Internet of Things Project

Repository: https://github.com/obremba/uni-iot-project

Created by Sergiusz Morga

Connection to the device (OPC UA server).

To connect to the server, the library (OPC UA client) **asyncua** was used, using it an instance of the Client class was created, with the specified server address contained in the configuration file.

Example configuration:

```
[FACTORY]
opcua_url = opc.tcp://localhost:4840/
```

After connecting to the server, we retrieve data on available devices using an instance of the Factory class.

```
async with Client(config.factory_url) as client:
  print("Połączono!")
  factory = await Factory(client).initialize()
  print("Pobieranie urządzeń...")
  for device in factory.devices:
      if not config.has_agent_connection_string(await device.name):
          config.set_agent_connection_string(
              await device.name,
              input(f"Podaj connection string dla urządzenia {await
device.name}:")
          )
      connection_string = config.get_agent_connection_string(await
device.name)
      print(f"Tworzenie instancji agenta dla urządzenia: {await
device.name}...")
      agent = Agent(device, connection_string)
      print("Instacja stworzona!")

      subscription: Subscription = await client.create_subscription(200,
agent)
      await subscription.subscribe_data_change(await
agent.observed_properties)

      subscriptions.append(subscription)
      agents.append(agent)

  while True:
      await asyncio.gather(*[task for agent in agents for task in
agent.tasks_list])
      await asyncio.sleep(1)
```

Agent configuration

The agent configuration process occurs after connecting to the OPC UA server and fetching the list of available devices. For each device, it is checked if its connection string is present in the configuration file, if it is missing, the user is prompted for this information.

Example configuration file:

```
[AGENT]
device 1 = <connection_string>
```

After retrieving the connection string information, an instance of the Agent class is created to which an instance of the Device class and Client class is passed

```
agent = Agent(device, connection_string)
```

`child` - obiekt Node pochodzący z biblioteki **asyncua**
`client` - instancja klasy Client z biblioteki **asyncua**

D2C Messages

The agent sends telemetry information and information about errors occurring to the IoT Hub. These messages are sent every 1 second.

An example message containing telemetry data:

```
{
  "body": {
    "ProductionStatus": 1,
    "WorkorderId": "609f96ef-fb12-4b43-8f6f-1500dd424c39",
    "GoodCount": 31,
    "BadCount": 1,
    "Temperature": 63.75163233783323
  },
  "enqueuedTime": "Mon Dec 26 2022 19:22:20 GMT+0100 (Central European
Standard Time)",
  "properties": {
    "message_type": "telemetry"
  }
}
```

An example of an error message:

```
{
  "body": {
    "device_error": 4
  },
  "enqueuedTime": "Mon Dec 26 2022 19:22:18 GMT+0100 (Central European
Standard Time)",
  "properties": {
    "message_type": "event"
  }
}
```


Device Twin

The following values are stored in device twin: `Last Maintanance Date`, `Last Error Date`, `Desired Production Rate`, `Reported Production Rate` and `Device Error`.

Example device twin:


```
{
    "deviceId": "device-iot-1",
    "etag": "AAAAAAAAAAY=",
    "deviceEtag": "MTUxMjkxMjk3",
    "status": "enabled",
    "statusUpdateTime": "0001-01-01T00:00:00Z",
    "connectionState": "Disconnected",
    "lastActivityTime": "2022-12-26T18:22:22.9784367Z",
    "cloudToDeviceMessageCount": 0,
    "authenticationType": "sas",
    "x509Thumbprint": {
        "primaryThumbprint": null,
        "secondaryThumbprint": null
    },
    "modelId": "",
    "version": 124,
    "properties": {
        "desired": {
            "production_rate": 45,
            "$metadata": {
                "$lastUpdated": "2022-12-24T02:50:44.5778393Z",
                "$lastUpdatedVersion": 6,
                "production_rate": {
                    "$lastUpdated": "2022-12-24T02:50:44.5778393Z",
                    "$lastUpdatedVersion": 6
                }
            },
            "$version": 6
        },
        "reported": {
            "error": 0,
            "last_error_date": "2022-12-26T19:22:18.603716",
```

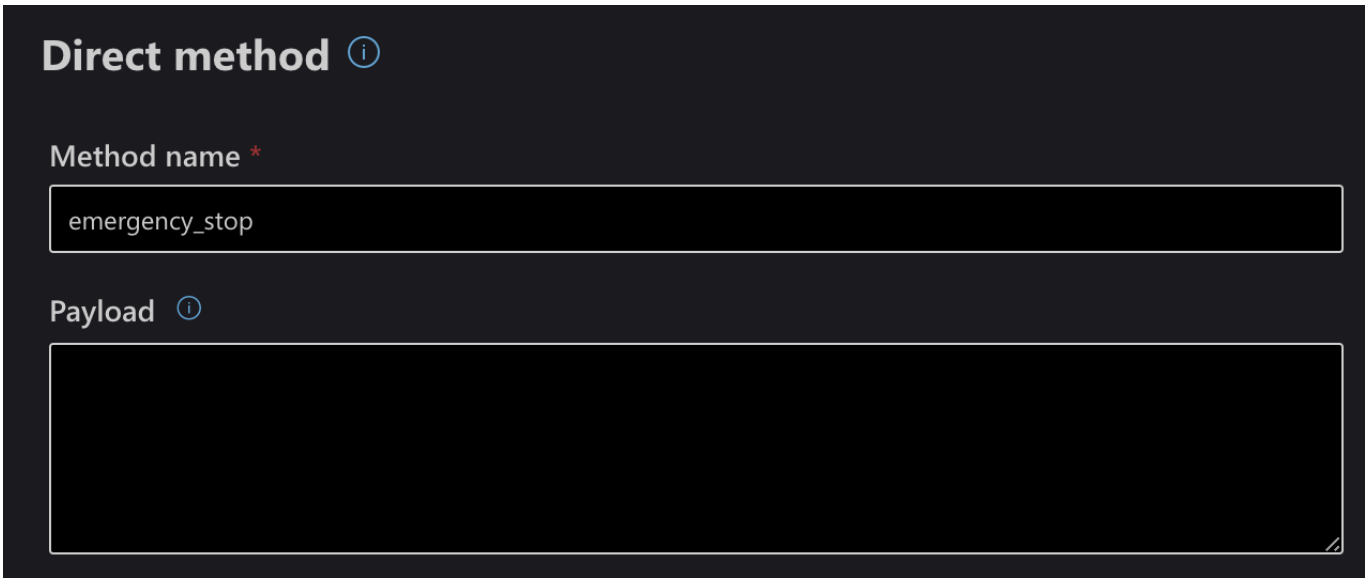
```
            "production_rate": 30,
            "device_error": 0,
            "$metadata": {
                "$lastUpdated": "2022-12-26T18:22:19.6345377Z",
                "error": {
                    "$lastUpdated": "2022-12-21T16:11:15.7267165Z"
                },
                "last_error_date": {
                    "$lastUpdated": "2022-12-26T18:22:18.6345197Z"
                },
                "production_rate": {
                    "$lastUpdated": "2022-12-26T18:21:59.9464415Z"
                },
                "device_error": {
                    "$lastUpdated": "2022-12-26T18:22:19.6345377Z"
                }
            },
            "$version": 118
        }
    },
    "capabilities": {
        "iotEdge": false
    }
}
```


Direct Methods

The agent has implemented support for 3 methods:

- emergency_stop
- reset_error_status
- maintenance_done

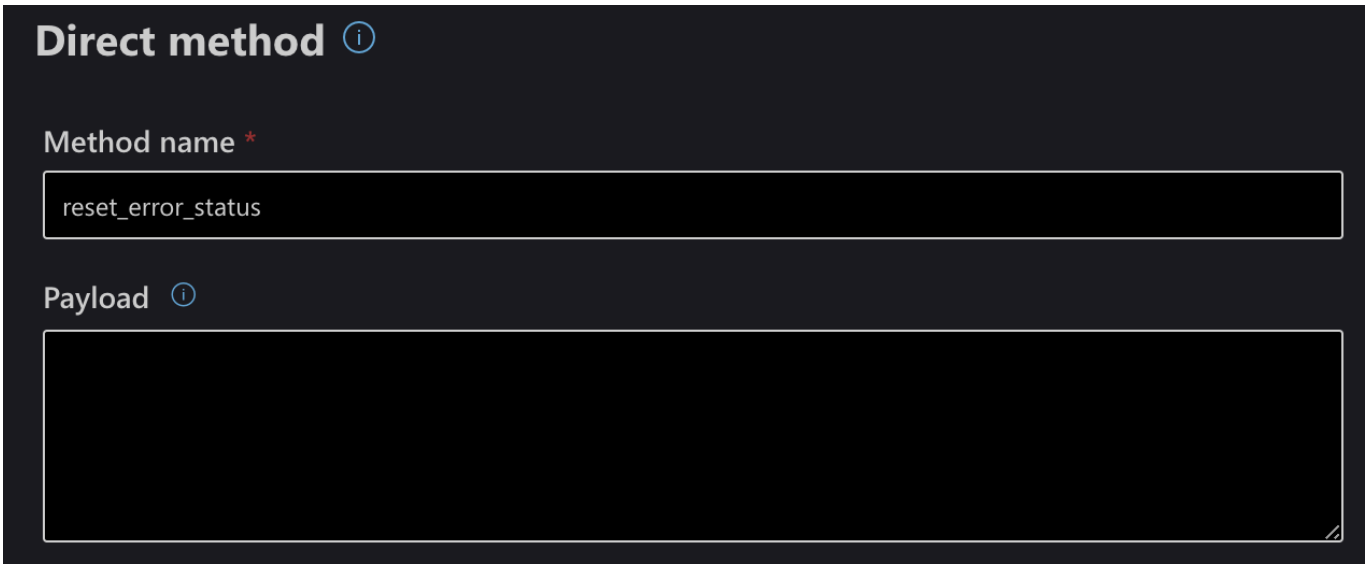
Example call of `emergency_stop` method:



Successfully invoked method 'emergency_stop' on device 'device-iot-1' response:

```
{
  "status": 200,
  "payload": null
}
```

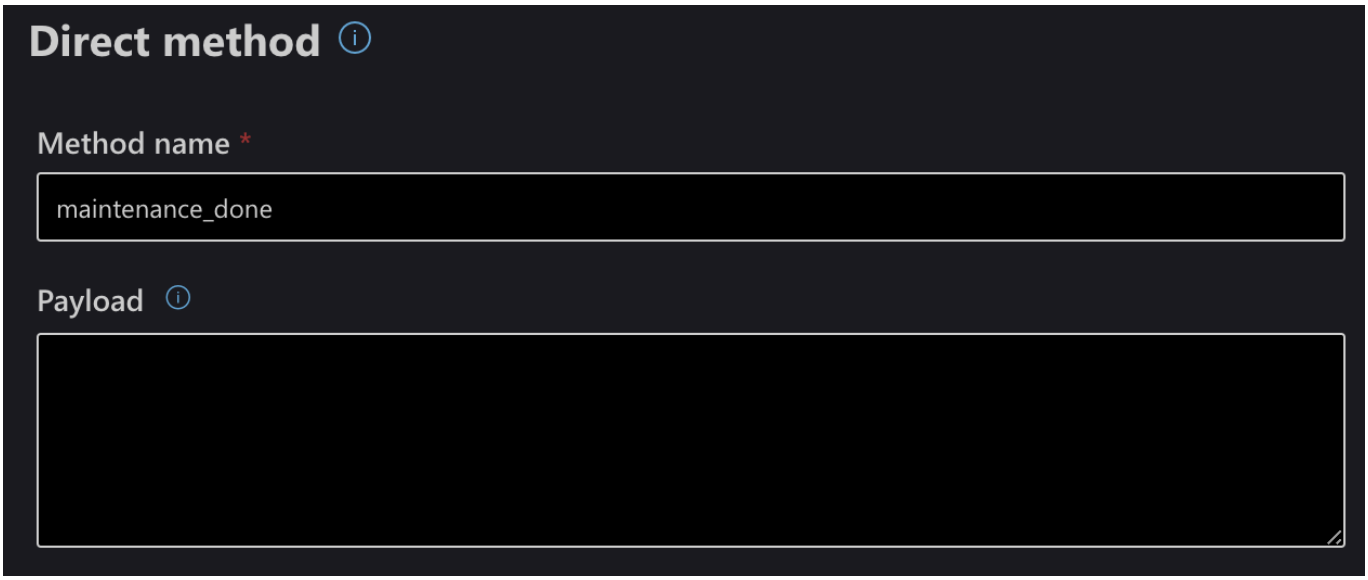

Example call of `reset_error_status` method:



Successfully invoked method 'reset_error_status' on device 'device-iot-1' response:


```
{
  "status": 200,
  "payload": null
}
```


Example call of `maintenance_done` method:



Successfully invoked method 'maintenance_done' on device 'device-iot-1' response:

```
{
  "status": 200,
  "payload": null
}
```

Implementation of calculation and business logic

Data calculations

Azure Stream Analytics was used to implement the calculations, whose source of information is the telemetry data sent D2C by the agent of a given device.

Calculations are performed using the following queries:

```
-- production per workorderId (sum of good and bad count)
SELECT
    WorkorderId,
    SUM(GoodCount) AS GoodCountSum,
    SUM(BadCount) AS BadCountSum,
    System.Timestamp() AS WindowEndTime
INTO asaGoodBadCount
FROM iothub TIMESTAMP BY EventEnqueuedUtcTime
GROUP BY
    WorkorderId, TumblingWindow(minute , 15)

-- production kpi
SELECT
    (SUM(GoodCount) / (SUM(GoodCount) + SUM(BadCount))) AS kpi,
    System.Timestamp() AS WindowEndTime
INTO asaProductionKPIs
FROM iothub TIMESTAMP BY EventEnqueuedUtcTime
GROUP BY
    TumblingWindow(minute , 15)

-- min, mean and max of temperature
SELECT
    WorkorderId,
    AVG(Temperature) AS AverageTemperature,
    MIN(Temperature) AS MinTemperature,
    MAX(Temperature) AS MaxTemperature,
    System.Timestamp() AS WindowEndTime
INTO asaMachineTemperatures
FROM iothub TIMESTAMP BY EventEnqueuedUtcTime
GROUP BY
    WorkorderId, TumblingWindow(minute , 5)

-- errors within 15 minutes
SELECT ih.IoTHub.ConnectionDeviceId as deviceId, COUNT(type) as errors
INTO asaErrorPerMachine
FROM iothub ih TIMESTAMP by EventEnqueuedUtcTime
```

```
WHERE message_type = 'event'
GROUP BY
    message_type, ih.IoTHub.ConnectionDeviceId, TumblingWindow(minute ,
15)
HAVING count(type) > 3
```

The results of queries, on the other hand, are stored in Blob Containers.

Business logic

The following services were used to compute data and call business logic:

- Azure Stream Analytics
- Azure Functions

The following queries were used to calculate the data needed to monitor and call business logic:

```
--- emergency stop trigger
SELECT ih.IoTHub.ConnectionDeviceId as deviceId, COUNT(type) as errors
INTO asaEmergencyStopTrigger
FROM iothub ih TIMESTAMP by EventEnqueuedUtcTime
WHERE message_type = 'event'
GROUP BY
    message_type, ih.IoTHub.ConnectionDeviceId, TumblingWindow(minute ,
15)

-- production kpi trigger
SELECT
    (SUM(GoodCount) / (SUM(GoodCount) + SUM(BadCount))) AS kpi,
    System.Timestamp() AS WindowEndTime
INTO asaProductionKpiTrigger
FROM iothub TIMESTAMP BY EventEnqueuedUtcTime
GROUP BY
    TumblingWindow(minute , 15)
```

They direct their results to Azure Functions, which processes this data and, using C2D, calls the appropriate methods when the given conditions occur.