Cloud Computing 2022/23 spring

Practice 11 - IoT device integration with cloud services

In this lab, you will learn how to set up Cloud service sof IoT device and data management, We will create an IoT Hub service in Azure, create a simple Python-based IoT device software and integrate it with Azure IoT hub.

References

- Python guide for Azure IoT Hub: https://learn.microsoft.com/en-us/azure/iot-hub/device-management-python
- Creating IoT devices: https://learn.microsoft.com/en-us/azure/iot-hub/iot-hub-create-through-portal#register-a-new-device-in-the-iot-hub
- Provisioning devices: https://learn.microsoft.com/en-us/azure/iot-dps/how-to-legacy-device-symm-key?tabs=linux&pivots=programming-language-python
- Python examples: https://github.com/Azure/azure-iot-sdk-python/tree/main/samples
- Python AMQP examples: https://github.com/Azure/azure-uamqp-python/tree/main/samples

PS! Code examples from these references have been used as source when designing this lab exercises

Exercise 11.1 Creating an IoT Hub service in Azure

We will configure the Azure IoT hub and register a new device record inside it.

Create a new IoT hub Service:

- Search for IoT Hub service
- Region: North EU
- Set a unique name which contains your last name
- Tier: Basic

Create a new IoT device:

- Open the Devices page under IoT Hub
- Add device
- Specify a device ID, which contains your last name
- Keep Authentication type: Symmetric key

Because we are using the free Basic tier, some of the functionality is disabled:

- We can not update the software running on the device from the cloud
- We can not send operations from the cloud to the device.

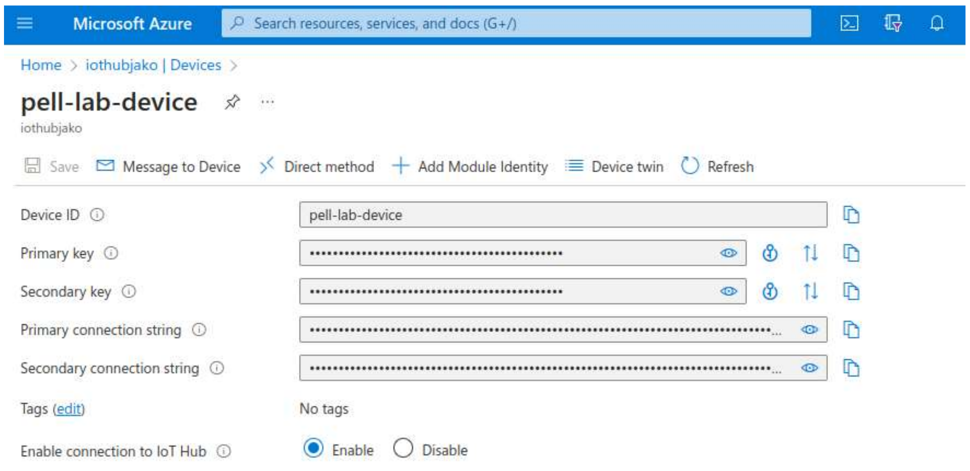
In the next task, we will create simple Python software for this new IoT device.

Exercise 11.2 Creating a new Python-based IoT device

1. Create a new Python project (e.g. in Pycharm)
2. Create a new method (or a module) named `generate_sas_token`
   - Take the method implementation from this guide: https://learn.microsoft.com/en-us/azure/iot-hub/iot-hub-dev-guide-sas?tabs=python
3. We are going to use AMQP protocol, but instead of the pika Python client (which we used for RabbitMQ), we are going to use the **uamqp** Python library
   - Import it using the code: `import uamqp`
   - Also import libraries:
     - uuid, urllib, json, os

Let's set up some Python variables:

- `iot_hub_name` - value should be the name of your IoTHub
- `hostname` - value should be the hostname of your IoTHub
  - e.g.: `iothubjako.azure-devices.net`
- `device_id` - value should be the ID of the IoT device you created inside your hub
- `username` - value should be in the format of: `{device_id}@sas.{iot_hub_name}`
  - e.g.: `pelledevice@sas.iothubjako`
- Read access key value from environment variables
  - access_key = os.getenv('ACCESS_KEY')
  - Set up ACCESS_KEY variable in environment variables
    - In Pycharm, you can configure it in run configuration - when executing the script. Under environment variables.
  - You can copy the ACCESS_KEY the value from the device configuration (Both primary and secondary keys are suitable) inside the Azure IoT Hub.



Device ID: pell-lab-device
Primary key: ••
Secondary key: ••
Primary connection string: ••...
Secondary connection string: ••...
Tags (edit): No tags
Enable connection to IoT Hub: [✓ Enable] Disable

- Let's now generate a sas token using the previously created `generate_sas_token` method:

  -
    ```
    sas_token = generate_sas_token('{hostname}/devices/{device_id}'.format(
        hostname=hostname, device_id=device_id), access_key, None)
    ```
- Define the message path to be `'/devices/{device_id}/messages/events'` inside the broker:

  -
    ```
    operation = '/devices/{device_id}/messages/events'.format(
        device_id=device_id)
    ```
- We will send messages there. This is the default message path for incoming telemetry messages from an IoT device.
- Define the full AMQP path:

  -
    ```
    uri = 'amqps://{}:{}@{}{}'.format(urllib.parse.quote_plus(username),
                                      urllib.parse.quote_plus(sas_token), hostname, operation)
    ```
    - It includes the username, authentication token, hostname of the service, and the message path.
      - @ character in username is quoted using `urllib.parse.quote_plus` method to avoid breaking the separation of username, password, and URL in the amqp URL.
      - We also use `urllib.parse.quote_plus` to escape any special characters inside the generated token string.

Now we can initiate the AMQP client and define the message to be sent from our IoT device.

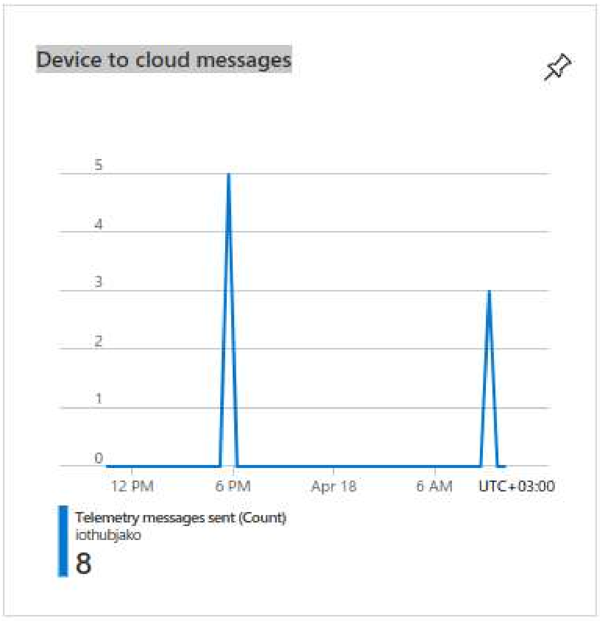
- Initiate the uamqp sender client:
  - `send_client = uamqp.SendClient(uri, debug=True)`
- Define message:
  - Create a new MessageProperties object for message metadata:
    - `msg_props = uamqp.message.MessageProperties()`
  - generate message id:
    - `msg_props.message_id = str(uuid.uuid4())`
  - create a new JSON message:
    - `message_json = {'temperature': 22 }`
  - Convert the message to a string object:
    - `msg_data = json.dumps(message_json)`
  - Define uamqp Message object:
    - `message = uamqp.Message(msg_data, properties=msg_props)`

Let's now send the message

- `send_client.send_message(message)`
- You can check/print the content of the results object to see if the sending of the message succeeded.
    - `None` is ok.
    - If the value is equal to `uamqp.constants.MessageState.SendFailed` variable value then sending failed.

You can verify from the IoT hub page whether the messages from the device have arrived.

- Check the "Device to cloud messages" statistics diagram on the Overview page of your IoTHub service. You should see at least one message arrival being visualized there.



Deliverable: Take a similar screenshot of a situation where you see that telemetry messages have arrived in IoT hub. Also, make sure the name of the IoT hub is visible from the screenshot (larger screenshot than the previous sample here)

Exercise 11.3 Getting access to the Device data

Let's create a Python script for listening to messages that our IoT device sends to the Event Hub

1. Create a new Python program/script in your project
    - Check this tutorial page: https://pypi.org/project/azure-eventhub/
        - And use the example code from the "Consume events from an Event Hub asynchronously" section as a basis for your Python script for listening to messages sent to IoT hub from the IoT device
2. Install the `azure-eventhub` python package in your Python project
3. Configure the following input variables for the script
    - **eventhub_name -** This is NOT IoT Hub name. You will find the event hub name from the "Built-in endpoints" page under "Event Hub-compatible name" field.
    - **Consumer group -** You will find the **name of the consumer group** under the "Built-in endpoints" page in IoT Hub, under Consumer Groups. Do NOT remove the "$" character.
    - **Connection String -** You will find under `Built-in endpoints -> Event Hub compatible endpoint`
        - Change the drop-down selection from iothubowner to service
        - NB! Do not copy the connections trying into code directly. Set its value using the Environment variable and use code like `CONNECTION_STR = os.getenv('CONNECTION_STR')` to access its value.
    - '* Otherwise, there is a significant risk that you will share the secret key when you submit the code.
4. Remove the parameter `starting_position` from the `client.receive` method call to receive only new messages.
5. Let's also print out the message content:
    - Add the following lines into the `on_event` method, which is executed on every arrived message:

```
    print("Telemetry received: ", event.body_as_str())
    print("Properties (set by device): ", event.properties)
    print("System properties (set by IoT Hub): ", event.system_properties)
```

Test that you are able to access the IoT data.

- Start the message listener process
- Use the IoT device to send data

Deliverable: Make a screenshot of a situation where you have two windows next to each other:

1. First one should show the running IoT device
2. Second one should show the data arriving in the message listener output

Exercise 11.4 Update what data is sent by the IoT device

- Update the code of the IoT device to generate IoT data similar to the last Practice session, where we read in Puhatu IoT data CSV files and sent one JSON message every 10 seconds.
  - Use a similar approach to prepare the content of the JSON message.
  - NB! DO NOT send messages too frequently! And do not leave the code running for an extended period. Otherwise, there is a risk that you will use up all the daily free quota of your IoT hub.

Exercise 11.5 Routing IoT data to other Cloud services

IoT hub mainly works as an integration platform for IoT devices and their arriving data. Data itself should be stored and analyzed by other cloud services. In the Hub, we can set up different routing rules to route data to other cloud services. In this task, we will create a routing, which stores the arriving IoT data in Azure Storage account.

1. Make sure the resource group you are using (Inside the one you created the IoT hub) contains a storage account Azure service.
   - If not, create one.
2. Create a new storage container inside the storage account
   - Name it: `iot-data`
3. In the IoT Hub, create a binding to store messages in some other azure service
   - Open the **Message routing** page
   - Add a new routing named **telemetry-to-storage**
   - Add a new Endpoint of Type: **Storage**
     - Endpoint name: *choose freely yourself*
     - Pick a container: Choose the same `iot-data` container you created earlier
     - File name format: *Can keep default*
       - This defines which sub folders will be created for storing the message files
     - Encoding: *Change this to JSON, so it is easier for us to read the content of the files*
     - Authentication type: *keep Key-based*

NB! If the platform now offers you to restore the original message routing, do so.

- Test the routing works by creating a new message from IoT device and checking the content of the `iot-data` container.

Deliverable: Make a screenshot of a situation where see objects created inside the `iot-data` container

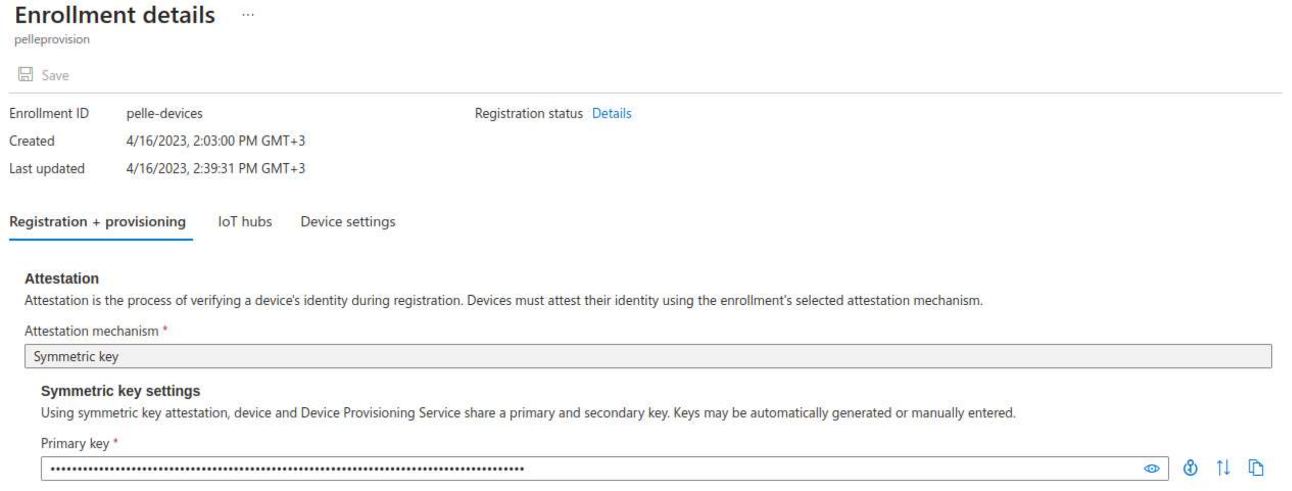
Exercise 11.6 Automate the IoT device registration

Let's now create a new version of our IoT device, which uses Azure SDK and is able to automatically join into the IoT Hub - without having to create its ID and credentials manually. We will use this to create multiple IoT devices.

1. Create a new "Azure IoT Hub Device Provisioning Services" service
   - make sure it uses the same Resource Group and Region as the IoT Hub
2. Wait for the resource to be created
3. Under the **Linked IoT Hub** settings page of the Provisioning Services
   - Add a new link - to your existing IoT hub
4. Under the **Managed enrollments** settings page
   - Create a new enrollment group
     - Attestation mechanism: Symmetric Key
     - Freely choose a group name
   - Under Iot Hubs, make sure Your IoT Hub is selected

Let's generate a new key for an IoT device.

- This should be done outside the IoT device software for security considerations.
- Follow the `Derive a device key` step in this tutorial to generate new symmetric key for (each) new IoT device: https://learn.microsoft.com/en-us/azure/iot-dps/how-to-legacy-device-symm-key?tabs=linux&pivots=programming-language-ansi-c#create-a-symmetric-key-enrollment-group
  - REG_ID: it is the device ID that we want our device to have.
    - For example: jakovits-device-01
    - We would generate new ID's for every new device.
  - KEY: Symmetric key from your Provisioning Service.
    - You can find it under the ENROLLMENT Symmetric key you created inside the provisioning Service


Enrollment details
pelleprovision

Save

Enrollment ID pelle-devices
Created 4/16/2023, 2:03:00 PM GMT+3
Last updated 4/16/2023, 2:39:31 PM GMT+3
Registration status Details

Attestation
Attestation is the process of verifying a device's identity during registration. Devices must attest their identity using the enrollment's selected attestation mechanism.

Attestation mechanism *
Symmetric key

Symmetric key settings
Using symmetric key attestation, device and Device Provisioning Service share a primary and secondary key. Keys may be automatically generated or manually entered.

Primary key *
••

Create a new Python program.

1. Install `azure-iot-device` Python package inside your project
2. Import needed libraries:
   ```
   import asyncio
   import uuid
   from azure.iot.device.aio import ProvisioningDeviceClient
   from azure.iot.device.aio import IoTHubDeviceClient
   from azure.iot.device import Message
   import json
   ```
   - We are no longer using AMQP but rather use the Azure SDK.
3. Define the needed variables:
   - `provisioning_host = "global.azure-devices-provisioning.net"`
   - `id_scope` - You will find it under **ID Scope** on the Provisioning Service main Overview page
     - For example: `id_scope = "0ne009EE5E5"` (DO NOT USE this specific ID)
   - `request_key` - this is the symmetric key generated for this IoT device in the previous step.
     - NB! DO NOT write the key value into the script, but rather read it from the system environment variable or from the script command line parameters. This is needed for both security reasons and to be able to use the same script for multiple IoT devices later.
   - `device_id` - this is the IoT device ID used in the previous step.
     - NB! DO NOT write the device ID value into the script, but rather read it from the system environment variable or from the script command line parameters. This is needed for both security reasons and to be able to use the same script for multiple IoT devices later.

4. Create an asynchronous main() method
   - `async def main():`
     - Call this method from the main program script:

     -
       ```
       if __name__ == "__main__":
           asyncio.run(main())
       ```

5. Update the `main()` method in the following steps:
6. Let's create the provisioning client, which we use to register the device or check its registration status.

   -
     ```
         provisioning_device_client = ProvisioningDeviceClient.create_from_symmetric_key(
                 provisioning_host=provisioning_host,
                 registration_id=device_id,
                 id_scope=id_scope,
                 symmetric_key=request_key,
             )
     ```

7. Let's initiate and wait for the registration to finish:
   - `registration_result = await provisioning_device_client.register()`
   - We can check for the registration state by using:
     ```
         if registration_result.status == "assigned":
             print("Registration succeeded")
     ```

8. Inside the IF statement block, if the registration succeeded, let's create a new device client object:

   -
     ```
     device_client = IoTHubDeviceClient.create_from_symmetric_key(
                 symmetric_key=request_key,
                 hostname=registration_result.registration_state.assigned_hub,
                 device_id=registration_result.registration_state.device_id,
             )
             # Connect the client.
             await device_client.connect()
             print("Device connected successfully")
     ```

9. Create the message JSON, also generating an ID to a new message using the `uuid` library:

   -
     ```
             message_json = {'temp': 132}
             msg_data = json.dumps(message_json)

             message = Message(msg_data)
             message.message_id = uuid.uuid4()
     ```

        - NB! Modify this part accordingly to your previous code to change what is the content of the message (e.g. Puhatu data)
10. Send a message
    - `await device_client.send_message(message)`

Individual task

- Check that the messages arrive in the IoT Hub service
- Deploy and run multiple IoT devices with their own (different) ID's and keys.

Deliverable: Make a screenshot of a situation where it is visible that two different IoT devices are sending messages at the same time

- E.g., two terminals/windows side-by-side outputting results/logs
- Make sure the ID of the device is shown in the output. May have to add additional logging or printing statements.

Deliverables:

1. Screenshots from exercises: 11.2, 11.3, 11.5, 11.6
2. Code from exercises:
    1. 11.3 (Data listener)
    2. 11.4 (First IoT device after modifying what data is sent)
    3. 11.6 (Second, self-registering IoT device)
    - DO NOT include any Python virtual environment folders/files (e.g., `.env` folder content)

Task	Lab 11 - Azure IoT Hub
Current submission	ZIP (18:16 19.04.2023) If your homework consists of multiple files (for example HTML + CSS + JS) it should be archived before submitting.
File	Choose File No file chosen
Comment	

Submit

issues and possible solutions

- If you get an error `AttributeError: __aexit__` in Exercise 11.3
    - Try whether commenting out the following line helps:
        - `async with client:`
            - to `#async with client:`