Case Study – Azure

Project Documentation

Spis treści

Main Class Logic

1. To connect to the **Opc Client**, a new instance of the **OpcClient** class is initialized using the specified server address stored in the Resources object. A connection to the OPC client server is established using the **Connect()** method. The **BrowseNode()** method of the opcClient object is used to view the **OpcObjectTypes.ObjectsFolder** node.

2. Initialized **RegistryManager** object is used the connection string stored in the Resources object. Then it uses the **GetDevicesAsync** method of the **RegistryManager** object to get the list of devices registered in the IoT Hub, limited to the maximum number specified in the Resources object. The code then goes through the list of devices and checks if the device ID contains the certain string. If so, the device ID is added to the list of **deviceIds**.

3. Initialized a new dictionary, which has a key of type **VirtualDevice** and a value of type **MachineData**, which is a public class used to store data read from the Opc client's in a readable, fast-access way. Then in loop is created a new **DeviceClient** object using the connection string and the current element in the **deviceIds** list. The device client is opens asynchronously. Then it initialized a new **VirtualDevice** object using the **device client** and the **opcClient** object. Handlers are initialized for the **virtual device** using the **machineId** property of the current item in the **machineDataList**. The **virtual device** and its corresponding **machineData** object are then added to the dictionary. Finally, the property **iotHubDeviceId** of the object **MachineData** is set to the current item in the list **deviceIds**.

4. For each key-value pair in the dictionary, the code does the following: Calls the **readNode** function and passes the value (of type **MachineData**) and **opcClient** as arguments. Calls the **SetTwinAsync** function on the key (of type **VirtualDevice**) and passes the values of **deviceErrors** and **productRate** values (of type **MachineData**) as arguments. If the **deviceErrors** value (of type **MachineData**) is greater than 0, which means that there are errors on the device. The function **reportNewError** is called for set error status at **Device Twin** and send it to the **IoT Hub**.

5. Then the fourth point is repeated every 5 seconds by reading the data from the device, if the device production status is active, the telemetry data are sent to the cloud with as a message, in addition, if new errors occur or their status changes, an error message is also sent to the cloud and the data are updated in the device twin.

Connection to the device (Opc UA server)

The Agent reads data from the device using a function **readNode** that takes as arguments an object of class **MachineData**(was described) and **OpcClient** and then writes the data in the current MachineData object by reading them from certain nodes on the device.

```
static void readNode(MachineData machineData, OpcClient client)
{
    machineData.productStatus = (int)client.ReadNode(machineData.machineId + NODE_PRODUCT_STATUS).Value;
    machineData.workorderId = (string)client.ReadNode(machineData.machineId + NODE_WORKORDER_ID).Value;
    machineData.goodCount = (int)(long)client.ReadNode(machineData.machineId + NODE_GOOD_COUNT).Value;
    machineData.badCount = (int)(long)client.ReadNode(machineData.machineId + NODE_BAD_COUNT).Value;
    machineData.temperature = (double)client.ReadNode(machineData.machineId + NODE_TEMPERATURE).Value;
    machineData.deviceErrors = (int)client.ReadNode(machineData.machineId + NODE_DEVICE_ERROR).Value;
    machineData.productRate = (int)client.ReadNode(machineData.machineId + NODE_PRODUCT_RATE).Value;
    machineData.readTimeStamp = DateTime.Now;
}
```

The data updating on the device takes place in a class **OnDesiredPropertyChanged** , which serves to track information in case of changes desired properties on Device Twin, to reduces the Machine production Rate and report new property to Device Twin.

```
private async Task OnDesiredPropertyChanged(TwinCollection desiredProperties, object userContext)
{
    Console.WriteLine($"\t{DateTime.Now}> Device Twin. Desired property change:\n\t{JsonConvert.SerializeObject(desiredProperties)}");
    string nodeId = (string)userContext;
    int newProdRate = desiredProperties["ProductionRate"];
    string node = nodeId + "/ProductionRate";

    OpcStatus result = opcClient.WriteNode(node, newProdRate);
    Console.WriteLine($"\t{DateTime.Now}> opcClient.WriteNode is result good: " + result.IsGood.ToString());
    TwinCollection reportedProperties = new TwinCollection();
    reportedProperties["DateTimeLastDesiredPropertyChangeReceived"] = DateTime.Now;
    reportedProperties["ProductionRate"] = desiredProperties["ProductionRate"];

    await client.UpdateReportedPropertiesAsync(reportedProperties).ConfigureAwait(false);
}
```

Example of logs. the Agent connected three devices from the Opc Server to three devices of the IoT Hub and updated their device twins.

```
24.06.2024 02:05:17> Program.Main()> Connected device: a1a1c244-a0dc-41a2-9457-9f4e972d7908

24.06.2024 02:05:17> Device Twin value was set.
24.06.2024 02:05:17> Program.Main()> Connected device: 6259d793-815e-4ab4-a970-adabd129a236

24.06.2024 02:05:17> Device Twin value was set.
24.06.2024 02:05:17> Program.Main()> Connected device: 7dc840d8-662e-49e5-891f-6cc2a147e902

24.06.2024 02:05:18> Device Twin value was set.
```

Configuration of the Agent

Project configuration file is Resourse.resx:

deviceConnectionString	<connection string>
iotDevicesMaxCount	100
opcClientServer	opc.tcp://localhost:4840/
ownerConnectionString	<connection string>

The way in which the Agent connects to the Opc Server and to the IoT Hub is described in the first section.

Device Twin

The new data of Production Rate, Device Errors and theirs last occur date is reported to Device Twin in **SetTwinAsync** or **UpdateTwinAsync** methods.

```
public async Task UpdateTwinAsync(int deviceError)
{
    var twin = await client.GetTwinAsync();
    Console.WriteLine($"{DateTime.Now}> Device Twin value was update.");
    Console.WriteLine();

    var reportedProperties = new TwinCollection();
    reportedProperties["DeviceErrors"] = deviceError;
    reportedProperties["LastErrorDate"] = DateTime.Now;

    await client.UpdateReportedPropertiesAsync(reportedProperties);
}
```

```
public async Task SetTwinAsync(int deviceError, int prodRate)
{
    var twin = await client.GetTwinAsync();
    Console.WriteLine();

    var reportedProperties = new TwinCollection();
    reportedProperties["DeviceErrors"] = deviceError;
    reportedProperties["ProductionRate"] = prodRate;
    if (deviceError != 0)
    {
        reportedProperties["LastErrorDate"] = DateTime.Now;
    }

    await client.UpdateReportedPropertiesAsync(reportedProperties);
    Console.WriteLine($"{DateTime.Now}> Device Twin value was set.");

}
```

Device Twin example:


```
{
    "deviceId": "device_1",
    "etag": "AAAAAAAAAAE=",
    "deviceEtag": "MjI5NzY1Nzg5",
    "status": "enabled",
    "statusUpdateTime": "0001-01-01T00:00:00Z",
    "connectionState": "Disconnected",
    "lastActivityTime": "2024-06-24T00:05:29.3544964Z",
    "cloudToDeviceMessageCount": 0,
    "authenticationType": "sas",
    "x509Thumbprint": {
        "primaryThumbprint": null,
        "secondaryThumbprint": null
    },
    "modelId": "",
    "version": 70,
    "properties": {
        "desired": {
            "$metadata": {
                "$lastUpdated": "2024-06-20T16:53:08.4989907Z"
            },
            "$version": 1
        },
        "reported": {
            "DeviceErrors": 0,
            "ProductionRate": 0,
            "LastErrorDate": "2024-06-24T01:38:49.8344861+02:00",
            "$metadata": {
                "$lastUpdated": "2024-06-24T00:05:18.1488238Z",
                "DeviceErrors": {
                    "$lastUpdated": "2024-06-24T00:05:18.1488238Z"
                },
                "ProductionRate": {
                    "$lastUpdated": "2024-06-24T00:05:18.1488238Z"
                },
                "LastErrorDate": {
                    "$lastUpdated": "2024-06-23T23:38:50.4394586Z"
                }
            },
            "$version": 69
        }
    },
    "capabilities": {
        "iotEdge": false
    }
}
```


Direct Methods

There are five Direct Methods implemented in the Agent.

```
public async Task InitializeHandlers(string userContext)
{
    await client.SetReceiveMessageHandlerAsync(OnC2dMessageReceivedAsync, userContext);

    await client.SetMethodHandlerAsync("EmergencyStop", EmergencyStopHandler, userContext);
    await client.SetMethodHandlerAsync("ResetErrorStatus", ResetErrorStatusHandler, userContext);
    await client.SetMethodHandlerAsync("DecreaseProductRate", DecreaseProductRateHandler, userContext);
    await client.SetMethodHandlerAsync("MaintenanceDone", MaintenanceDoneHandler, userContext);
    await client.SetMethodDefaultHandlerAsync(DefaultServiceHandler, userContext);

    await client.SetDesiredPropertyUpdateCallbackAsync(OnDesiredPropertyChanged, userContext);
}
```

The “**EmergencyStop**” and “**DecreaseProductRate**” are two methods that serve to call the method on the machine.

Their code:

```
private async Task<MethodResponse> DecreaseProductRateHandler(MethodRequest methodRequest, object userContext)
{
    string productionRate = "/ProductionRate";
    string deviceError = "/DeviceError";
    Console.WriteLine($"\t{DateTime.Now}> METHOD EXECUTED: {methodRequest.Name}");
    string nodeId = (string)userContext;
    int rate = (int)opcClient.ReadNode(nodeId + productionRate).Value;
    int error = (int)opcClient.ReadNode(nodeId + deviceError).Value;
    OpcStatus result = opcClient.WriteNode(nodeId + productionRate, rate - 10);
    Console.WriteLine(result.ToString());
    await SetTwinAsync(error, rate - 10);
    return new MethodResponse(0);
}
```

```
private async Task<MethodResponse> EmergencyStopHandler(MethodRequest methodRequest, object userContext)
{
    Console.WriteLine($"\t{DateTime.Now}> METHOD EXECUTED: {methodRequest.Name}");
    string nodeId = (string)userContext;
    object[] result = opcClient.CallMethod(
            nodeId,
            nodeId + "/EmergencyStop"
            );
    return new MethodResponse(0);
}
```

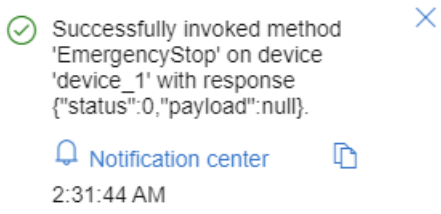
In userContext the device id that was initialized for each device in the main class:

```
var device = new VirtualDevice(deviceClient, opcClient);
await device.InitializeHandlers(machineDataList[i].machineId);
```

DecreaseProductRate and MaintenanceDone serve to update reported property in Device Twin.

```
private async Task<MethodResponse> MaintenanceDoneHandler(MethodRequest methodRequest, object userContext)
{
    Console.WriteLine($"\t{DateTime.Now}> METHOD EXECUTED: {methodRequest.Name}");

    var twin = await client.GetTwinAsync();

    var reportedProperties = new TwinCollection();
    reportedProperties["LastMainTenanceDone"] = DateTime.Now;

    await client.UpdateReportedPropertiesAsync(reportedProperties);

    Console.WriteLine($"\n{DateTime.Now}> Device Twin Maintenance Done.");
    Console.WriteLine();

    return new MethodResponse(0);
}
```

In the case of a call to a method with a uninitialized name, a default response is initiated that does nothing. In case of successful execution the methods return 0.



```
24.06.2024 02:31:44> METHOD EXECUTED: EmergencyStop
24.06.2024 02:32:14> METHOD NOT EXIST: EmergencyStart
```

Data Calculations

All data calculation take place in Azure Stream Analytics.

Stream input is **IoT Hub.**

Outputs are six Blob storages and two Service Bus queue.

Azure Stream Analytics queries:

1) 3 or more errors in the 15-minute window (per machine):

```
SELECT System.Timestamp() emergency_stop_time, machine_id, SUM(COALESCE(emergency_stop,0)) + SUM(COALESCE(power_failure,0))
+ SUM(COALESCE(sensor_failure,0)) + SUM(COALESCE(unknown,0)) as error_sum
INTO [asa-out-emergency-stops]
FROM [CwIOT]
GROUP BY machine_id, TumblingWindow(minute, 15) HAVING error_sum > 3
```

2) Percentage of good production in 15-minute window:

```
SELECT System.Timestamp() log_time, machine_id,
(MAX(good_count)*100)/(MAX(good_count)+MAX(bad_count)) as proc_of_good_production
INTO [asa-out-proc-good-count]
FROM [CwIOT]
GROUP BY machine_id, TumblingWindow(minute, 15)
```

3) Temperature per machine:

```
SELECT System.Timestamp() log_time, machine_id, MAX(temperature) as temperature_max, MIN(temperature) as temperature_min,
AVG(temperature) as temperature_avg
INTO [asa-out-temp]
FROM [CwIOT]
GROUP BY machine_id, TumblingWindow(minute, 5)
```

4) Production per workorder:

```
SELECT System.Timestamp() log_time, workorder_id, MAX(good_count) as good_count_sum, MAX(bad_count) as bad_count_sum
INTO [asa-out-aggreg-counts] FROM [CwIOT] WHERE
workorder_id IS NOT NULL GROUP BY workorder_id, TumblingWindow(minute, 30)
```

5) Count of every error type per machine (in the last 30 min):

```
SELECT System.Timestamp() AS log_time,
    machine_id,
    SUM(COALESCE(emergency_stop, 0)) AS emergency_stop_count,
    SUM(COALESCE(power_failure, 0)) AS power_failure_count,
    SUM(COALESCE(sensor_failure, 0)) AS sensor_failure_count,
    SUM(COALESCE(unknown, 0)) AS unknown_error_count
INTO [asa-out-errors]
FROM [CwIOT]
GROUP BY machine_id, TumblingWindow(minute, 30)
```

6) If percentage of good products falls below 90% in the 15-minute window, sending message to the Service Bus Queue:

```
SELECT System.Timestamp() time, deviceId INTO [device-decrease-production-rate]
FROM [CwIOT]
GROUP BY deviceId, TumblingWindow(minute, 15)
HAVING (MAX(good_count)*100)/(MAX(good_count)+MAX(bad_count)) < 90
```

7) If there are more then 3 errors in the 15-minute window for each machine, sending message to the Service Bus Queue:

```
SELECT System.Timestamp() emergency_stop_time, deviceId,
SUM(COALESCE(emergency_stop,0)) + SUM(COALESCE(power_failure,0)) +
SUM(COALESCE(sensor_failure,0)) + SUM(COALESCE(unknown,0)) as error_sum
INTO [device-errors-bus]
FROM [CwIOT]
GROUP BY deviceId, TumblingWindow(minute, 2) HAVING error_sum > 3
```

All logs are located in the "***Logs Examples of ASA jobs***" folder.

Function App

Function Apps solution was added to this project for executing triggers when there are new messages in device-decrease-production-rate or device-emergency-stop service bus queues. These functions executing direct methods on specified device for decreasing device's production rate or executing emergency stop if there is more, than 3 error within 15 minutes on device. This project uses Microsoft.Azure.Devices.CloudToDeviceMethod for calling methods on device. IoT Hub connection string is stored inside project's properties. Other connection data is stored in local.settings.json.

Example of received message

For emergency stop:

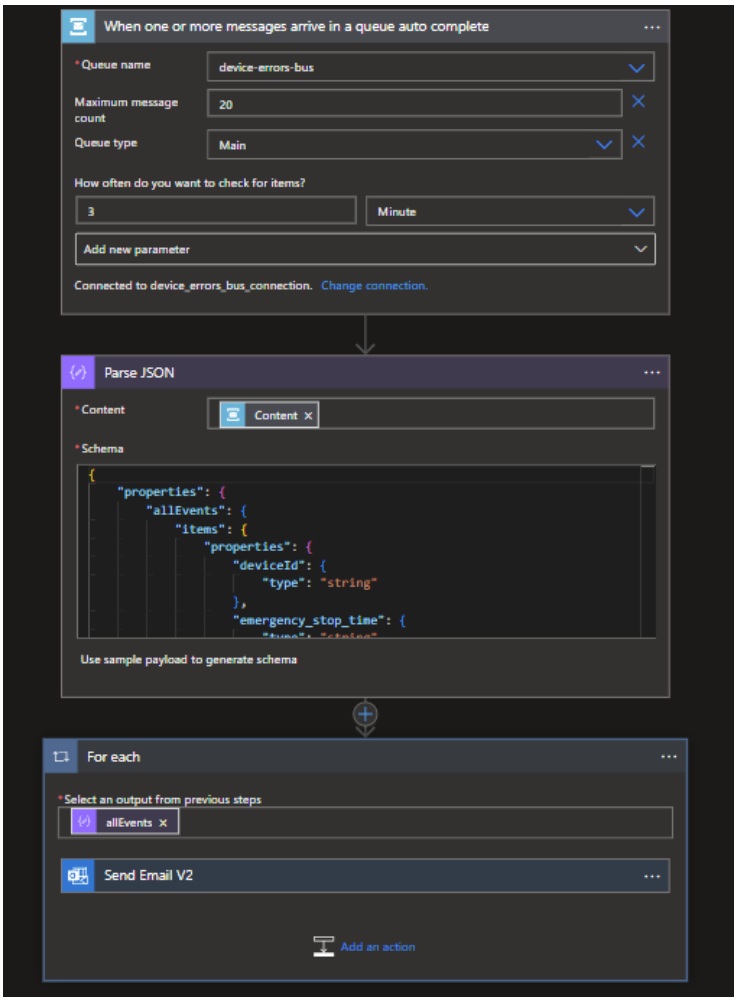
{"emergency_stop_time":"2024-06-23T20:15:00.0000000Z","machine_id":"ns=2;s=Device 1","error_sum":23.0}

For decrease prod. rate:

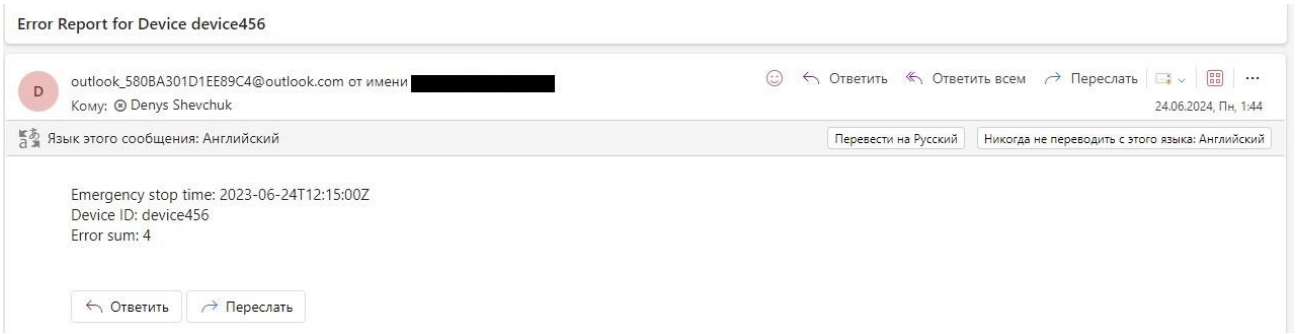
{ "time": "2024-06-24T15:00:00Z", "deviceId": "device123", "good_count": 80, "bad_count": 20 }

Logic App

“device-errors-bus” queue is also used as the input date source for the logic app, which has the main purpose of sending email, when the emergency stop occurs. The main code of the app is located in the folder “***Logic App***”.



Email example:



Business Logic