

Jesse Reinikka

Connecting TAMK Sensor Board to Amazon Web Services IoT

IoTti

Documentation

Spring 2018

SeAMK Technology

IoTti



SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Contents

Contents.....	2
1 Introduction	3
1.1 Devices	3
1.2 Amazon Web Services	4
1.3 Requirements.....	4
1.3.1 Equipment.....	4
1.3.2 Software.....	5
1.3.3 AWS account	5
2 AWS IoT	6
2.1 Creating a device and certificate	6
2.2 Creating a policy	15
2.3 API endpoint and monitoring device.....	19
3 Arduino Mega.....	22
4 Python program	26
5 The code.....	29
5.1 Arduino code	29
5.2 Python code	33
Appendix	36

1 Introduction

Connecting small devices, such as Arduino Uno, Nano or Mega to Amazon Web Services (AWS) is a bit trickier, thanks to its security. None of the mentioned boards currently have the crypto library or enough runtime memory to pass the authentication process. How do we do it then? By putting something in between. Arduino could send its data to a server or a gateway (e.g. Raspberry Pi) and then let it take care of the authentication and sending of data. In this document, however, the demo version is done using a computer.

Arduino will send the JSON data through serial port to a Python program that is used to handle and send it by using AWS API. Luckily, Amazon has made a Python SDK to further simplify the process of connecting to IoT service.

The program was made with simplicity and practicality in mind. All the code is available in the appendix of this document, but also on [GitHub](#).

1.1 Devices

The Sensor Board consists of **Arduino Mega 2560** that is connected to **Arduino Ethernet Shield V1**, **Dallas temperature sensor** and **LCD screen**. Ethernet shield won't be used in this documentation, as there is no use for it. However, it could be used to connect to a socket, instead of using serial connection to send data.

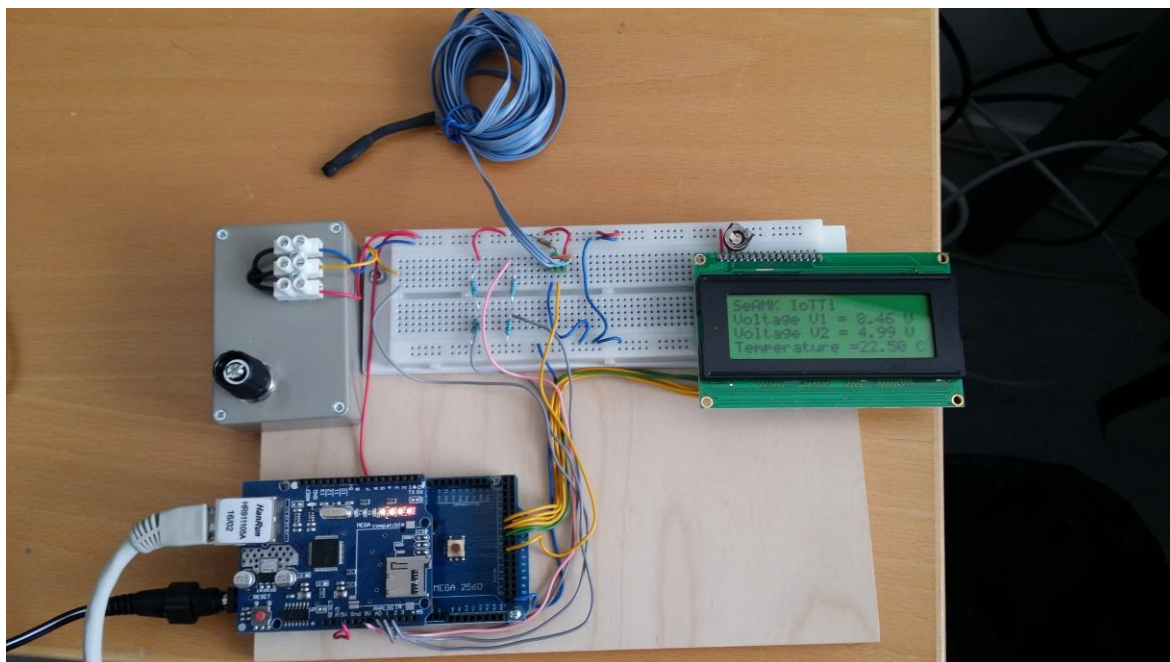


Figure 1. TAMK Sensor Board.

1.2 Amazon Web Services

Amazon Web Services is Amazon's secure cloud services platform for businesses to base their IT-infrastructure on. Its major strong point is its scalability, making it ideal for both small and large businesses, as you can base your services on what you need. It uses a pay-as-you-go system, meaning that you only pay when expenses occur, rather than paying in advance or afterwards. AWS is capable of hosting many different services, but this documentation only focuses on their IoT service.

1.3 Requirements

1.3.1 Equipment

USB B to A cable (also known as printer cable) for communicating with Arduino Mega through PC.

1.3.2 Software

Arduino IDE to easily modify and upload the code to Arduino Mega and for using the serial monitor tool.

Official Arduino USB drivers won't work on a Chinese copy of Arduino Mega. You need a Chinese driver going by the name **CH341SER** (should be included in the zip file of the document).

1.3.3 AWS account

If you don't have Amazon Web Services (AWS) account, you can create one at <https://aws.amazon.com/free/>. For account creation, you need a credit card. AWS won't charge you of anything, but if you are not careful there may be some expenses.

Should someone hack your account or use your services without your consent and thus cause you expenses, contact customer support and they will take care of the situation (you most likely won't have to pay for anything).

2 AWS IoT

2.1 Creating a device and certificate

Before the IoT service can be used to send data, the thing (i.e. device) must be registered on AWS IoT. It also needs an x.509 certificate attached to it that will be used to securely authenticate the device. All of this can be accomplished on the AWS IoT service, making the process rather simple. This chapter of the documentation will guide through the whole process of creating a device and certificate on IoT service.

To access IoT Core, log in to AWS console at <https://aws.amazon.com/console/>.



Figure 2. AWS Management Console.

On the top right, choose a proper location, as it will default to US. If you live in the EU, any of the four options should be fine, for there won't be that large of a difference.

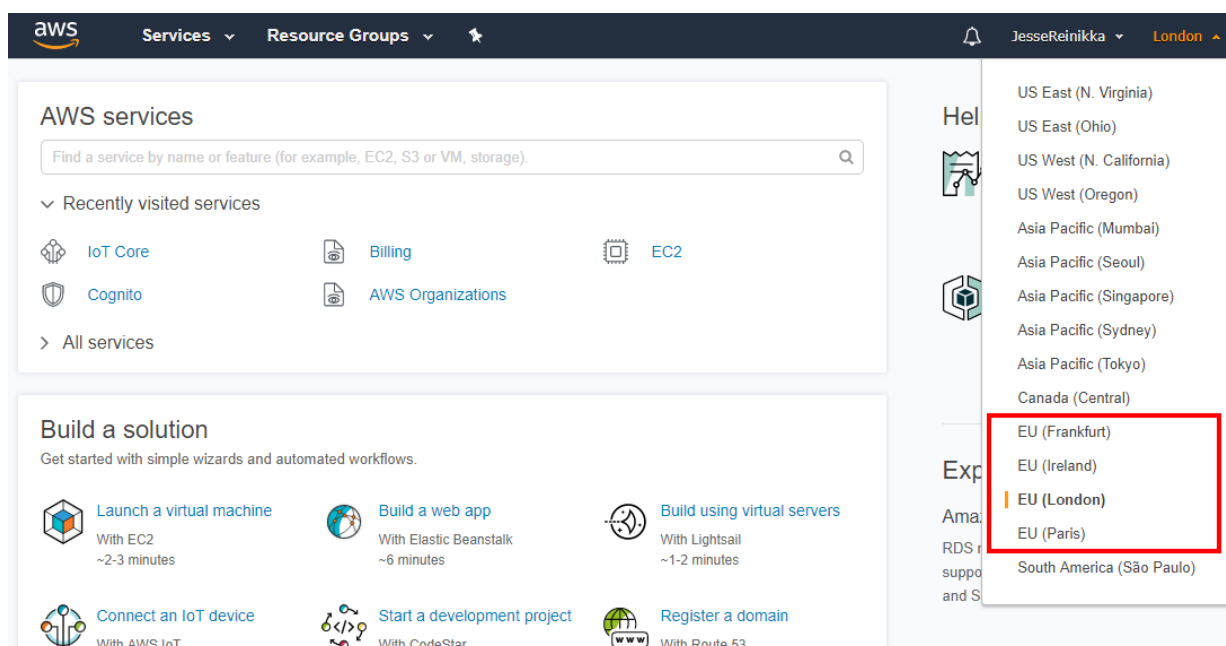


Figure 3. EU locations. (For Finland, fastest may be either Paris or Frankfurt.)

Now click **Services** and on the search box, write “**iot**” and from the list select **IoT Core**.

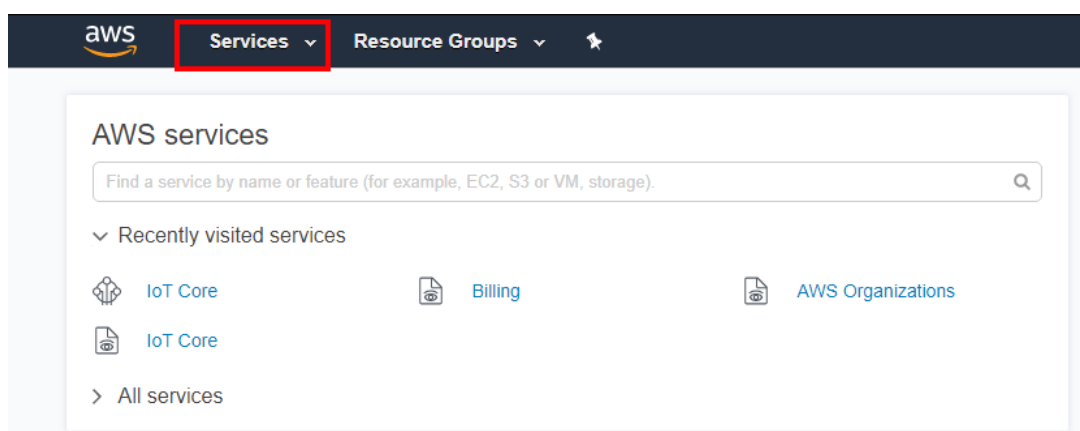


Figure 4. Console dashboard.

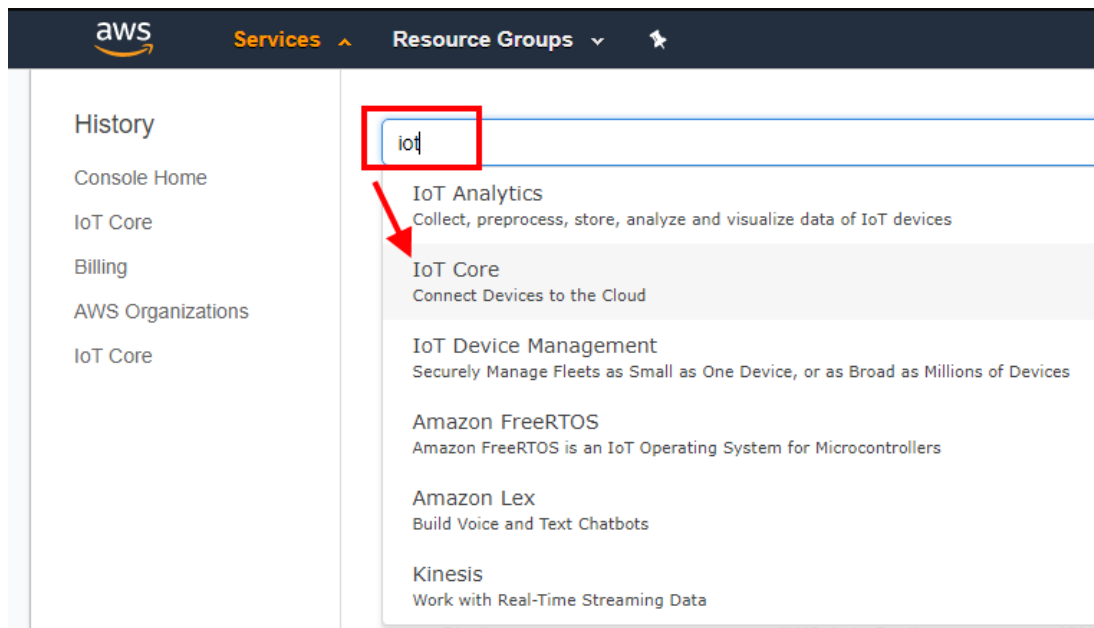


Figure 5. Searching services.

For creating things (i.e. devices), on the left, click **Manage**, then **Things** and finally **Create**.

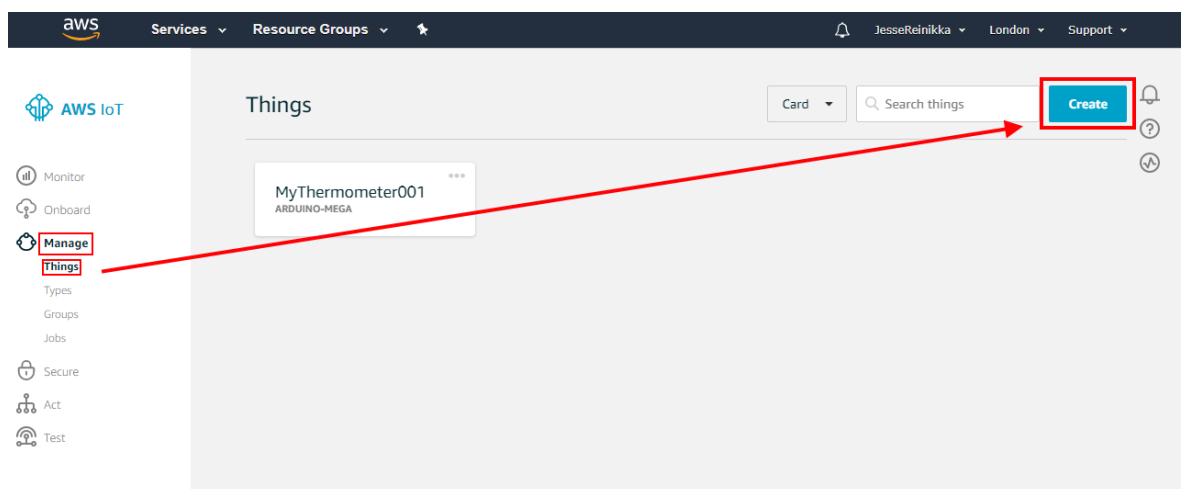
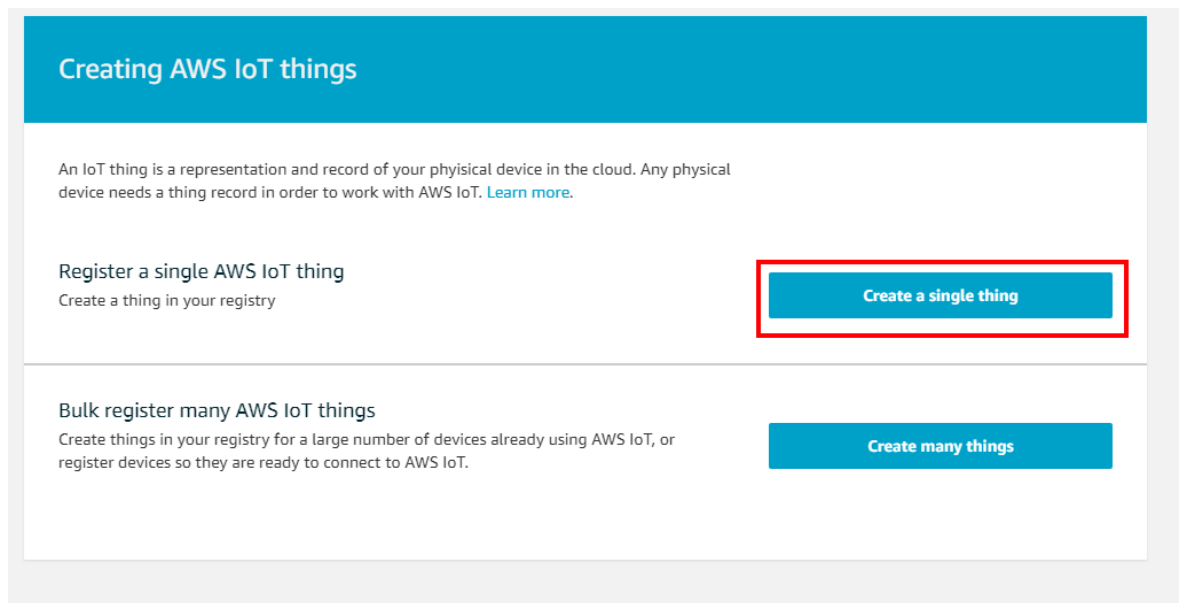


Figure 6. Creating a thing (i.e. device).

Click **Create a single thing**, for simplicity's sake.



Creating AWS IoT things

An IoT thing is a representation and record of your physical device in the cloud. Any physical device needs a thing record in order to work with AWS IoT. [Learn more.](#)

Register a single AWS IoT thing
Create a thing in your registry

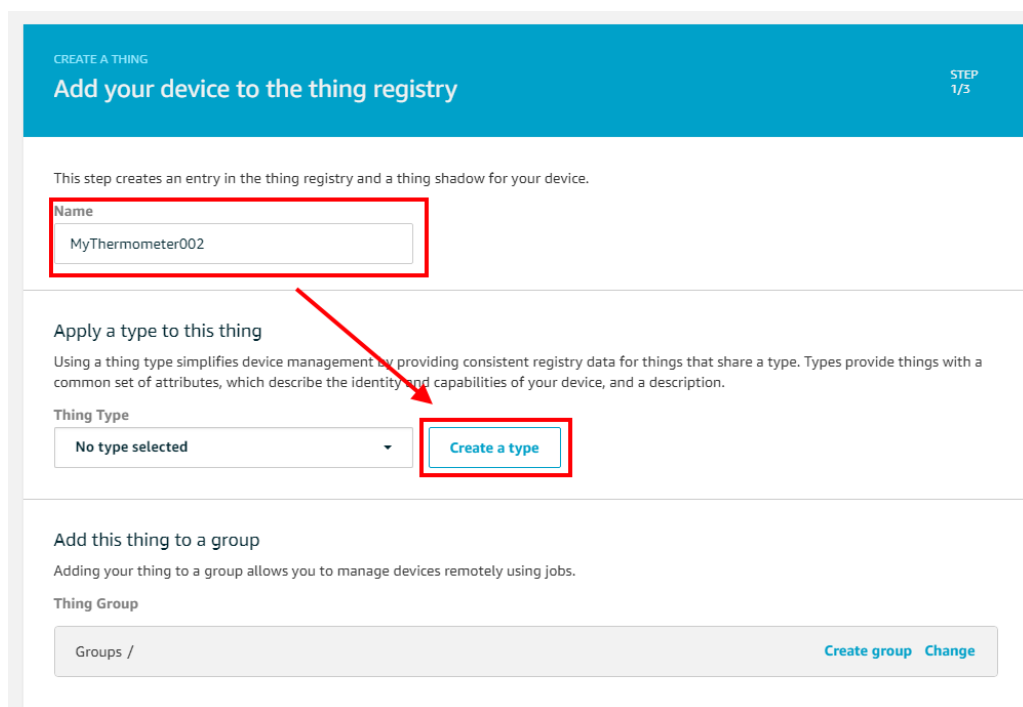
Create a single thing

Bulk register many AWS IoT things
Create things in your registry for a large number of devices already using AWS IoT, or register devices so they are ready to connect to AWS IoT.

Create many things

Figure 7. Creating a single thing.

Enter the desired name. We don't have any types created, so we must create one. Click **Create a type**.



CREATE A THING

Add your device to the thing registry

STEP 1/3

This step creates an entry in the thing registry and a thing shadow for your device.

Name
MyThermometer002

Apply a type to this thing
Using a thing type simplifies device management by providing consistent registry data for things that share a type. Types provide things with a common set of attributes, which describe the identity and capabilities of your device, and a description.

Thing Type
No type selected

Create a type

Add this thing to a group
Adding your thing to a group allows you to manage devices remotely using jobs.

Thing Group
Groups /

[Create group](#) [Change](#)

Figure 8. Thing name and type.

Enter the desired name and description and click **Create thing type** to create it.

Create a thing type

This will help you organize, categorize, and search for your things.

Name
ArduinoMega

Description
Our Arduino Mega

Set searchable thing attributes
You can define up to three attributes for a thing type. Things associated with this type can be searched by using these fields.

[Add another](#)

[Cancel](#) [Create thing type](#)

Figure 9. Creating a thing type.

Newly created thing type should be automatically on the box. Next, we can optionally create a group by clicking **Create Group**. Enter the desired name and description. You can optionally set group attributes to better describe your group. Finally, click **Create thing group**.

Apply a type to this thing
Using a thing type simplifies device management by providing consistent registry data for things that share a type. Types provide things with a common set of attributes, which describe the identity and capabilities of your device, and a description.

Thing Type
ArduinoMega

[Create a type](#)

Add this thing to a group
Adding your thing to a group allows you to manage devices remotely using jobs.

Thing Group
Groups / [Create group](#) [Change](#)

Figure 10. Check Thing Type and optionally create a group.

Create a thing group

Create a thing group to help you organize things.

Parent group

Groups / [Change](#)

Name

SeAMK-IoTTi

Description

SeAMK's IoTTi boards

Set group attributes

Enter a value for one or more of these attributes

Attribute key	Value
Location	SeAMK

[Add another](#) [Clear](#)

Cancel

Create thing group

Figure 11. Create a thing group.

For the thing, you can enter some attributes as well to better identify it. Add another –button gives you another field to enter information on. When you are finished, click **Next** to continue.

Add this thing to a group
Adding your thing to a group allows you to manage devices remotely using jobs.

Thing Group

Groups / [SeAMK-IoTTI](#) /

Create group Change

Groups /

Create group Change

Set searchable thing attributes (optional)
Enter a value for one or more of these attributes so that you can search for your things in the registry.
This thing type does not have searchable attributes

Set non-searchable thing attributes (optional)
You can use thing attributes to describe the identity and capabilities of your device.

Attribute key	Value	
<input type="text" value="Manufacturer"/>	<input type="text" value="TAMK"/>	<input type="button" value="Clear"/>

Show thing shadow ▾

Figure 12. Setting attributes.

On this page, we have many different options to determine what we want to use to authenticate our devices. For simplicity, let's choose One-click certificate creation, by clicking **Create certificate**.

CREATE A THING

Add a certificate for your thing

STEP 2/3

A certificate is used to authenticate your device's connection to AWS IoT.

One-click certificate creation (recommended)
This will generate a certificate, public key, and private key using AWS IoT's certificate authority.

Create certificate

Create with CSR
Upload your own certificate signing request (CSR) based on a private key you own.

Create with CSR

Use my certificate
Register your CA certificate and use your own certificates for one or many devices.

Get started

Skip certificate and create thing
You will need to add a certificate to your thing later before your device can connect to AWS IoT.

Create thing without certificate

Figure 13. Adding certificate.

As you see from the notifications, both the thing and the certificate has been created. You should **download all of the files** and click **Activate** to activate the certificate for your device. You also need the root CA for AWS IoT. Right-click on the download link right above Activate-button and choose **Save link as...** Save it to a good and secure location, alongside with your other certificates, as you are going to need them later.

Certificate created!

Successfully created thing.

Successfully generated certificate. Please download certificate files.

Download these files and save them in a safe place. Certificates can be retrieved at any time, but the private and public keys cannot be retrieved after you close this page.

In order to connect a device, you need to download the following:

A certificate for this thing	9f72d3400c.cert.pem	Download
A public key	9f72d3400c.public.key	Download
A private key	9f72d3400c.private.key	Download

You also need to download a root CA for AWS IoT from Symantec:
A root CA for AWS IoT [Download](#)

Activate

Figure 14. Thing and certificate creation successful.

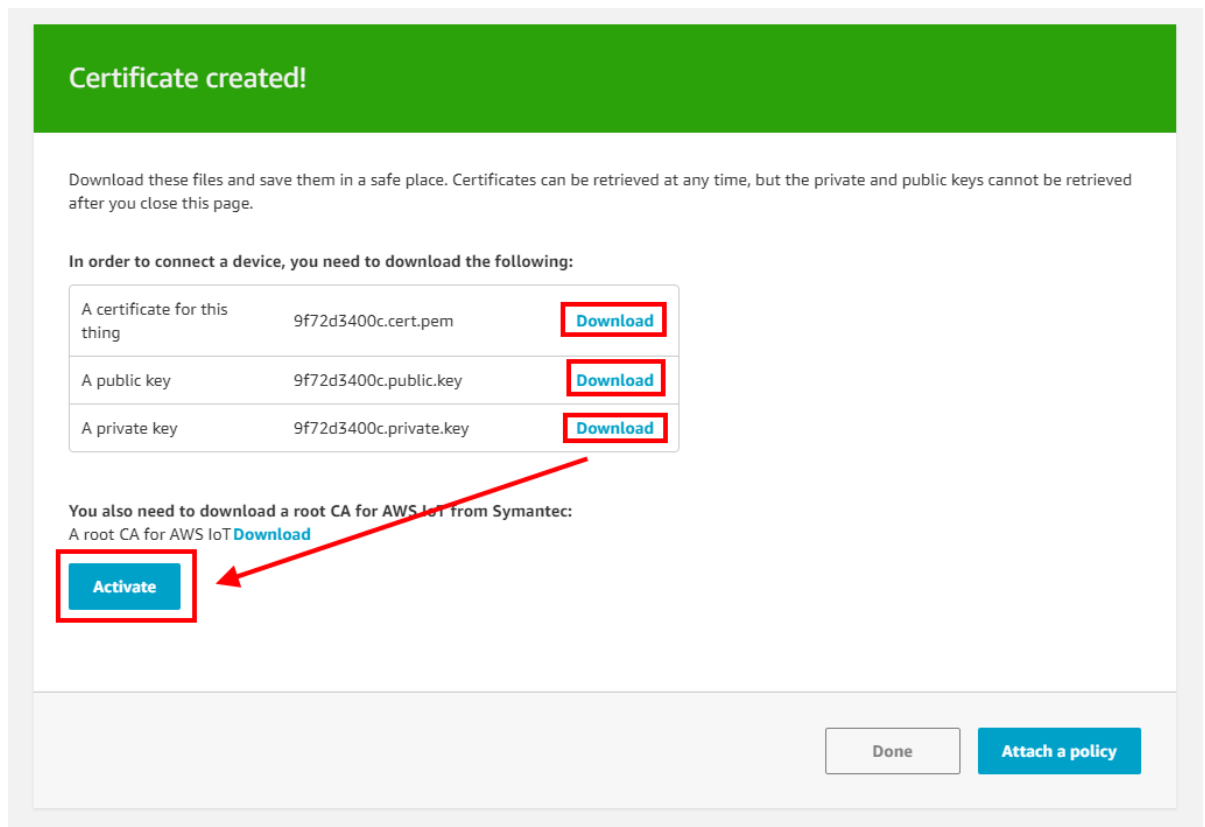


Figure 15. Download and activate certificates.

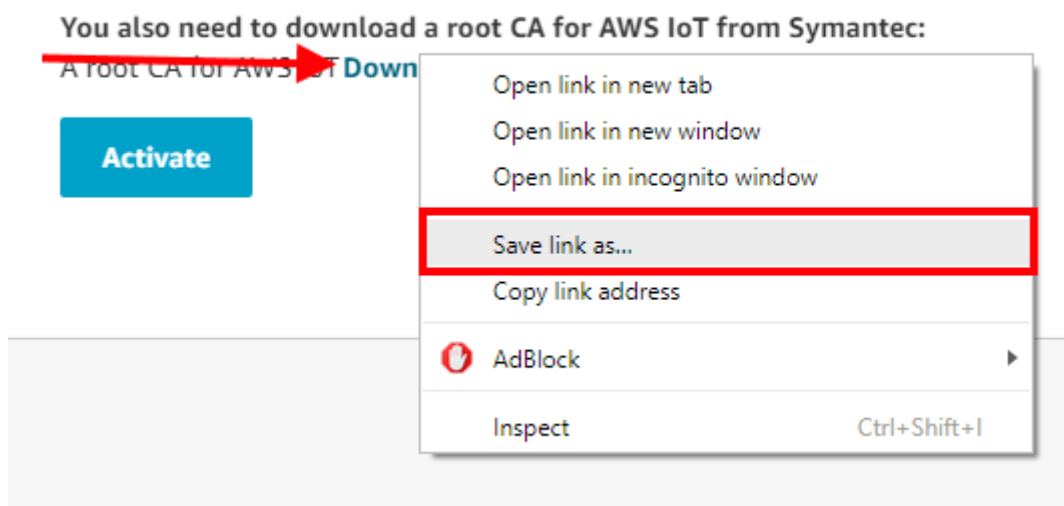


Figure 16. Save public certificate.

Since we don't have any policies available yet, click **Done**. You can now see your newly created thing on Things page.

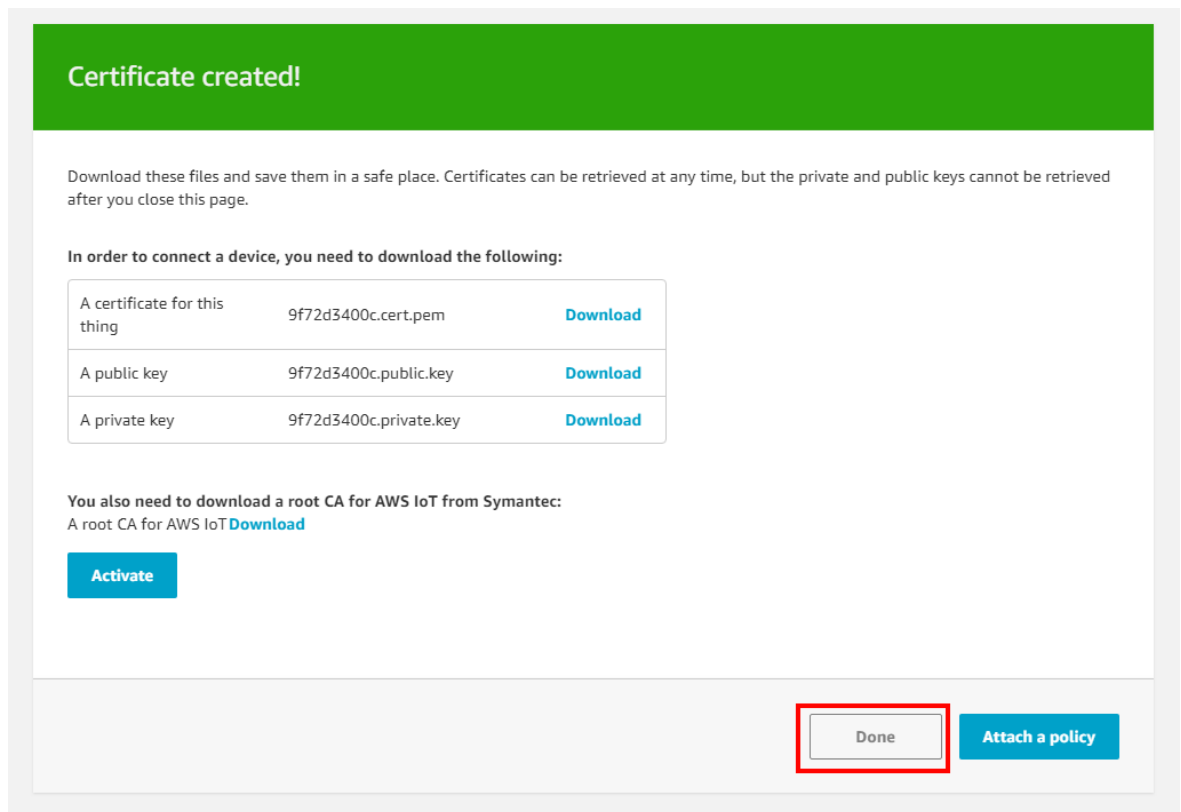


Figure 17. Finish certificate creation.

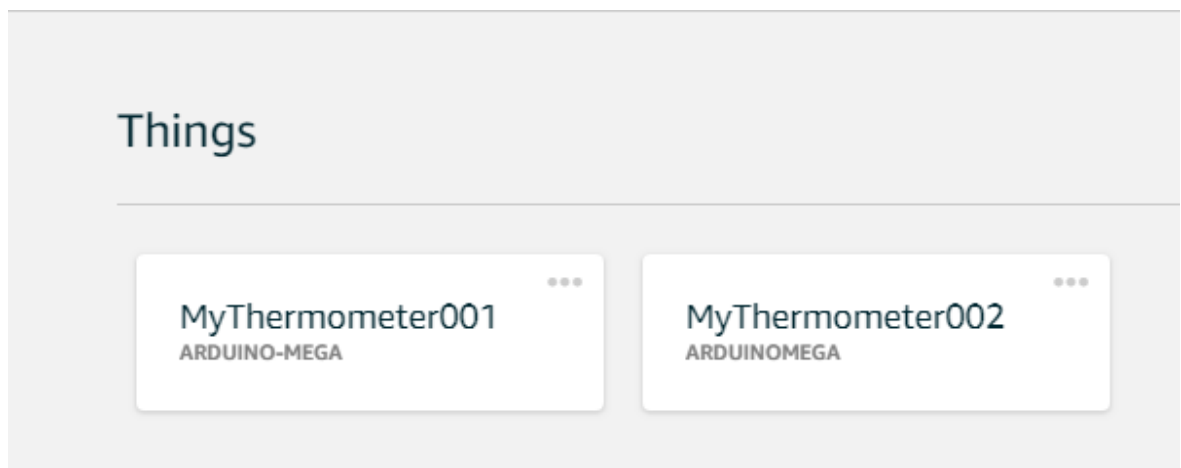


Figure 18. Created things.

2.2 Creating a policy

Now that a device has been registered and with a proper certificate, next a policy needs to be created. While certificates are used to authenticate the device, a policy is used to authorize various AWS IoT operations. These policies are attached to

certificates. Should you attach the same certificate to multiple devices, they all will be able to perform the same actions defined on the policy. To create one, from the left, click **Secure**, then **Policies** and finally **Create**.

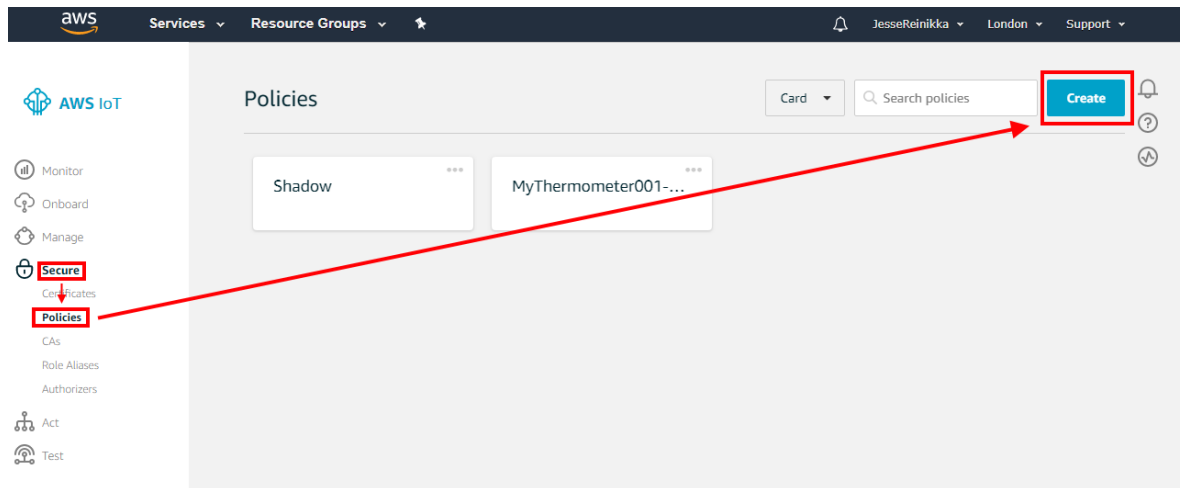


Figure 19. Creating a policy.

Enter the desired name for the policy.

Figure 20. Policy name.

On the Add statements, write in `iot` and choose **iot:*** to add all of the actions into the policy, thus making it ideal for testing. Lastly, choose **Allow** as an effect and click **Create**.

Add statements

Policy statements define the types of actions that can be performed by a resource. Advanced mode

Action

iot*

- iot:Publish
- iot:Subscribe
- iot:Connect
- iot:Receive
- iot:UpdateThingShadow
- iot:GetThingShadow
- iot>DeleteThingShadow

Add statement

Figure 21. Defining statement actions.

Add statements

Policy statements define the types of actions that can be performed by a resource. Advanced mode

Action

Resource ARN

Effect

☒ Allow ☐ Deny

Remove

Add statement

Create

Figure 22. Make statement to allow and create the policy.

To put the newly created policy to use, we must attach it to the certificate. On the left, Click **Secure**, then **Certificates** and click ...-button on the certificate. Select **Attach policy**.

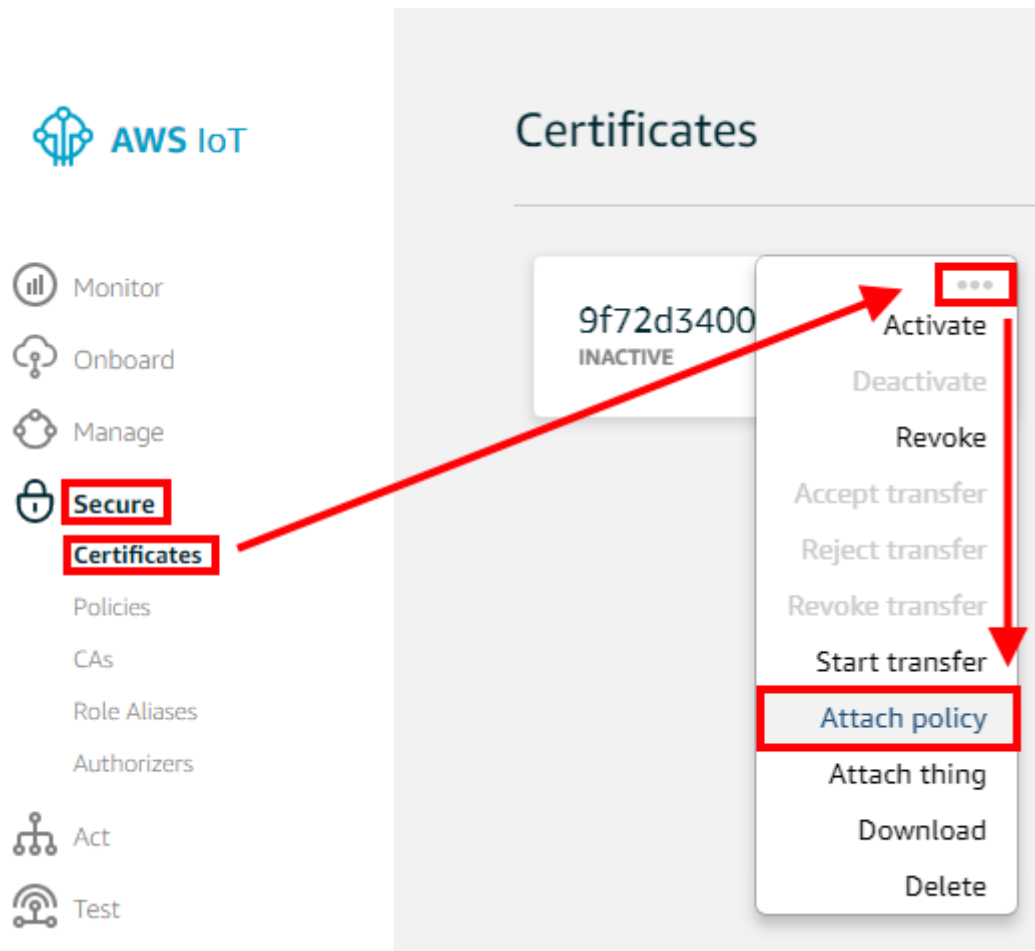


Figure 23. Attaching a policy to certificate.

You are presented with a list of your policies. Select a policy by clicking on a box and click **Attach**. A notification tells us if it was successful.

Attach policies to certificate(s)

Policies will be attached to the following certificate(s):
9f72d3400c9598ab546d7a89cd988cf88762685cfce98b92b4acff8bf9b8f9b1

Choose one or more policies

Search policies	
<input checked="" type="checkbox"/> Test	View
<input type="checkbox"/> Shadow	View
<input type="checkbox"/> MyThermometer001-Policy	View

1 policy selected Cancel Attach

Figure 24. Available policies.

Certificates

Card Successfully attached policy. X

9f72d3400c9598ab54... INACTIVE	e5dafa828c39f8bf06... ACTIVE
-----------------------------------	---------------------------------

Figure 25. Successful attachment.

2.3 API endpoint and monitoring device

Before we get to program the device, we must write down our API endpoint at Settings to safe location. If AWS has finished setting up your IoT service, it should be

right at the beginning of the page. If you don't have an endpoint yet, wait for the AWS to set your IoT Service.

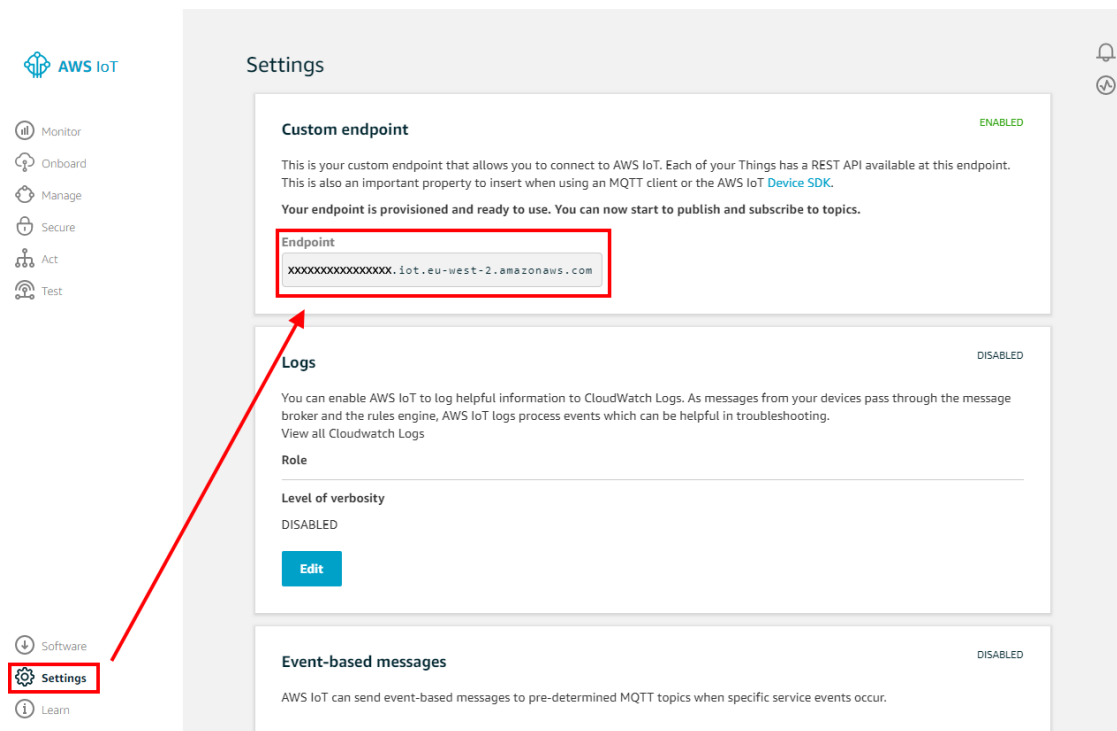


Figure 26. Custom endpoint.

When you have your API endpoint written down, navigate back to your things at **Manage -> Things**. Click your thing (MyThermometer002).

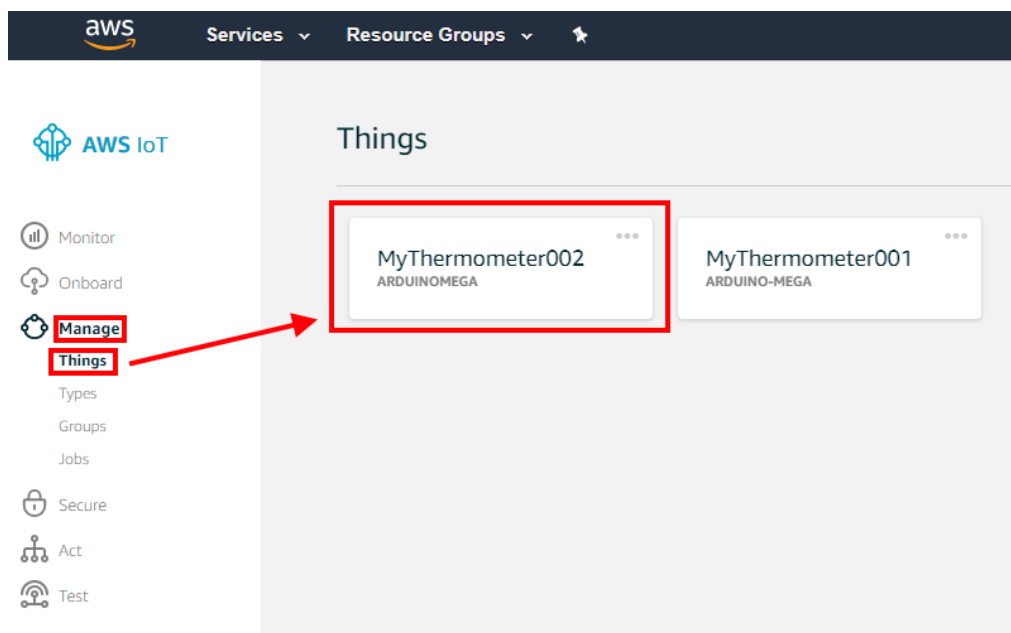


Figure 27. Select a thing.

From the left, choose **Activity**. Here AWS IoT will automatically update the page with API calls. It starts listening for actions the moment you open the page (notice Listening for 0 minute(s)), so should you have send data before coming to this page, you will not be able to see it. Keep this page open for troubleshooting the device, so that you know it is able to connect and send data to AWS.

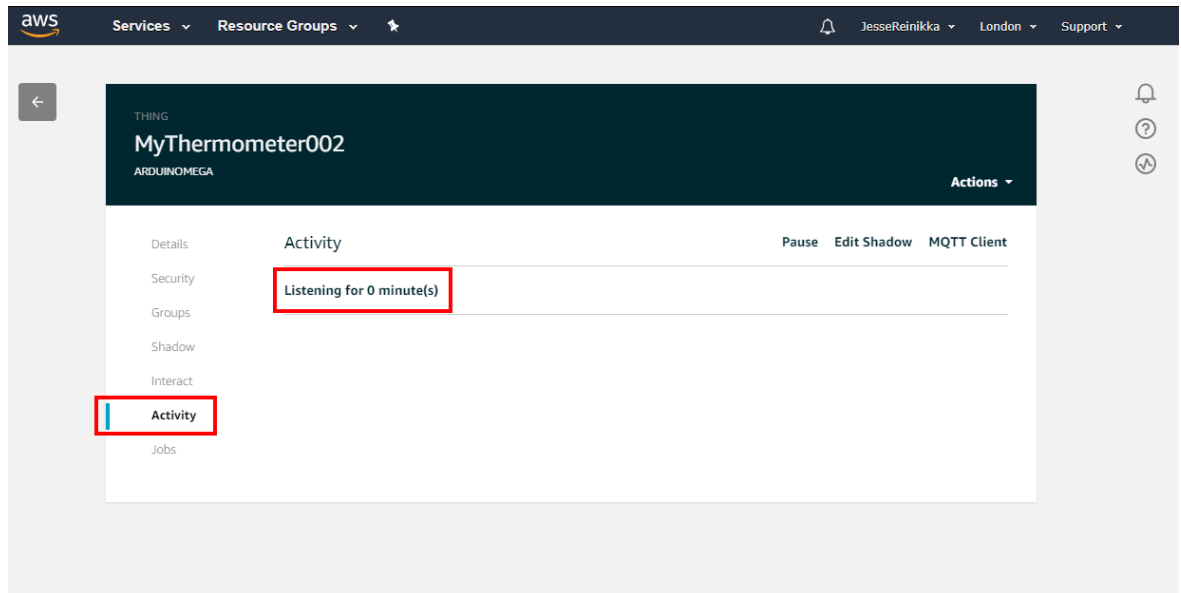


Figure 28. Monitoring thing activity.

3 Arduino Mega

In this part we will go through using the Arduino IDE and uploading the code to Arduino Mega. You can optionally skip this part if you are already well-versed with Arduino IDE (simply upload the code to the board, as no changes need to be made). If you didn't get the code with this document, you can get it from [GitHub](https://github.com) or the appendix of this document.

First, open Arduino IDE or install it from <https://www.arduino.cc/en/Main/Software>. Should you require further assistance, use their installation manual at <https://www.arduino.cc/en/Guide/Windows>.

Open the code file (JSON-To-PC.ino). Make sure, that you are compiling to **Arduino Mega 2560** a different board that you may be using.

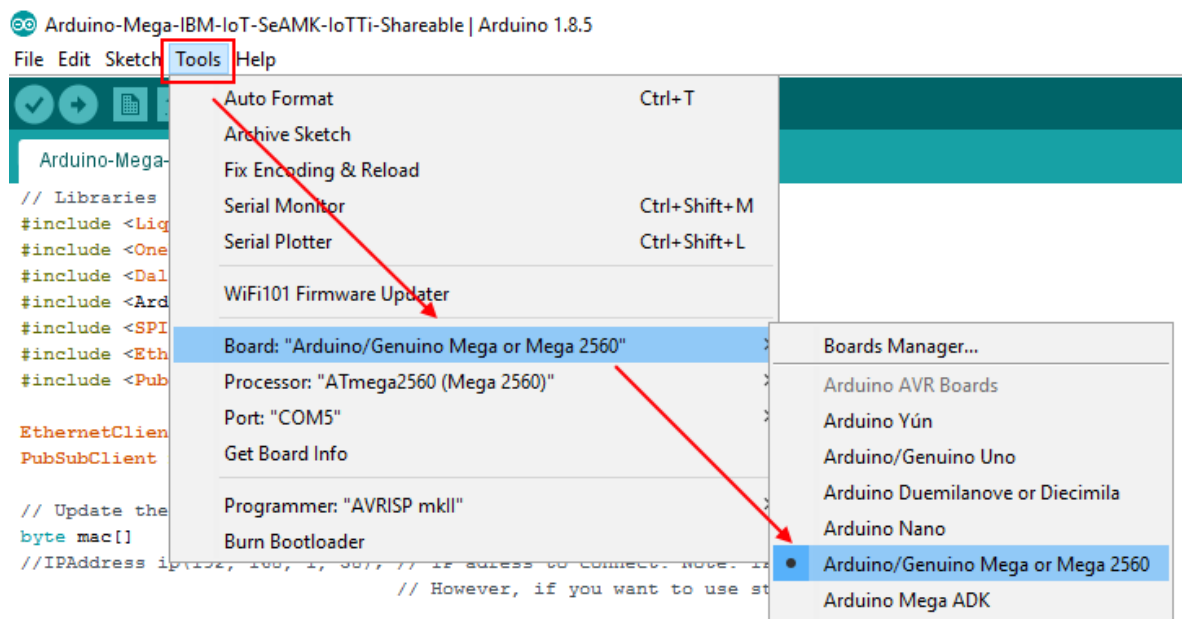


Figure 29. Choose the board to develop to.

Next, check that all the libraries are installed. That can be done either through **Sketch -> Verify/Compile...**

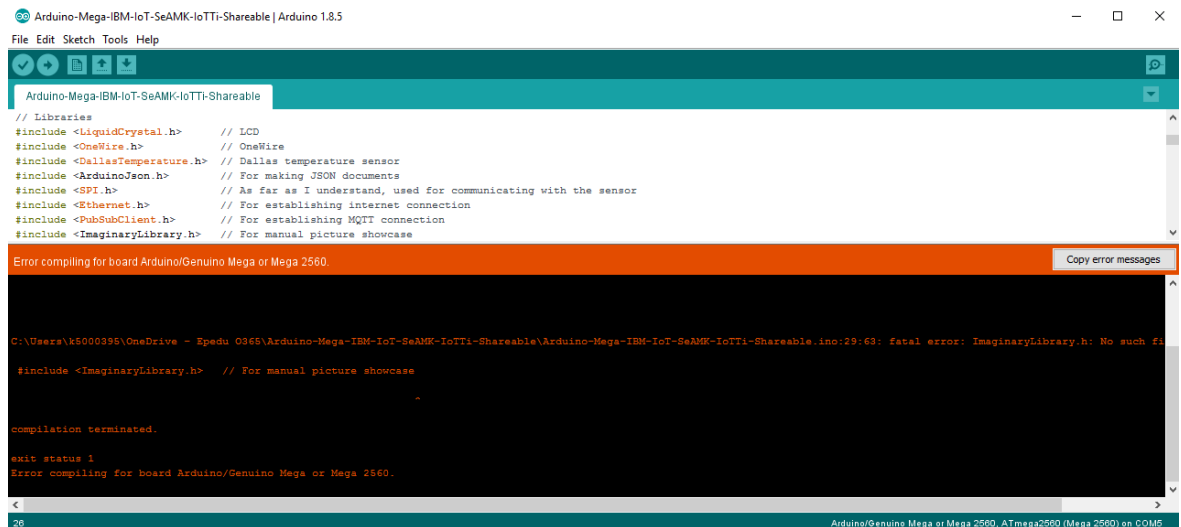


Figure 30. Results of compiling without having the libraries installed (error).

Or going straight to **Sketch -> Include Library -> Manage Libraries...** and with the aid of Library Manager, check and install all the required libraries. (The libraries are: LiquidCrystal, OneWire, DallasTemperature and ArduinoJson).

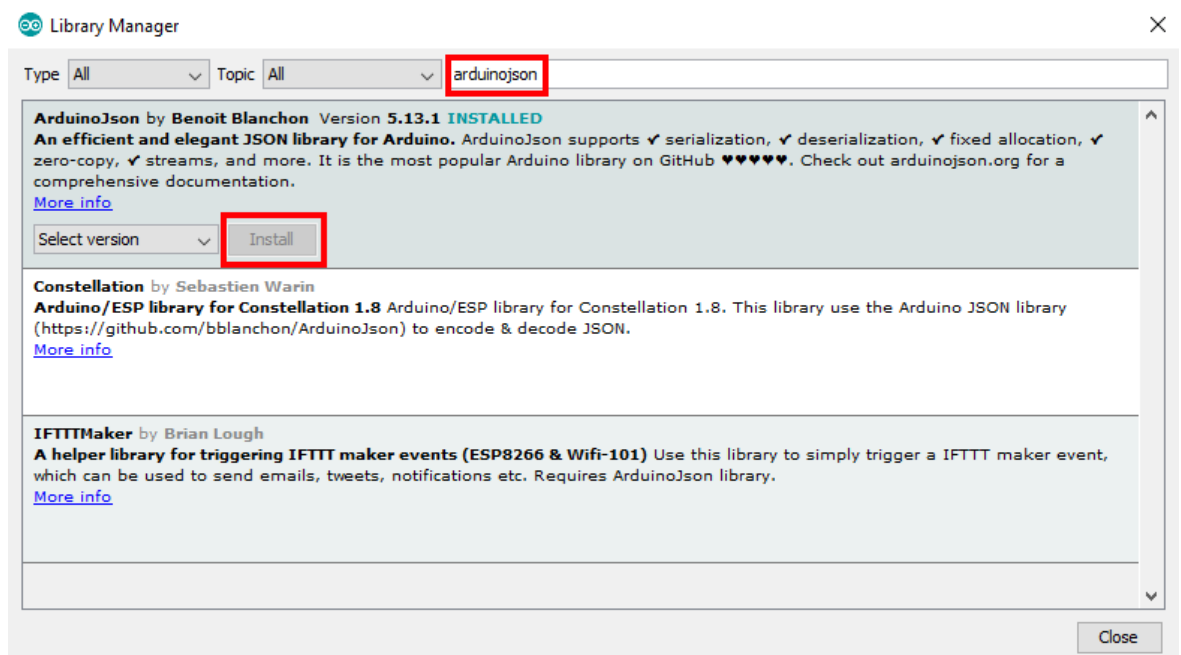


Figure 31. Searching libraries with Library Manager.

With **Sketch -> Verify/Compile** check that everything is correct.

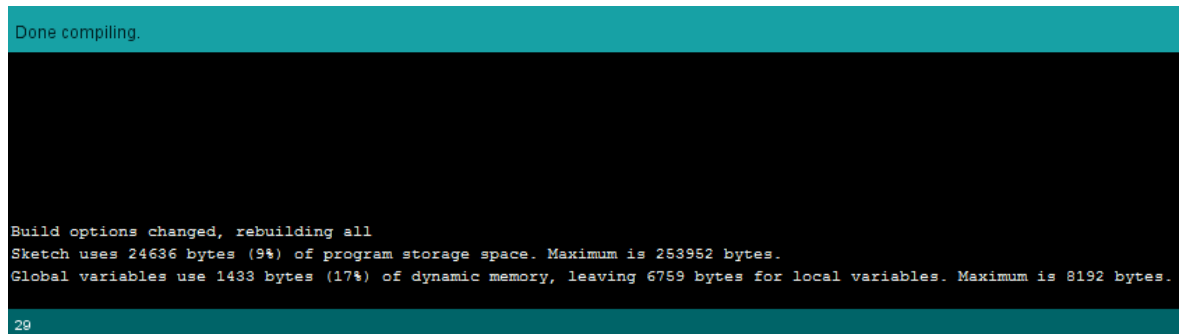


Figure 32. Successful compiling.

Finally, to upload the code, **connect the Arduino Mega to PC with USB B to A cable**. Check that there is a port selected at **Tools -> Port** (if you have only one board connected, there should be only one option). Check that everything is set up the way it is in Figure 33.

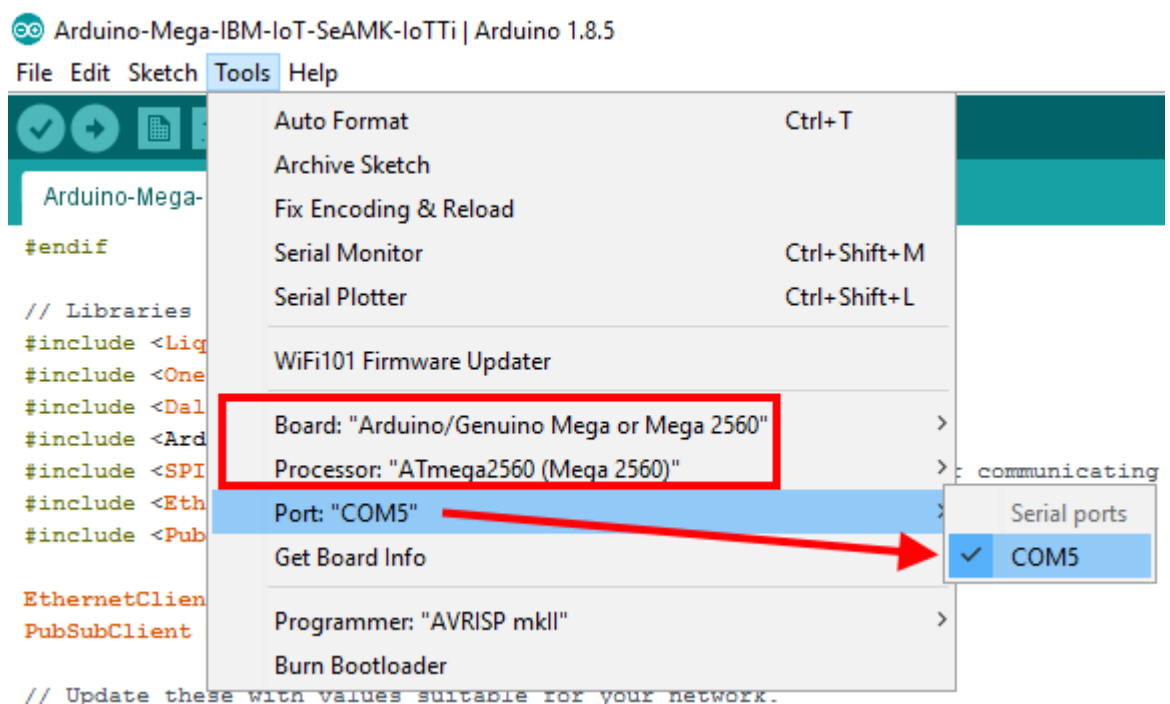


Figure 33. Choose the correct port (note: port number may be different from the picture).

Now you can **Sketch -> Upload**. When uploading is completed you can move onto the next part where we set up the simple Python program.

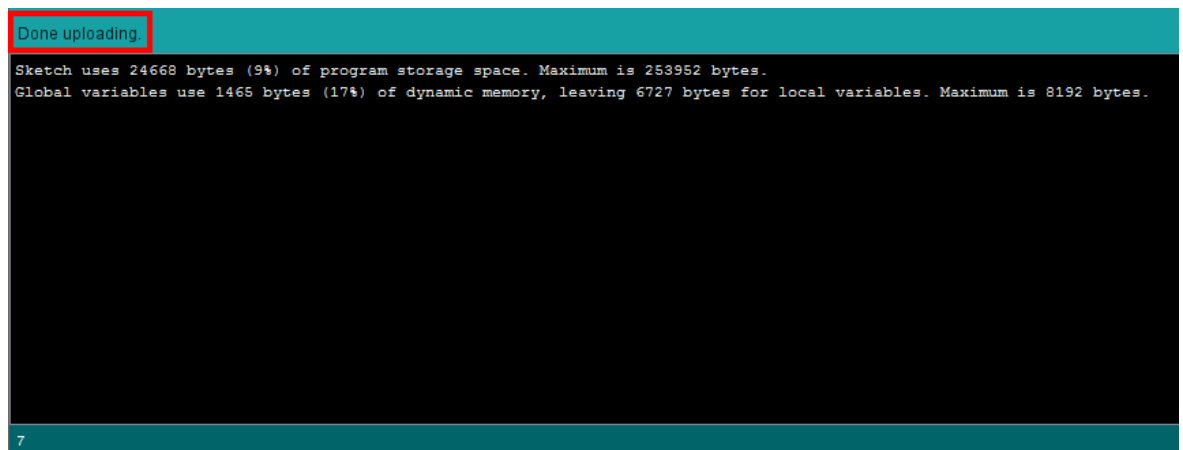


Figure 34. Successful upload to the board.

4 Python program

Before running the python program with your favorite IDE (this documentation uses Visual Studio Code), you need to change the AWS login details and provide your own certificates created in chapter 2.1. As always, if you didn't get the code with this document, it can always be found from [GitHub](#) or the appendix of this document. The most up to date code will always be at [GitHub](#).

To edit **AWS_details.py**, open it with your favorite IDE. It is used to store the details of your IoT service, for easier accessibility. Put **your certificates into login folder** and **rename them accordingly**. Next change the hostname to **your** API endpoint. Thing name should be the name you gave to your thing. Lastly, if you are using different JSON object data, then change the names in dataNames variable in callbacks.py.

```
# root and device certificate, alongside with device private key should be put into the login folder
loginFolder = "login/"

# Paths to the certificates, rename the certificates accordingly.
# Note: Make sure there is no spaces in the name
rootCertificatePath = loginFolder + "rootCertificate.pem"
privateKeyPath = loginFolder + "MyThermometer001.private.key"
certificatePath = loginFolder + "MyThermometer001.cert.pem"

appName = "testIoTPySDK" # Name of the app, can be anything.
hostname = "xxxxx.iot.eu-west-2.amazonaws.com" # Your API endpoint.
port = 8883 # 8883 when connecting with MQTT, 8443 for HTTP, 443 for WebSocket and HTTP.
thingName = "MyThermometer001" # Name of your thing (i.e. device) on AWS
```

Figure 35. AWS details.

dia > Python > AWS-ShadowUpdate > login				Search login	
Name	Date modified	Type	Size		
MyThermometer001.cert.pem	23.3.2018 12.07	PEM File	2 KB		
MyThermometer001.private.key	23.3.2018 12.07	KEY File	2 KB		
rootCertificate.pem	23.3.2018 12.07	PEM File	2 KB		

Figure 36. Example login folder.

```
# Insert the name fields of your JSON data into the following array.
# They are used to dynamically print the sent data in customShadowCallback_Update method.
dataNames = ["sensor", "temperature", "voltage"]
```

Figure 37. dataNames in callbacks.py.

And that is all. Run the code while Arduino is connected to the serial port and it should be sending data to AWS. If you don't have an Arduino or there are some issues, you may try the test_AWS_connection.py to test the connection to AWS IoT service.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Delete request with token: 7470dba7-9c3c-4b60-b292-5bf5b54fe9f3 accepted!

Update request with token: 39b3e433-2ef0-46cf-8a47-d67110c5e6ab accepted!
sensor: Dallas
temperature: 22.875
voltage: [4.345703, 4.990234]
```

Figure 38. VS Code debug console.

Details	Activity	Pause	Edit Shadow	MQTT Client
Security	Listening for 24 minute(s)			
Groups				
Shadow	▶ ● Shadow update accepted			Mar 27, 2018 10:21:13 AM +0300
Interact	▶ ● Shadow update accepted			Mar 27, 2018 10:20:13 AM +0300
Activity	▶ ● Shadow update accepted			Mar 27, 2018 10:19:13 AM +0300
Jobs	▶ ● Shadow update accepted			Mar 27, 2018 10:18:13 AM +0300
	▶ ● Shadow update accepted			Mar 27, 2018 10:17:13 AM +0300
	▶ ● Shadow update accepted			Mar 27, 2018 10:16:13 AM +0300
	▶ ● Shadow update accepted			Mar 27, 2018 10:15:13 AM +0300
	▶ ● Shadow update accepted			Mar 27, 2018 10:14:13 AM +0300
	▶ ● Shadow update accepted			Mar 27, 2018 10:13:13 AM +0300
	▶ ● Shadow update accepted			Mar 27, 2018 10:12:13 AM +0300

Figure 39. Thing activity.

▼ ● Shadow update accepted

Mar 27, 2018 10:21:13 AM +0300

```
{
  "state": {
    "desired": {
      "sensor": "Dallas",
      "temperature": 23,
      "voltage": [
        4.335938,
        5
      ]
    }
  },
  "metadata": {
    "desired": {
      "sensor": {
        "timestamp": 1522135273
      },
      "voltage": [
        {
          "timestamp": 1522135273
        },
        {
          "timestamp": 1522135273
        }
      ]
    }
  },
  "version": 978,
  "timestamp": 1522135273,
  "clientToken": "aab0931c-5425-4fb6-9087-d80b531af65d"
}
```

Figure 40. Sent data.

5 The code

This part is for those that want a better look at the code and understand what is happening and why.

5.1 Arduino code

The method `setup` is mainly used to initialize peripherals and open serial communications at startup.

```
void setup()  
{  
  Serial.begin(9600); // Open serial communications.  
  sensors_1.begin(); // Start Dallas temperature sensor  
  lcd.begin(20, 4); // Set up the LCD's number of columns: 20 and rows: 4  
}
```

In `loop()`, it checks if it has received data through serial. When it has, it will execute the `workflow` method, which will be explained afterwards. Now the most important part here is `Serial.flush()`. This forces Arduino to stop the program from running, until the transmission of serial data has completed. Why is this important? To maintain the timing. We want Arduino to be in sync with the program that gets the data, so there won't be any errors when reading the data. Without `Serial.flush()`, the read JSON document could be missing some parts, thus causing an error when we are attempting to send the incomplete JSON document to AWS IoT as it won't be even recognized as JSON document to begin with. In essence, without `Serial.flush()` our setup wouldn't work as intended (sending data every minute), as when one JSON document fails, the rest will fail as well. It would loop an error message in python program every minute, never achieving anything until we restart it. That would be very impractical, so remember to use `Serial.flush()` when sending data through serial.

```

void loop()
{
    // Send data only when you receive any data (i.e. command) from client.
    if (Serial.available() > 0) {
        workflow();
        Serial.flush();
    }
}

```

Method `workflow()` is used to execute three methods in the following order.

```

// The order of methods to handle data
void workflow(void)
{
    measure();           // First measure and save the data
    print_measurement(); // Print them on the LCD screen
    send_data();         // Publish data to AWS
}

```

Method `measure()` is TAMK's code for getting temperature from sensor and voltage of certain analog pins. All I did was add comments to explain what it does. `requestTemperatures()` gets the temperature from the sensor and then it can be stored by using `getTempCByIndex()`. Since there is only one sensor, its index is 0.

```

void measure(void)
{
    sensors_1.requestTemperatures(); // Get temperature from Dallas sensor

    voltageV1 = analogRead(A0); // Get the voltage on analog pin
    voltageV1 = voltageV1*10.0/1024;

    voltageV2 = analogRead(A1); // Get the voltage on analog pin
    voltageV2 = voltageV2*10.0/1024;

    temperature1=sensors_1.getTempCByIndex(0); // Save the temperature value on
    global variable
}

```

Method `print_measurement()` is also TAMK's code. Only thing that is changed is what it prints on the LCD screen. The `setCursor()` method is used to control from where we start writing the text, while `print()` method is used to print text on the screen.

```

void print_measurement(void)
{
    lcd.setCursor(0,0);
    lcd.print("SeAMK IoTti");

    lcd.setCursor(0,1);
    lcd.print("Voltage V1 =");
    lcd.setCursor(13,1); lcd.print(voltageV1,2);lcd.print(" V");

    lcd.setCursor(0,2);
    lcd.print("Voltage V2 =");

    lcd.setCursor(13,2); lcd.print(voltageV2,2);lcd.print(" V");

    lcd.setCursor(0,3);
    lcd.print("Temperature =");
    lcd.setCursor(13,3); lcd.print(temperature1,2);lcd.print(" C");
}

```

Method `send_data()` does exactly what its name implies. To keep things organized and clear, the creation of JSON document is put to a separate method.

```

// Send data to serial, to be read by a program
void send_data(void)
{
    createJSON(); // Set up the data to be sent

    // Send data through serial to the client
    Serial.println(JSON_Data); // Note: needs to be Serial.println !!! In the
    client (one running the python code),
                                // it is used as a way to determine when there is
    no more data to be sent.
}

```

With the aid of `ArduinoJson.h` library, you can easily create JSON documents without having to make the process all cryptic and hard to understand for those without much programming knowledge (it gets very complex and impractical, when you do it by yourself without the aid of libraries and when you have a lot of data). The library doesn't use `malloc()` to dynamically allocate memory, which makes it even better to use on embedded systems. Process itself should be simple. Since AWS Shadow updates require an object *desired* under the object *state*, it needs to be created like so.

```

StaticJsonBuffer<200> jsonBuffer; // Allocate JSON buffer with 200-byte pool

JsonObject& rootJsonObject = jsonBuffer.createObject();
JsonObject& nestedJsonObject_STATE = rootJsonObject.createNestedObject("state");
create JSON object state - required by AWS
JsonObject& nestedJsonObject_DESIRED =
nestedJsonObject_STATE.createNestedObject("desired"); // create JSON object
desired, under state - required by AWS

```

Thus, creating a following JSON document.

```
{
  "state"
  {
    "desired": {
      }
    }
  }
}
```

Then to populate *desired* with our data.

```
// Add following data to object desired
nestedJsonObject_DESIRED["sensor"] = "Dallas"; // Create JSON object named
"Sensor", assigned with the name of our sensor
nestedJsonObject_DESIRED["temperature"] = temperature1; // Create JSON object
named "Temperature", assigned with our temperature data
```

To create an array of data.

```
JsonArray& nestedJsonArray_VOLTAGE =
nestedJsonObject_DESIRED.createNestedArray("voltage"); // Creates an array of
data with name Voltage
nestedJsonArray_VOLTAGE.add(voltageV1); // Add V1 to the array
nestedJsonArray_VOLTAGE.add(voltageV2); // Add V2 to the array
```

To save the data into char[] variable.

```
rootJsonObject.printTo(JSON_Data);
```

Now, it should be in the following format.

```
{
  "state"
  {
    "desired": {
      "sensor": "Dallas",
      "temperature": 23,
      "voltage": [
        4.335938,
        5
      ]
    }
  }
}
```

Ready to be printed to serial.


```
// Send data through serial to the client
Serial.println(JSON_Data); // Note: needs to be Serial.println !!! In the client
                             // it is used as a way to determine when there is
                             // no more data to be sent.
                             (one running the python code),
```

5.2 Python code

This part will be only covering `__main__.py` as others are self-explanatory.

At the beginning, we are mainly creating and defining the client used to connect to AWS IoT. By storing the AWS login details on a dictionary variable in a separate file, the code becomes much more compact.

```
# Create an AWS IoT MQTT Client using TLSv1.2 Mutual Authentication
myAWSIoTMQTTShadowClient = AWSIoTMQTTShadowClient(details["appName"])
myAWSIoTMQTTShadowClient.configureEndpoint(details["hostname"], details["port"])
myAWSIoTMQTTShadowClient.configureCredentials(details["rootCertificate"],
                                              details["privateKey"],
                                              details["certificate"])

# AWS IoT MQTT Client connection configuration
myAWSIoTMQTTShadowClient.configureAutoReconnectBackoffTime(1, 32, 20)
myAWSIoTMQTTShadowClient.configureConnectDisconnectTimeout(10) # 10 sec
myAWSIoTMQTTShadowClient.configureMQTTOperationTimeout(5) # 5 sec

myAWSIoTMQTTShadowClient.connect() # Connect to AWS IoT

deviceShadowHandler =
myAWSIoTMQTTShadowClient.createShadowHandlerWithName(details["thingName"], True)

# Delete shadow JSON doc.
# Note: If there is no data to be deleted, it will be rejected (doesn't crash/stop
the program)
deviceShadowHandler.shadowDelete(customShadowCallback_Delete, 5)
```

Afterwards, the main method is executed, starting with defining the serial connection. COM4 stands for the USB port Arduino is connected to. This can be easily found out by launching the Arduino IDE and checking Tools -> Port while it is connected to PC. `Time.sleep(2)` is used to stop the code for two seconds, to give Arduino time to become ready.

```
def main():
    """ Get data from Arduino and sent it to AWS IoT """

    # Establish serial connection.
    try:
        arduinoSerial = serial.Serial("COM4", timeout=1, baudrate=9600)
    except:
        print("Unable to establish serial connection with Arduino. Check the port.")

    time.sleep(2) # Give time for arduino to wake up.
```

The while-loop is used to send data as long as the Arduino is connected to the computer. It starts by sending number 1 to Arduino. The value itself doesn't matter, as the code in Arduino is only used to start sending data after it has received any sort of data. This way, Arduino doesn't continuously send data, when it doesn't have to. It also helps keeping the Python program and Arduino synced with each other.

```
# Keep looping while the port is open.
while arduinoSerial.isOpen():
    arduinoSerial.write(1) # Tell Arduino to send data.
```

Next line of code, reads and returns a line of text from the Arduino. It will stop reading, when a newline (\n) is spotted. Should you forget to make your Arduino to print with newline, it will stay there reading till the end of time. Strip(), without any arguments, removes any whitespace in the data. Note that the data will be in binary.

```
JSON_Data = arduinoSerial.readline().strip() # Get the data from arduino serial
```

Next is a for-loop that is used to decode the binary data to UTF-8 and split upon hitting newline. With this, the newline that is added to the data will be removed (also note, that this loop will execute only once, as there is no more data after that newline). Lastly, device shadow handler is used to execute the shadow update (i.e. upload data). We convert the encoded data to string, make it use our custom callback (prints out the sent data, if successful) and make it timeout in five seconds, should there be any issues with connection.

```
# As the data is in bytes, next we make sure it is in the right format and upload the data to AWS
for data in JSON_Data.decode('utf-8').split('\n'):
    deviceShadowHandler.shadowUpdate(str(data), customShadowCallback_Update, 5)
```

In the last part of the while-loop, the serial port is closed as it won't be used for a while. It waits 58 seconds, before attempting to open the port again. As always, Arduino is given two seconds to be ready. Should the Arduino to be disconnected from computer during this time, the while loop will stop executing the code and it has to be restarted to continue.

```
arduinoSerial.close() # Close the port
time.sleep(58) # Wait for 58 seconds before uploading again

try:
    arduinoSerial.open() # Open the port again.
except:
    print("Unable to open port.")
time.sleep(2) # Give time for arduino to wake up.
```

Appendix

Annex 1. Arduino code.

Annex 2. Python code (old).

ANNEX 1. Arduino code

```
// This code is used by Arduino Mega that is connected to Dallas temperature
// sensor and
// LCD screen. Before any data is send, any message has to be send to Arduino's
// Serial.
// After that, the JSON data is send to Serial, that can be read with e.g. Py-
// thon program.

// Libraries
#include <LiquidCrystal.h>      // LCD
#include <OneWire.h>           // OneWire
#include <DallasTemperature.h> // Dallas temperature sensor
#include <ArduinoJson.h>       // For making JSON documents

#define ONE_WIRE_BUS_1 40      // Data wire is plugged into port 2 on the
// Arduino
OneWire oneWire_1(ONE_WIRE_BUS_1); // Setup a oneWire instance to communicate
// with any OneWire devices (not just Maxim/Dallas temperature ICs)

DallasTemperature sensors_1(&oneWire_1); // Pass our oneWire reference to Dallas
// Temperature.

//          RS  E   D4  D5  D6  D7
LiquidCrystal lcd(37, 36, 35, 34, 33, 32);

// Data variables
float voltageV1,      // Used to store voltage V1
      voltageV2,      // Used to store voltage V2
      temperature1;   // Used to store temperature
char JSON_Data[200];  // Used to store the generated data in JSON

void setup()
{
  Serial.begin(9600); // Open serial communications.
  sensors_1.begin();  // Start Dallas temperature sensor
  lcd.begin(20, 4);    // Set up the LCD's number of columns: 20 and rows: 4
}

void loop()
{
  // Send data only when you receive data (i.e. command) from client.
  if (Serial.available() > 0) {
    workflow();
    Serial.flush();
  }
}

// Prints the information on LCD
void print_measurement(void)
{
  lcd.setCursor(0,0);
  lcd.print("SeAMK IoTti");

  lcd.setCursor(0,1);
  lcd.print("Voltage V1 =");
  lcd.setCursor(13,1); lcd.print(voltageV1,2);lcd.print(" V");

  lcd.setCursor(0,2);
  lcd.print("Voltage V2 =");

  lcd.setCursor(13,2); lcd.print(voltageV2,2);lcd.print(" V");
}
```

```

    lcd.setCursor(0,3);
    lcd.print("Temperature =");
    lcd.setCursor(13,3); lcd.print(temperature1,2);lcd.print(" C");
}

// Measures all the values available to us (at least those that I know of)
void measure(void)
{
    sensors_1.requestTemperatures(); // Get temperature from Dallas sensor

    voltageV1 = analogRead(A0);        // Get the voltage on analog pin
    voltageV1 = voltageV1*10.0/1024;

    voltageV2 = analogRead(A1);        // Get the voltage on analog pin
    voltageV2 = voltageV2*10.0/1024;

    temperature1=sensors_1.getTempCByIndex(0); // Save the temperature value on
    global variable
}

// Send data with MQTT to IBM Cloud IoT platform
void send_data(void)
{
    createJSON(); // Set up the data to be sent

    // Send data through serial to the client
    Serial.println(JSON_Data); // Note: needs to be Serial.println !!! In the cli-
    ent (one running the python code),
                                // it is used as a way to determine when there is
    no more data to be sent.
}

// A human and Arduino friendly way to create JSON documents in Arduino
void createJSON(void)
{
    StaticJsonBuffer<200> jsonBuffer; // Allocate JSON buffer with 200-byte pool

    JsonObject& rootJsonObject = jsonBuffer.createObject();
    JsonObject& nestedJsonObject_STATE = rootJsonObject.createNestedOb-
    ject("state"); // create JSON object state - required by AWS
    JsonObject& nestedJsonObject_DESIRED = nestedJsonObject_STATE.createNestedOb-
    ject("desired"); // create JSON object desired, under state - required by AWS

    // Add following data to object desired
    nestedJsonObject_DESIRED["sensor"] = "Dallas"; // Create JSON object named
    "Sensor", assigned with the name of our sensor
    nestedJsonObject_DESIRED["temperature"] = temperature1; // Create JSON object
    named "Temperature", assigned with our temperature data

    JsonArray& nestedJsonArray_VOLTAGE = nestedJsonObject_DESIRED.createNestedAr-
    ray("voltage"); // Creates an array of data with name Voltage
    nestedJsonArray_VOLTAGE.add(voltageV1); // Add V1 to the array
    nestedJsonArray_VOLTAGE.add(voltageV2); // Add V2 to the array

    rootJsonObject.printTo(JSON_Data);
}

// The order of methods to handle data
void workflow(void)
{
    measure(); // First measure and save the data

```

```
    print_measurement(); // Print them on the LCD screen  
    send_data();         // Publish data to AWS  
}
```

ANNEX 2. Python code

```
# This code is used to retrieve data from Arduino Mega that is connected through
serial port and the data
# itself is being sent to Amazon Web Services (AWS) Internet of Things (IoT)
service. As the data is
# sent in JSON format (using ArduinoJson) all we have to do, is make sure that it
is in the correct format
# for the AWS IoT Python SDK. The data is being sent every minute and during
waiting the port is closed.

# Note: Edit callbacks.py - dataNames variable if your data is in different format
It is used to define
# the objects in your JSON document, that will be printed on console after each
successful update.
# Note 2: For security reasons private keys and certificates have been deleted from
the login folder and
# API endpoint name is replaced with xxxxxx

import time
import serial
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
from callbacks import customShadowCallback_Update, customShadowCallback_Delete

# ***** AWS details
# ***** #
# root and device certificate, alongside with device private key should be put in
the login folder
loginFolder = "login/"

# Paths to the certificates, rename the certificates accordingly.
# Note: Make sure there is no spaces in the name
rootCertificatePath = loginFolder + "rootCertificate.pem"
privateKeyPath = loginFolder + "MyThermometer001.private.key"
certificatePath = loginFolder + "MyThermometer001.cert.pem"

appName = "testIoTPySDK" # Name of the app, can be anything.
hostname = "xxxxx.iot.eu-west-2.amazonaws.com" # Your API endpoint.
port = 8883 # 8883 when connecting with MQTT, 8443 for HTTP, 443 for WebSocket and
HTTP.
thingName = "MyThermometer001" # Name of your thing (i.e. device) on AWS
topicToPublish = "$aws/things/" + thingName + "/shadow/update"
#
# ***** #

# Create an AWS IoT MQTT Client using TLSv1.2 Mutual Authentication
myAWSIoTMQTTShadowClient = AWSIoTMQTTShadowClient(appName)
myAWSIoTMQTTShadowClient.configureEndpoint(hostname, port)
myAWSIoTMQTTShadowClient.configureCredentials(rootCertificatePath, privateKeyPath,
certificatePath)

# AWS IoT MQTT Client connection configuration
myAWSIoTMQTTShadowClient.configureAutoReconnectBackoffTime(1, 32, 20)
myAWSIoTMQTTShadowClient.configureConnectDisconnectTimeout(10) # 10 sec
myAWSIoTMQTTShadowClient.configureMQTTOperationTimeout(5) # 5 sec

myAWSIoTMQTTShadowClient.connect() # Connect to AWS IoT
```



```

deviceShadowHandler =
myAWSIoTMQTTShadowClient.createShadowHandlerWithName(thingName, True)

# Delete shadow JSON doc.
# Note: If there is no data to be deleted, it will be rejected (doesn't crash/st
the program)
deviceShadowHandler.shadowDelete(customShadowCallback_Delete, 5)

def main():
    """ Get data from Arduino and sent it to AWS IoT """

    # Establish serial connection.
    try:
        arduinoSerial = serial.Serial("COM5", timeout=1, baudrate=9600)
    except:
        print("Unable to establish serial connection with Arduino. Check the
port.")

    time.sleep(2) # Give time for arduino to wake up.

    # Keep looping while the port is open.
    while arduinoSerial.isOpen():
        arduinoSerial.write(1) # Tell Arduino to send data.
        JSON_Data = arduinoSerial.readline().strip() # Get the data from arduino
serial
        # As the data is in bytes, next we make sure it is in the right format a
upload the data to AWS
        for data in JSON_Data.decode('utf-8').split('\n'):
            deviceShadowHandler.shadowUpdate(str(data),
customShadowCallback_Update, 5)
        arduinoSerial.close() # Close the port
        time.sleep(58) # Wait for 58 seconds before uploading again

    try:
        arduinoSerial.open() # Open the port again.
    except:
        print("Unable to open port.")
        time.sleep(2) # Give time for arduino to wake up.

if __name__ == "__main__":
    main()

```

```

import json

# Insert the name fields of your JSON data into the following array.
# They are used to dynamically print the sent data in
customShadowCallback_Update method.
dataNames = ["sensor", "temperature", "voltage"]

# Custom Shadow callback
def customShadowCallback_Update(payload, responseStatus, token):
    # payload is a JSON string ready to be parsed using json.loads(...)
    if responseStatus == "timeout":
        print("Update request " + token + " time out!")
    if responseStatus == "accepted":
        payloadDict = json.loads(payload)
        print("~~~~~")
        print("Update request with token: " + token + " accepted!")

        # for dynamically printing sent JSON data.
        for i in range(0, len(dataNames)):
            print(dataNames[i] + ": " +
str(payloadDict["state"]["desired"][dataNames[i]]))

        print("~~~~~\n\n")
    if responseStatus == "rejected":
        print("Update request " + token + " rejected!")

def customShadowCallback_Delete(payload, responseStatus, token):
    if responseStatus == "timeout":
        print("Delete request " + token + " time out!")
    if responseStatus == "accepted":
        print("~~~~~")
        print("Delete request with token: " + token + " accepted!")
        print("~~~~~\n\n")
    if responseStatus == "rejected":
        print("Delete request " + token + " rejected!")

```