cloud. There is and must only run one Telegram Bot instance at a time. For this reason, the Python script must include a unique bot token, generated by the BotFather Bot on the Telegram servers.
- **Microsoft Azure**: In contrary to the other applications, Azure is less about programming and more about configuration. Therefore, the required services on the Azure Cloud must be configured for our exact needs. To be able to write data into the Cosmos DB, a Stream Analytics service must be running which listens to data sent by an IoT device in the IoT Hub. Every IoT device needs to be configured as such in an IoT Hub instance.
- **IoT Device Applications**: The last section are the IoT devices and their applications. Their task is to interact with their distinct peripherals and communicate with the Azure Cloud, more precisely to send and receive data.

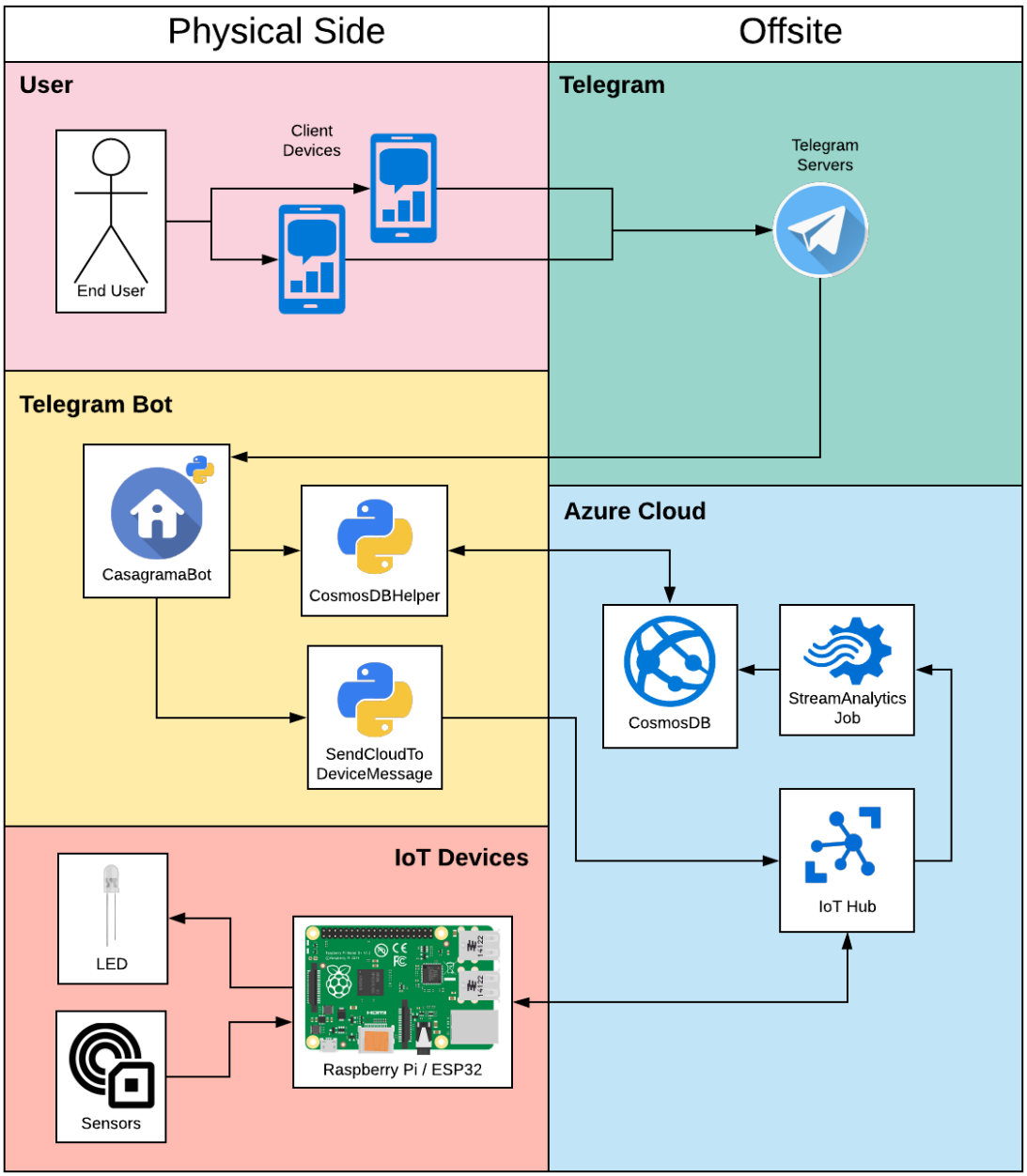

Figure 2: Overview of the whole prototype system

4.2 System Architecture

4.2.1 Telegram Bot

The Telegram Bot is actually divided into a main application and two subclasses. Logging has been implemented in all of them to guarantee the right functionality.

The **CasaGramaBot** serves as the main application of the Bot. It contains the bot token to access the Telegram Bot API and the necessary attributes to access the Cosmos DB. If an user enters a command on his client, it will be handled and forwarded to the appropriate function.

The **CosmosDBHelper** is called in the main application to access the latest entry of the Cosmos DB we used during this project. It receives the URL and access key it needs to access the Cosmos DB from the main application. It simply reads the database and the latest entry of its table and is able to return the required attributes.

The **SendCloudToDeviceMessage** is called in the main application as well. It contains the connection string to the IoT Hub itself and the device ID it wants to communicate with. Its sole purpose for this project is to send a message to the Raspberry Pi, through which the Raspberry Pi knows what to do with the LED. When it is called in the main application, it receives the state that the LED should be in as the `command` parameter.

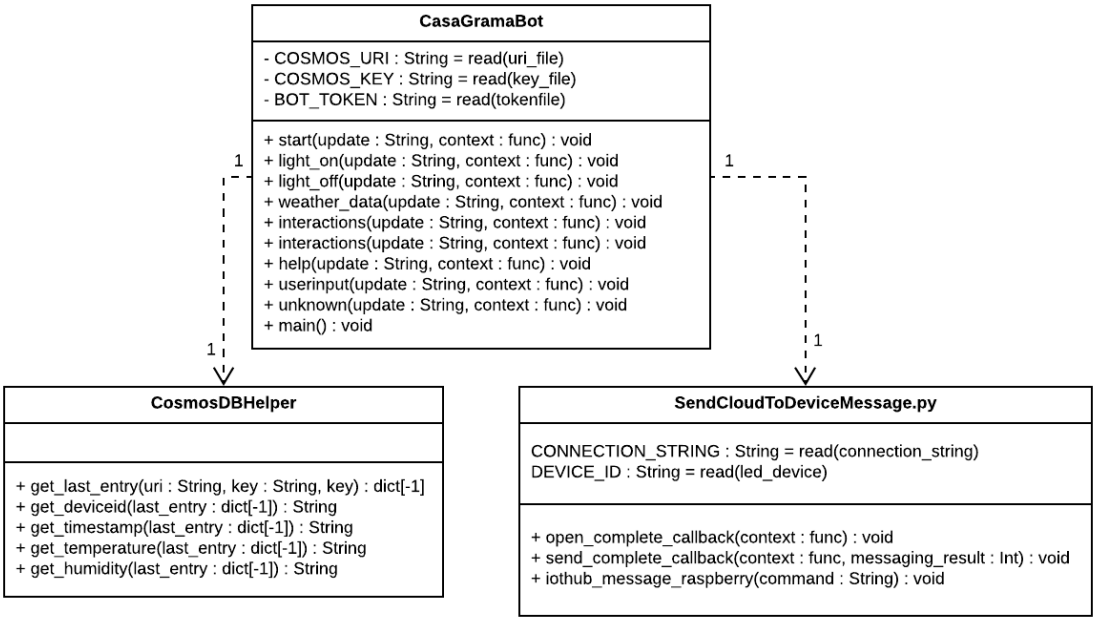

Figure 3: Class diagram of the Telegram Bot application

4.2.2 IoT Devices

Theoretically, there could be a large amount of IoT Devices which are running with different applications and communicating with the cloud. This is what the Azure IoT Hub was made for. However, this would require some prior knowledge about the different technologies on the devices and in the cloud. Due to this reason, the project uses only two types of applications which are implemented on the Raspberry Pi.

The applications were divided into two categories. Both categories are implemented to be extendable with other use cases. They are described in the following paragraphs:

Send Data to the Azure IoT Hub with a BME280 The goal of the first use case is to send data continuously to the Azure IoT Hub. In the cloud, the data will be processed with Stream Analytics and afterwards saved to Cosmos DB. The data flow in this application is just in one direction from the Raspberry Pi to the cloud. In this project, the BME280 sensor was used. It measures the temperature, the humidity and the air pressure. For other projects, the BME280 sensor can be exchanged with any other sensor that collects data from its environment.

Following class diagram shows the class structure of the application:

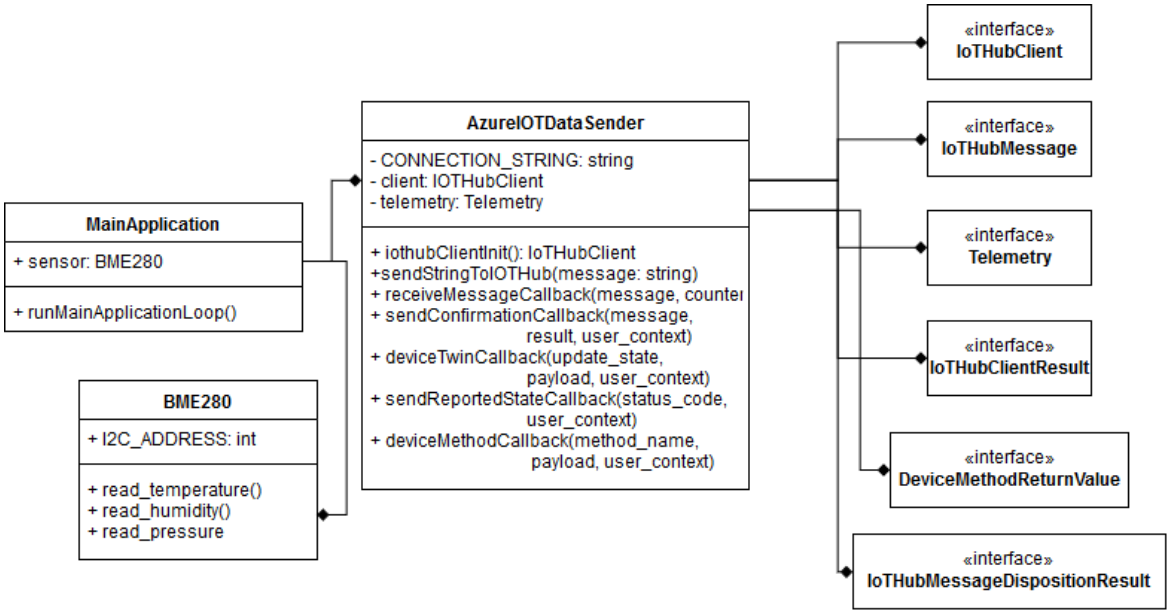

Figure 4: Class diagram for the weather data application

Listen to commands from the Azure IoT Hub and turn the lights on and off The goal of the second application is to receive commands from the cloud and perform a pre defined action. The action is issued from the user over the Telegram Bot. The telegram bot sends a message over the Azure IoT Hub to the device. The device, which is listening to the Azure IoT Hub, receives the message. In this project, an LED can be turned off and on by over the Telegram Bot. For other projects, it could be possible to open the window or inform the user, that the clothes in the washing machine are ready to be taken out.

Following class diagram shows the class structure of the application:

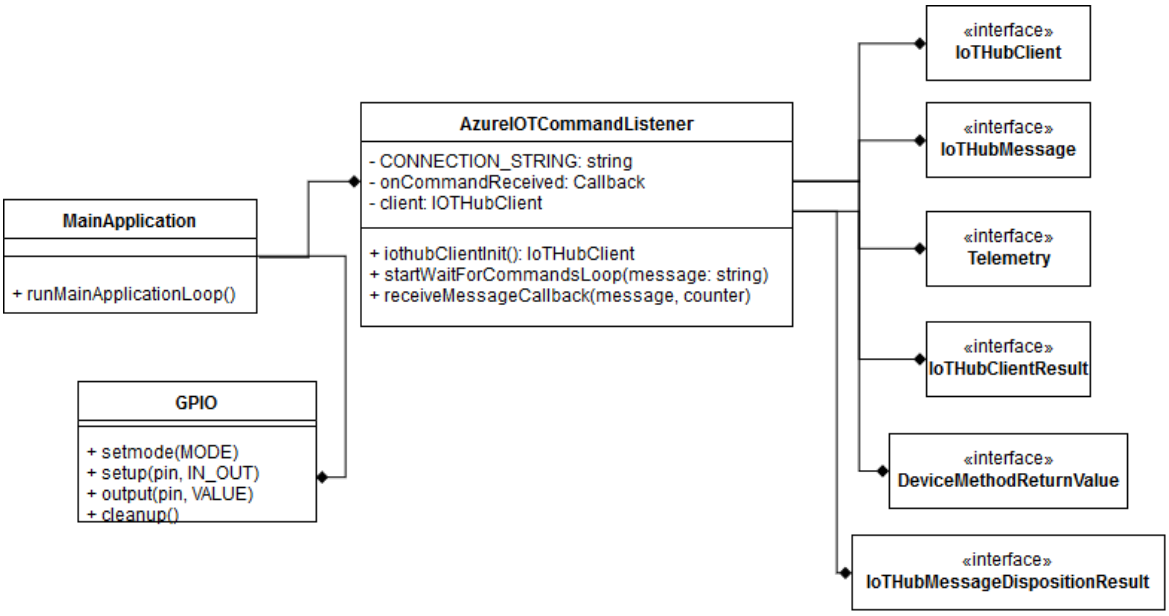

Figure 5: Class diagram for the LED application

4.3 Software Architecture Layers & Modules

4.3.1 Telegram Bot

Telegram Bot - CasaGramaBot.py	
Interface	**Description**
Provides	
`start(update, context)`	Returns a start message, invoked by the `\start` command.
`light_on(update, context)`	Turns the light on for an IoT device, invoked by the `\light_on` command.
`light_off(update, context)`	Turns the light off for an IoT device, invoked by the `\light_off` command.
`weather_data(update, context)`	Returns the most recent weather data from an IoT device, invoked by the `\weather_data` command.
`interactions(update, context)`	Returns a custom keyboard to select different interactions, invoked by the `\interactions` command.
`help(update, context)`	Returns a help message, invoked by the `\help` command.
`userinput(update, context)`	Logs text messages sent by a client.
`unknown(update, context)`	Returns error message for unknown commands, invoked by any unknown command.
`main()`	Starts the Telegram Updater and binds all commands to handlers.
Interface	**Description**
Uses	
`telegram.ext.Updater()`	Enables retrieving of updates with the Telegram Bot API, needs to be given a Bot token.
`*.startPolling()`	Starts communication and starts listening for actions.
`telegram.ext.CommandHandler()`	Handles commands received by the bot from a client, needs to be given a command name and the action to be fulfilled.
`telegram.ext.MessageHandler()`	Handles messages received by the bot, needs to be given a message filter and the action to be fulfilled.
`telegram.ext.Filters()`	Filter incoming messages by content or type (e.g. messages, commands)
`telegram`	
`*.KeyboardButton()`	Defines a button for a custom keyboard.
`*.ReplyKeyboardMarkup()`	Enables a custom keyboard in a Telegram client.
`logging.getLogger()`	Activates Logging for this script.
`logging.basicConfig()`	Configures logging aspects, like formatting, log level etc.
`logging.FileHandler()`	Enables logging to a file in a given directory.
`logging.StreamHandler()`	Enables logging to the console.

Table 1: Interfaces used from Libraries and Methods used in CasaGramaBot

CasaGramaBot The `CasaGramaBot.py` is the main application which needs to be executed in order to run the Telegram bot. It uses the interfaces from *python-telegram-bot* to access the Telegram Bot API in turn to communicate with a Telegram client.

Telegram Bot - CosmosDBHelper.py	
Interface	**Description**
Provides	
`get_last_entry(uri, key)`	Gets the latest (most recent) database entry, needs to be given the database's URL and the access key.
`get_temperature(last_entry)`	Get the temperature value from the latest entry, needs to be given the latest entry.
`get_humidity(last_entry)`	Get the humidity value from the latest entry, needs to be given the latest entry.
Interface	**Description**
Uses	
`document_client.DocumentClient()`	Creates a connection to a database, needs to be given the database URL and the access key.
`.QueryDatabases()`	Reads databases from a database connection, needs to be given a query.
`.QueryCollections()`	Reads tables or collections from a database, needs to be given a query.
`.ReadDocuments()`	Reads documents in a table or collection, needs to be given a query.
`logging.getLogger()`	Activates Logging for this script.
`logging.basicConfig()`	Configures logging aspects, like formatting, log level etc.
`logging.FileHandler()`	Enables logging to a file in a given directory.
`logging.StreamHandler()`	Enables logging to the console.

Table 2: Interfaces used from Libraries and Methods used in CosmosDBHelper

CosmosDBHelper The `CosmosDBHelper.py` serves as a connection helper to our Azure Cosmos DB to receive the latest weather data. It uses the interface *document_client* from the Python package *pydocumentdb*, which allows to query entries from a database.
It is used by the main application to

Telegram Bot - SendCloudToDeviceMessage.py	
Interface	**Description**
Provides	
`open_complete_callback(context)`	Logs the callback for opened communication
`close_complete_callback(context)`	Logs the callback for closed communication, needs to be given messaging result.
`iothub_message_raspberry(command)`	Sends a message to the predefined device in the IoT Hub, needs to be given a message to be sent.
Interface	**Description**
Uses	
`iothub_service_client`	Package for the IoT Hub Message sender / service
`.IoTHubMessaging()`	Enables communication with an IoT Hub instance, needs to be given the connection string to the IoT Hub.
`.open()`	Opens communication with the IoT Hub, needs to be given a callback and context.
`.sendasync()`	Sends a cloud-to-device message in the IoT Hub, needs to be given a device name, message, callback and message index.
`.close()`	Ends communication with the IoT Hub.
`.IoTHubMessage()`	Object which can be delivered to a target device in the IoT Hub, needs to be given message content.
`.IoTHubError()`	An exception which can be thrown when there are problems with the connection to the IoT Hub
`logging.getLogger()`	Activates Logging for this script.
`logging.basicConfig()`	Configures logging aspects, like formatting, log level etc.
`logging.FileHandler()`	Enables logging to a file in a given directory.
`logging.StreamHandler()`	Enables logging to the console.

Table 3: Interfaces used from Libraries and Methods used in SendCloudToDeviceMessage

SendCloudToDeviceMessage The `SendCloudToDeviceMessage.py` makes Cloud-to-device messaging possible for our Bot. It utilizes the Python package *azure-iothub-service-client*, which serves as a handler for cloud-to-device messages.

4.3.2 IoT Devices

There are two different kinds of applications for the IoT Devices. The first type continuously sends data to the Azure IoT Hub. It does not receive any input from the IoT Hub except for the confirmation messages. The second one listens to the Azure IoT Hub and waits for a command to execute some functionality like turning the lights on or off.

For this reasons, there are two types of wrapper classes, which wraps exactly this functionality. They are called `AzureIOTDataSender` and `AzureIOTCommandListener`. The code for this classes is derived from an example application to communicate with Azure IoT Hub. Which methods they provide and which methods they use is described in the following two tables.

Raspberry Pi - AzureIOTDataSender	
Interface	**Description**
Provides	
`__init__(connectionString)`	Constructor for the class AzureIOTHubDataSender
`iothubClientInit()`	This method is used to create an instance of the class IoTHubClient. This object is used to send data to the IoT Hub.
`sendStringToIOTHub(messageString)`	Sends a string to the Azure IOT Hub.
`receiveMessageCallback(message, counter)`	Callback function. Is called if the IoT Hub sends a message to the device.
`sendConfirmationCallback( message, result, user_context)`	Callback function. Is called after the IoT Hub sends a confirmation that the message has been received.
`deviceTwinCallback(update_state, payload, user_context)`	Callback function. Called for Digital Twins functionallity
`sendReportedStateCallback( status_code, user_context)`	Callback function. Is called, if the reported state has been received by the IoT Hub
`deviceMethodCallback(method_name, payload, user_context)`	Callback function. Is called if the device receives a message
Interface	**Description**
Uses	
`IoTHubClient`	
`.send_event_async(message, confirmationCallback, messageCount)`	Send asynchronous message to the IoT Hub
`.send_reported_state( reported_state, str_length, reportedStateCallback, context)`	Report IoTHubClient state to Azure
`Telemetry`	
`.send_telemetry_data( IoTHubName, str_length, event_success, message)`	Send telemetry data to Azure

Table 4: Interfaces used from the main application to communicate with the Azure IoT Hub

AzureIOTDataSender The `AzureIOTDataSender.py` is the part of the application, which communicates with the Azure IoT Hub. It is wrapper for all the Azure features like Telemetry and IoTHubClient.

Raspberry Pi - AzureIOTCommandListener	
Interface	**Description**
Provides	
`__init__(connectionString)`	Constructor for the class AzureIOTHubCommandListener
`iothubClientInit()`	This method is used to create an instance of the class IoTHubClient. This object is used to send data to the IoT Hub.
`startWaitForCommandsLop( onCommandReceivedCallback)`	Starts the loop to wait for commands.
`receiveMessageCallback(message, counter)`	Callback function. Is called if the IoT Hub sends a message to the device.
Interface	**Description**
Uses	
`IoTHubClient .set_message_callback( callbackFunction, RECEIVE_CONTEXT)`	Sets the callback function which is called if the IoT Hub sends a message

Table 5: Interfaces used from the main application to receive messages from the Azure IoT Hub

AzureIOTCommandListener The `AzureIOTCommandListener.py` is the part of the application, which communicates with the Azure IoT Hub. It is wrapper for all the Azure features like IoTHubClient. It receives the IoTHubMessage, gets the actual message from it and calls a given callback.

4.4 System Implementation / Functional Software Architecture

4.4.1 Telegram Bot

For the Telegram bot to work, some requirements have to be met before running it.

- **Required packages**
  There are a couple of packages, as mentioned in previous chapters, which need to be installed for the Bot to work. We created a requirements file with the console application `pipreqs`, which outputs a requirements text file. Through `pip` (Python package manager) the requirements file can be handed over as a parameter and `pip` will install the required packages by itself. The required packages are:
  - `python-telegram-bot`
  - `pydocumentdb`
  - `telegram`
  - `azure-iothub-service-client`
    (On Windows, this package can easily be installed through `pip`. On Mac/Linux, you need to clone the Azure IoT Hub C SDK from Github and compile it yourself. A guide on how to do this can be found on the Azure IoT Hub Python SDK repository on Github)

- **Required values**
  There are some values which need to be created so the Bot knows how he is able to communicate with Azure. The required values have been outsourced from the script into the folder `properties` and are read during runtime. Following are the names of the files and what they should contain:

  - `TOKEN.txt`
    This file contains the Telegram bot token for your Bot. As mentioned, it needs to be generated by starting a chat with the *@BotFather* on Telegram and creating a new bot instance.
    It is accessed in the CasaGramaBot.py file.
  - `COSMOSDB_READKEY.txt`
    This file contains the first access key to your Cosmos DB, which can be read directly from the Cosmos DB instance in Azure.
    It is accessed in the CasaGramaBot.py file.
  - `COSMOSDB_URI.txt`
    This file contains the URL which allows a connection to your Cosmos DB. It can also be retrieved directly from your Cosmos DB instance.
    It is accessed in the CasaGramaBot.py file.
  - `CONNECT_STRING.txt`
    This file contains the connection string of the IoT Hub containing the device whose LED should be controlled. It can be accessed by navigating to the access control of the IoT Hub and getting the connection string of a IoT Hub owner.
    It is accessed in the SendCloudToDeviceMessage.py file
  - `LED_DEVICE.txt`
    This file contains the name of the device whose LED should be controlled. The device name can easily be found in the IoT Hub which contains it.
    It is accessed in the SendCloudToDeviceMessage.py file.

- **Running the bot**
  If all the requirements above are met, simply navigate to the `CasaGramaBot` directory and run the script with Python 3 as followed:
  (The project was tested with Python 3.7, we cannot guarantee it works under Python 2.7 due to incompatibility issues)

```
python3 CasaGramaBot.py
```

4.4.2 IoT Devices

Following is described how the applications are implemented and how to set them up for running.

BME280 application on the Raspberry Pi

- **Required packages for the BME280 application on the Raspberry Pi**
  There are two packages which are needed in order to run this application. The driver for the `BME280 sensor` can be installed over "`pip3 install adafruit-circuitpython-bme280`". The `azure-iothub-device-client` is not installable over `pip`. It needs to be pulled from the Github Repository of Microsoft and compiled by the user himself. Since the application only runs on the Raspberry Pi, the package is deployed with source code.
- **Wiring the Raspberry Pi with the BME 280** Wire the Raspberry Pi and the BME 280 Sensor like in the following picture:
  Here will be an image of the wiring

- **Required values**
  There are different values which can be set to configure the application. Some are required and some are optional with a default value. To change this values, open the `config.py` file in the `RASPBERRY_PYTHON_BME280` and set the values.
  - `CONNECTION_STRING`(**required**): Connection string for the device to connect to the Azure IoT Hub
  - `SLEEP_DELAY_IN_S`(optional): Time in seconds to wait between messages
  - `I2C_ADDRESS`(optional): Address of the `BME280 Sensor`

- **Running the application** If all the requirements above are met, simply navigate to the `RASPBERRY_PYTHON_BME280` directory and run the script with Python 3 as followed:
  `python3 BME280_Application.py`

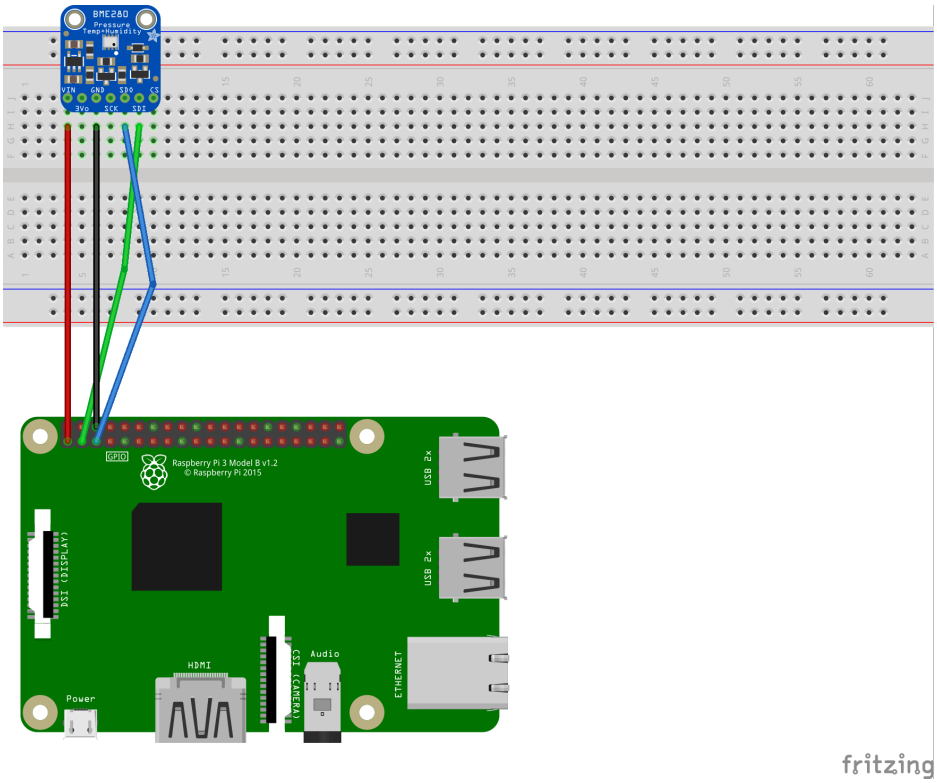

Figure 6: Wiring diagram for the BME280

LED application on the Raspberry Pi

- **Required packages**
  There are two packages which are needed in order to run this application. The first one is the library to handle the output over the GPIO. It can be installed with "`pip3 install RPi.GPIO`". The `azure-iothub-device-client` is not installable over `pip`. It needs to be pulled from the Github Repository of Microsoft and compiled by the user himself. Since the application only runs on the Raspberry Pi, the package is deployed with source code.

- **Wiring the Raspberry Pi with the LED**
  Wire the Raspberry Pi and the LED like in the following picture:

- **Required values**
  There are different values which can be set to configure the application. Some are required and some are optional with a default value. To change this values, open the `config.py` file in the `RASPBERRY_PYTHON_LED` and set the values.
  - `CONNECTION_STRING`(**required**): Connection string for the device to connect to the Azure IoT Hub
  - `GPIO_PIN`(optional): Address of the `BME280 Sensor`

- **Running the application** If all the requirements above are met, simply navigate to the `RASPBERRY_PYTHON_LED` directory and run the script with Python 3 as followed:
  `python3 LED_Application.py`

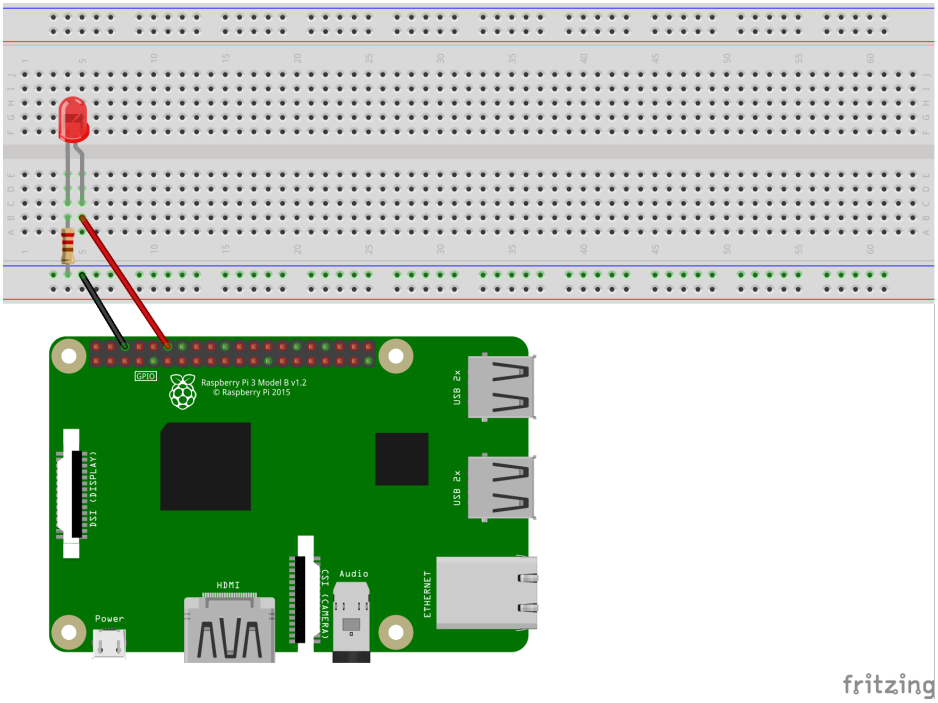

Figure 7: Wiring diagram for the Raspberry Pi with LED

5 Evaluation / Experiments / Results / Discussion

5.1 Assets Evaluation

As for our devices, we settled for a Raspberry Pi 3 Model B and an ESP32. We chose the Raspberry for its processing power and all the possibilities you get with it. The ESP32 was our choice for the Weather Data Collector for its minimal power consumption and size, which we would equip with a BME280 sensor. The main reason we chose these two devices was that they are both equipped with a WiFi-Module, which allows us to connect them directly to Azure. In the end we realized that we could simulate an ESP32 with a BME280 sensor on our Raspberry. This is why we ended up using only one device, the Raspberry, for our final prototype, which is able to do all necessary tasks.

5.2 Experiments

As for the start, we had to construct an overview of the whole system, because we had to think about how we wanted to solve the different connections to Azure, Bot- and device-sided. The first part was to get data from our IoT devices and how we wanted to deliver them to Azure, so the Bot could somehow access them. Our main focus was set to the weather data which we wanted to be able to read and the LED we wanted to control over Azure.

For the **Weather Data** we have learnt that you could easily access the Cosmos DB from outside, which is why we chose to deliver our data through a Stream Analytics service from our device to the Cosmos DB. We didn't run into a lot of problems with this, as we learnt the whole process in the AIOT module. The only difficulty was the way we could read this data with a Python script. After some attempts we learnt that the *pydocumentdb* package was a simple but very effective way to connect to a dedicated database and easily read data by queries. At first we couldn't get our script to read the latest, but only the first entry of the database. As we're both not very experienced in Python, we learnt about the different datatypes in Python. So by changing the read value from our data container datatype from 0 to -1, this problem was solved very quickly.

For the **LED Changing** mechanism, we first tried to write an Azure function which we could access over Python to send predefined values to the Raspberry to control our LED. We ended up struggling, as we couldn't develop an Azure function that was able to send the required data to the Raspberry to change the state of the LED. The best way to solve this was over the Cloud-to-Device Message, which at first we were only able to use through the Azure interface. Through an Azure tutorial for the IoT Hub Python SDK we learnt that you could also integrate this into a Python script, running on the receiving and the sending side. With some major adjustments, we were able to get this messaging mechanism working on both, the bot and the device side. We now had to integrate the switch point on the device, so it knows when to turn the LED on and off. After we did this, our LED changing was working as well.

For the **Bot Hosting** we thought first about hosting it on Azure itself. Sadly most of the references about hosting a Telegram bot on Azure were implemented on an outdated version of Azure itself. In the end, we were not able to create a service to host the Telegram Bot on Azure. Alternatively we tried hosting it on PythonAnywhere, which offers a free service to host Python scripts on their servers with limited data usage and processing power. However, we couldn't get it working in time due to the inability to get the package, which was needed for the Cloud-to-device messaging, working. Ultimately we have learnt that you could only install it through `pip` on Windows. On Linux and Mac, you would need to compile the IoT Hub C SDK on the service which executes the script, which we were not able to get working in the end. So for phase of this project, we decided to host our Bot on our personal computers themselves.

5.3 Results

As for the deadline of this project, our final prototype is able to do the following:

- Run the bot on a dedicated server or device with the required packages installed
- Read data from a sensor and forward it through Azure into a Cosmos DB (Device)
- Gather data from a Cosmos DB and return them to a client (Bot)
- Controlling an LED and offer access to it over the Azure IoT Hub (Device)
- Controlling an LED in an IoT Hub over a Python script (Bot)

6 Applications

Our final prototype is not very suitable for every environment, as we only collect data and control peripherals from preconfigured devices for this. In our vision, a more developed version of our Bot could be integrated in a household with many more devices. A user could either use DIY devices in its household and add them to the Bot environment or manage existing Smart Home devices from more common producers. To be able to integrate said devices, we would also need to expand our Bot with commands integrating the API of existing devices. Finally, our prototype delivers an unrefined projection of a very easy way to interact with a Smart Home and could become a mighty tool for end users with a technical aptitude.

7 Conclusion

We are content with the results we achieved for our final prototype. However, we would have loved to host our Bot on a dedicated server, we simply ran out of time in order to do this for our final prototype. The reason for this is mainly our focus on other aspects we wanted to finish for certain until the deadline. So in order to achieve this goal for another project, we should gather more information in the run-up to so we manage to do it in time.
We also think our functionality is not as perfected as we imagined it to be in the end. This happened mostly due to the complications with Azure or the Telegram API in the development process. But as stated, we are content with what our prototype is able to do at the time being. We could imagine that we would have been able to fit more functionality into this prototype, but maybe the final solution would not have been as refined as our final prototype.

8 Contributions / Acknowledgements

We would like to acknowledge each other, as a member of this team, for all the time we have put into this project. We could both learn from each other about software aspects new to each other, how to tinker with the hardware and how to elaborate a working solution. We are thankful for our team to have worked great as one unit and would like to acknowledge the support we were able to give each other. Also, a big acknowledgement goes to Dr. Angela Nicoara by expanding our knowledge in the world of IoT and the great support we were given by her.

9 Major Milestones & Deliverables

9.1 Team and Roles

In the following, all team members are listed, each with their respective roles:
Heer Raphael Sensors, Devices, Azure, Report
Thalmann Maurin Donat Telegram Bot, Azure, Report

9.2 Project Planning

9.2.1 Project Organization

The project was structured in an agile way. While we always worked together during the AIOT lessons, we managed to work on our individual parts outside and inside of school. Due to our workload from other modules, we couldn't both manage to achieve the same amount of work every week, but we could say the effort invested into this project was in a very good balance. In regular intervals, we kept each other up to date with the progress and met up to merge our progress.

9.2.2 Timelines

We followed the given timeline of the project and therefore the given deadlines:

28.02.2019 Choice of Team Members & Hands On Partner
14.03.2019 Final Project Proposal Submission
02.06.2019 Final Project Report & Code Submission

9.2.3 Milestones

In the following, these are the milestones we achieved:

14.03.2019 Final Project Proposal submitted
16.04.2019 Weather Data reading into Cosmos DB finished
01.05.2019 Telegram Bot base functionality finished
24.05.2019 LED controlling over Azure finished
01.06.2019 Main functionality of Bot tested and finished
02.06.2019 Final Report & Code submitted

9.2.4 Deliverables

At the end of the project, following deliverables are available:
- Code for the Telegram Bot and IoT Devices
- Azure instances on student's accounts
- Final Report of the project

10 References / Bibliography

How to Set Up Home Automation Through the Telegram Messenger App (22 February 2019).
Retrieved 28 May 2019, from
https://blog.hackster.io/how-to-set-up-home-automation-through-the-telegram-messenger-app-8551d6f493a1

Home Automation With Telegram Bot (13 October 2015).
Retrieved 28 May 2019, from
http://www.lucadentella.it/en/2015/10/13/automazione-con-bot-telegram/

Telegram bot for chatting with arduino - Smart Home (26 May 2015).
Retrieved 28 May 2019, from
https://www.youtube.com/watch?v=UrwlOUQ1JMc

GitHub : python-telegram-bot (n.d.).
Retrieved 28 May 2019, from
https://github.com/python-telegram-bot/python-telegram-bot

PythonAnywhere (n.d.).
Retrieved 2 June 2019, from
https://www.pythonanywhere.com/

Github : Azure IoT Hub SDK for Python (n.d.).
Retrieved 2 June 2019, from
https://github.com/Azure/azure-iot-sdk-python

Github : Azure IoT Hub SDK for C (n.d.).
Retrieved 2 June 2019, from
https://github.com/Azure/azure-iot-sdk-c

List of Figures

List of Tables