AWS IoT Core Connector

Device compatible with the MQTT protocol

- Name: **MQTTAWS**
- Version: 1.0.0.3
- Protocol: MQTT
- Interface: TCP/IP
- Runtime: .NET 2.0 (Multiplatform)
- Configuration:
    - Devices / Protocols

Overview
AWS IoT Core
Channels Configuration
Nodes Configuration
Points Configuration
Troubleshoot
Driver Revision History

Overview

Communication Driver Information	
Driver name	MQTTAWS
Assembly Name	T.ProtocolDriver.MQTTAWS
Assembly Version	1.0.0.3
Available for Linux	True
Devices supported	Any device compatible with the MQTT protocol
Manufacturer	Microsoft
Protocol	MQTT (Message Queuing Telemetry Transport) is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol.
Interface	TCP/IP
PC Requirements	Ethernet port
Implemented Methods	Connect, Disconnect and Publish.

AWS IoT Core

Overview

AWS IoT Core is a managed cloud service that enables communication between Internet of Things (IoT) devices and AWS cloud applications or services using the MQTT protocol, HTTPS, and WebSockets. It provides secure, bi-directional message exchange between connected devices and the AWS infrastructure.

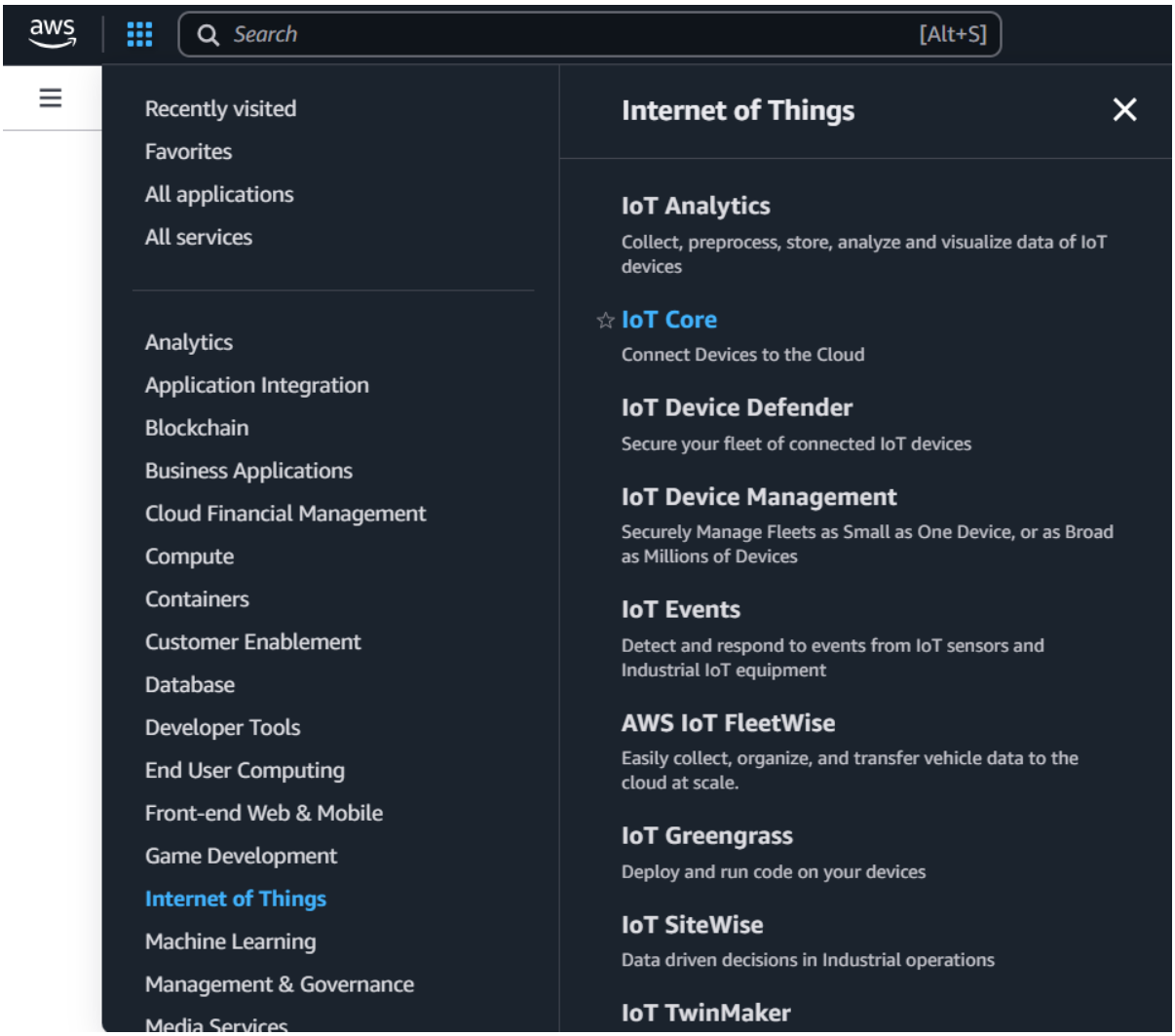
System Requirements

The requirements for a successful configuration of MQTT and AWS are listed below:

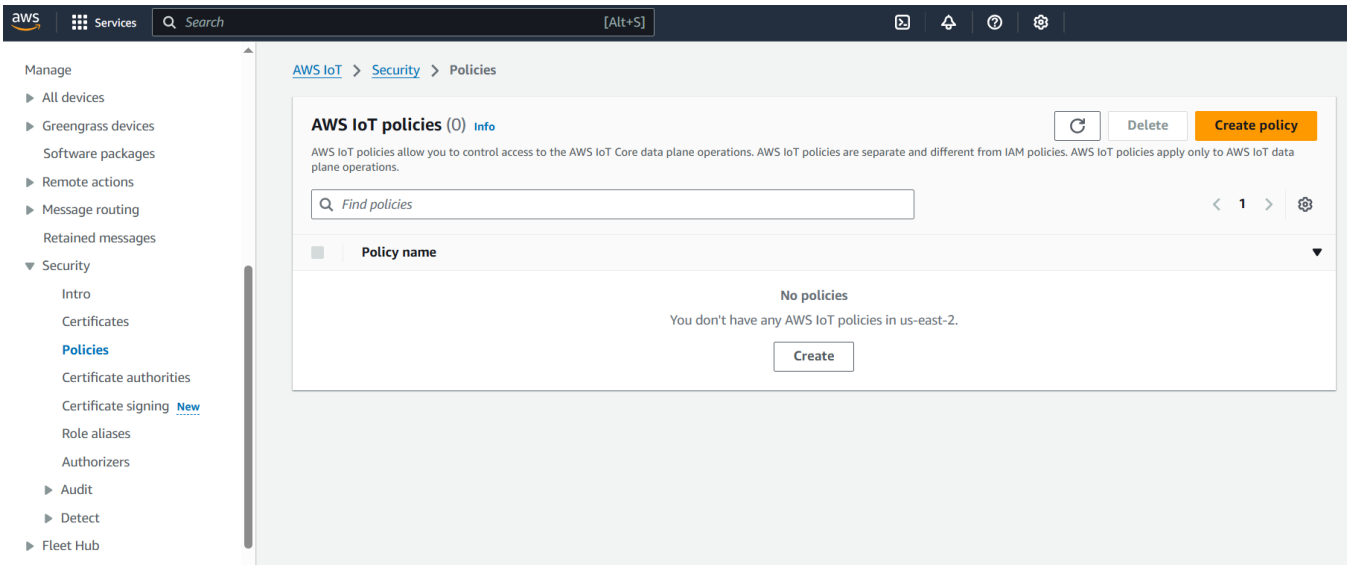
- Have an AWS Account - see here.

AWS IoT Core settings

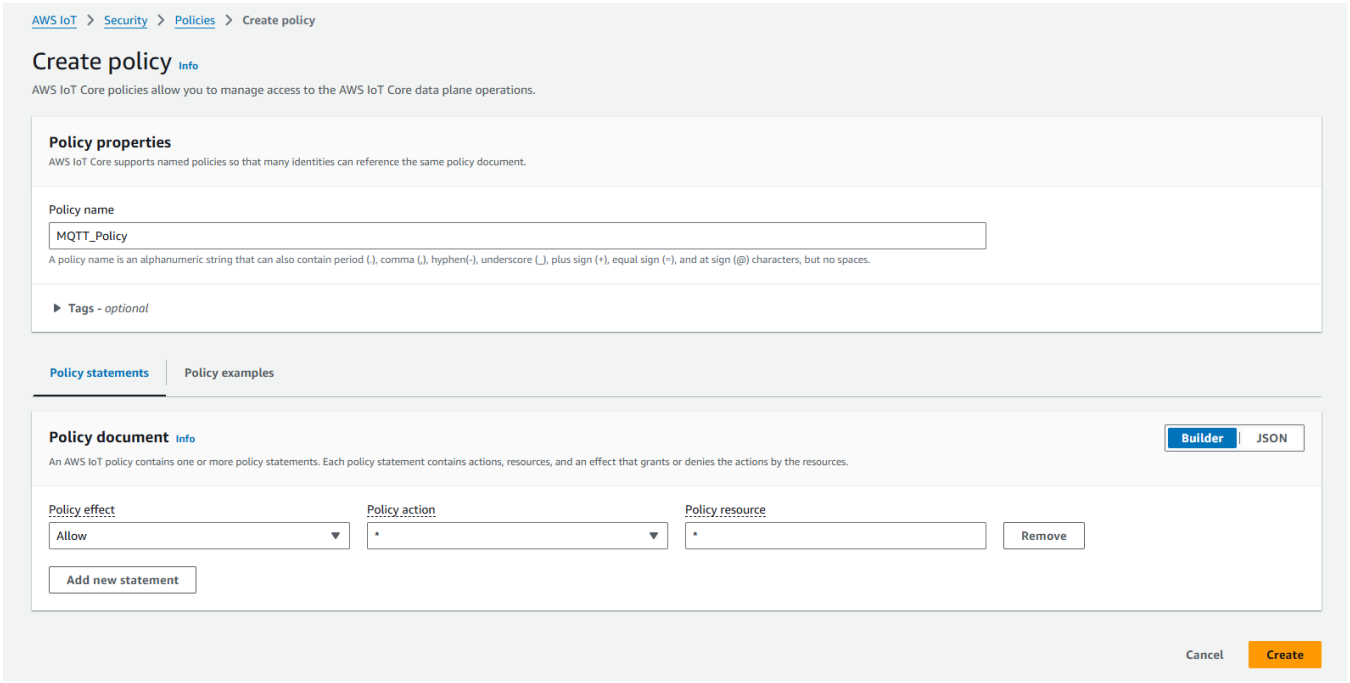
First, register a Thing in AWS IoT Core.
To do this, use an AWS account with all required registration information completed. Access the AWS Management Console and go to **Internet of Things > IoT Core**.



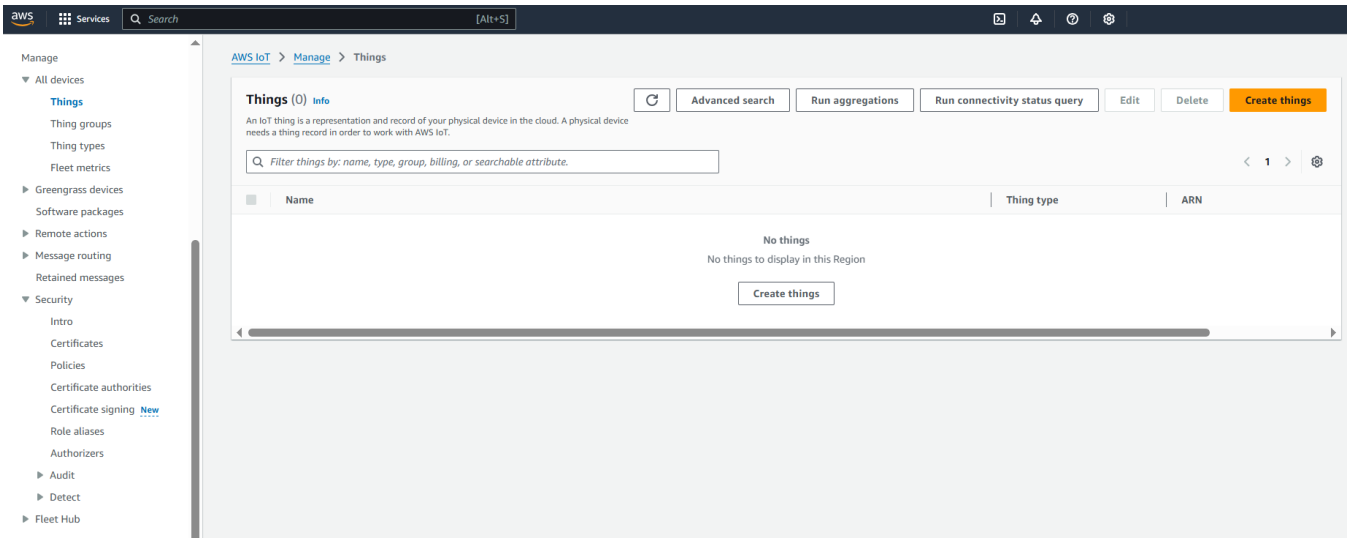
Expand the **Security** section and select **Policies**. Create a new security policy according to your requirements.



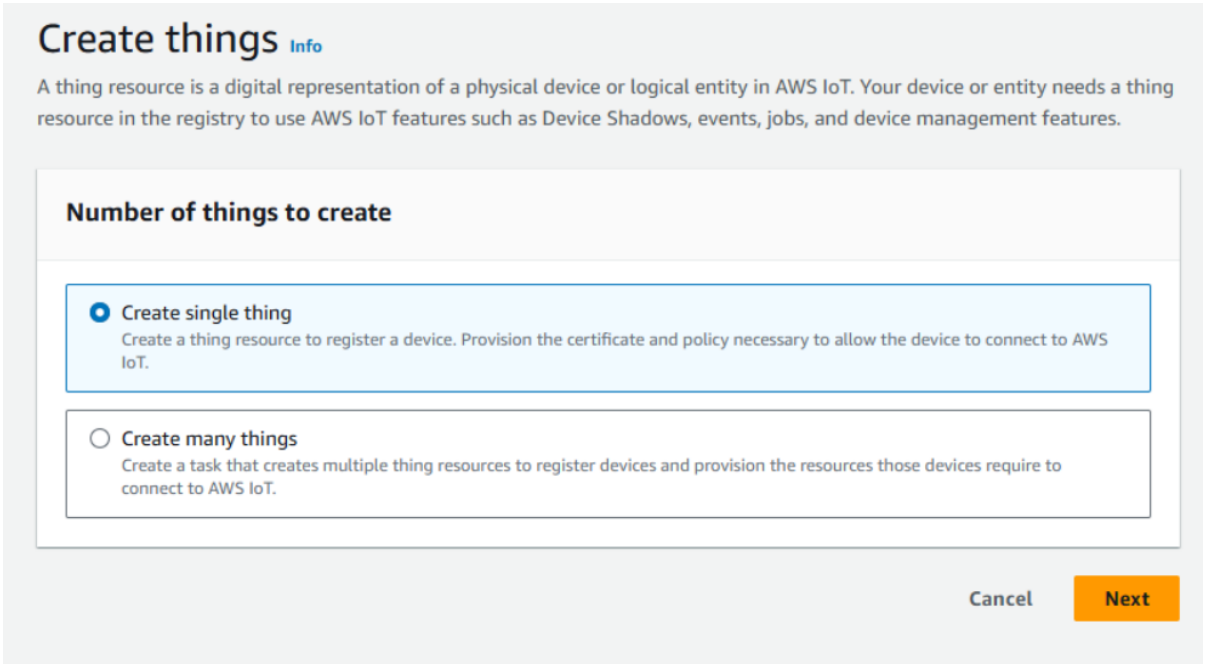
For testing purposes, you can create a security policy using * in both the **Policy action** and **Policy resource** fields. However, for production use, these fields must be defined more specifically according to application requirements.
Once the policy is properly configured, click **Create**.



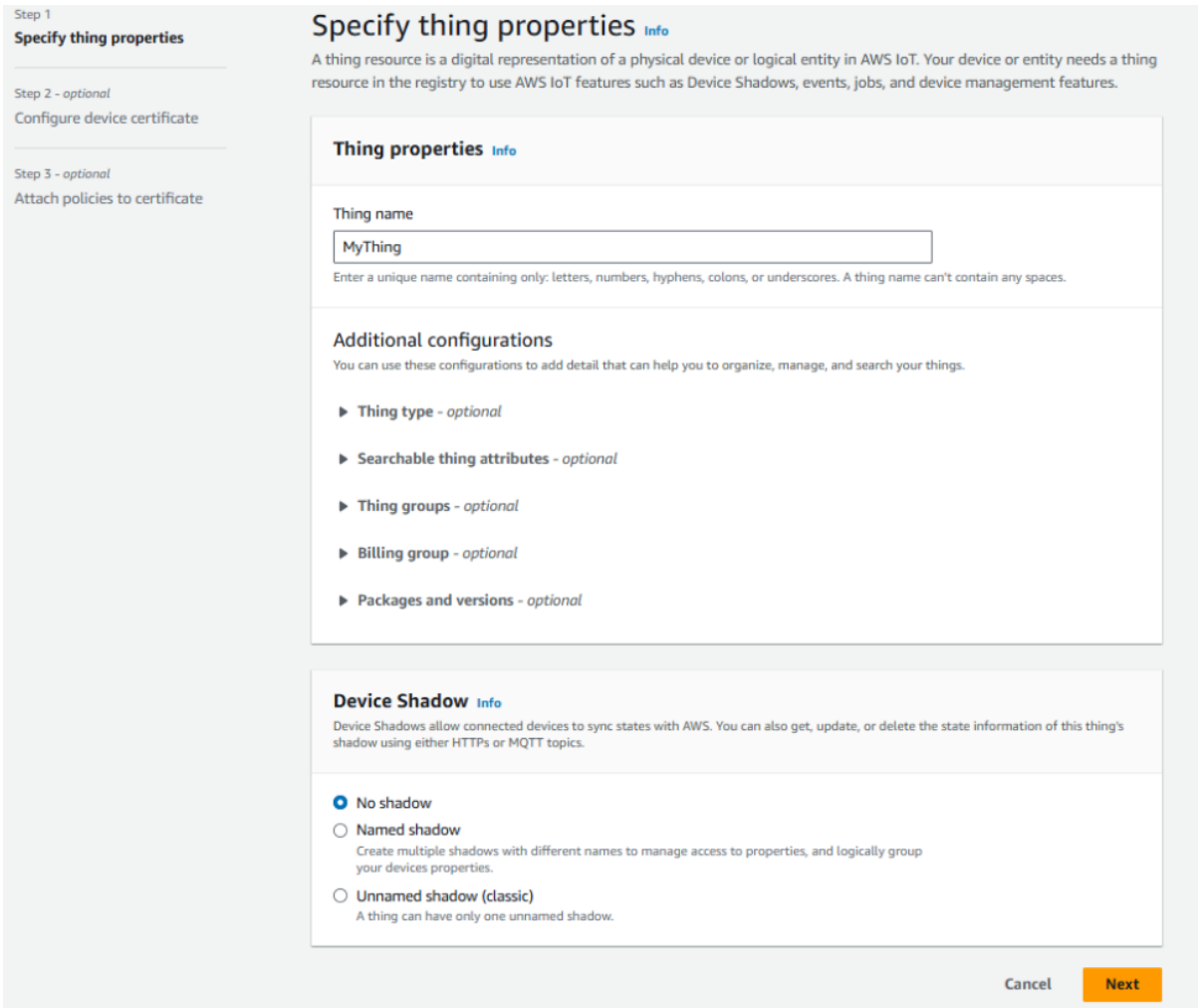
Next, proceed to create the Thing.
To do this, expand the **Manage** section, then navigate to **All Devices > Things**.



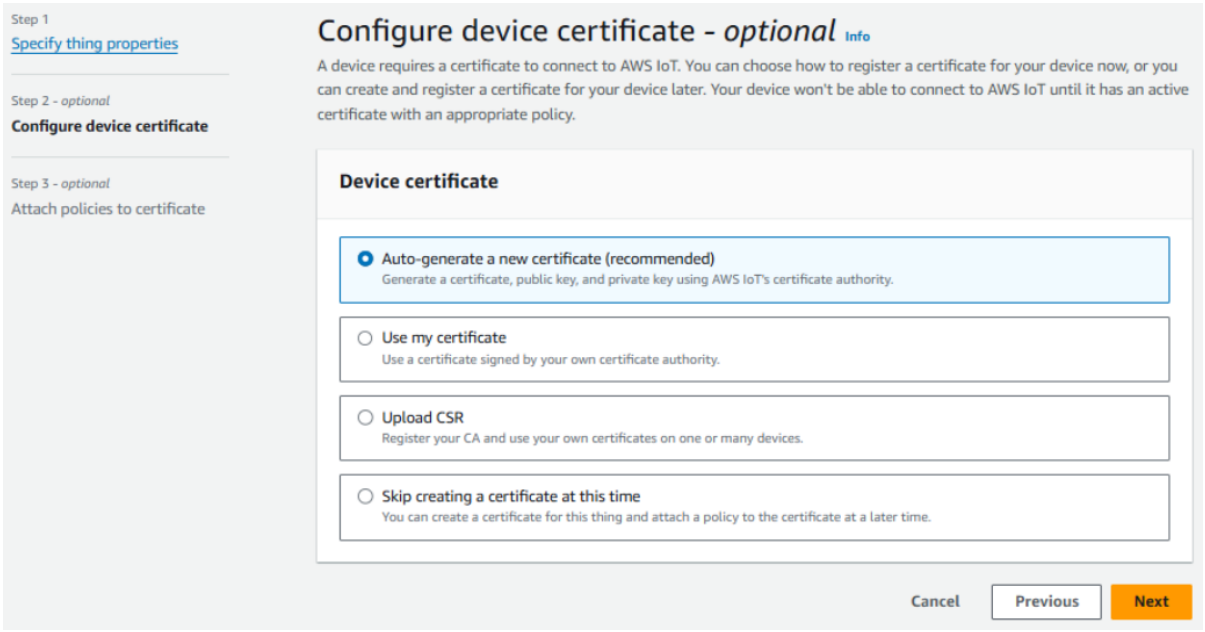
Create a new Thing.
Select the option to create a single Thing.



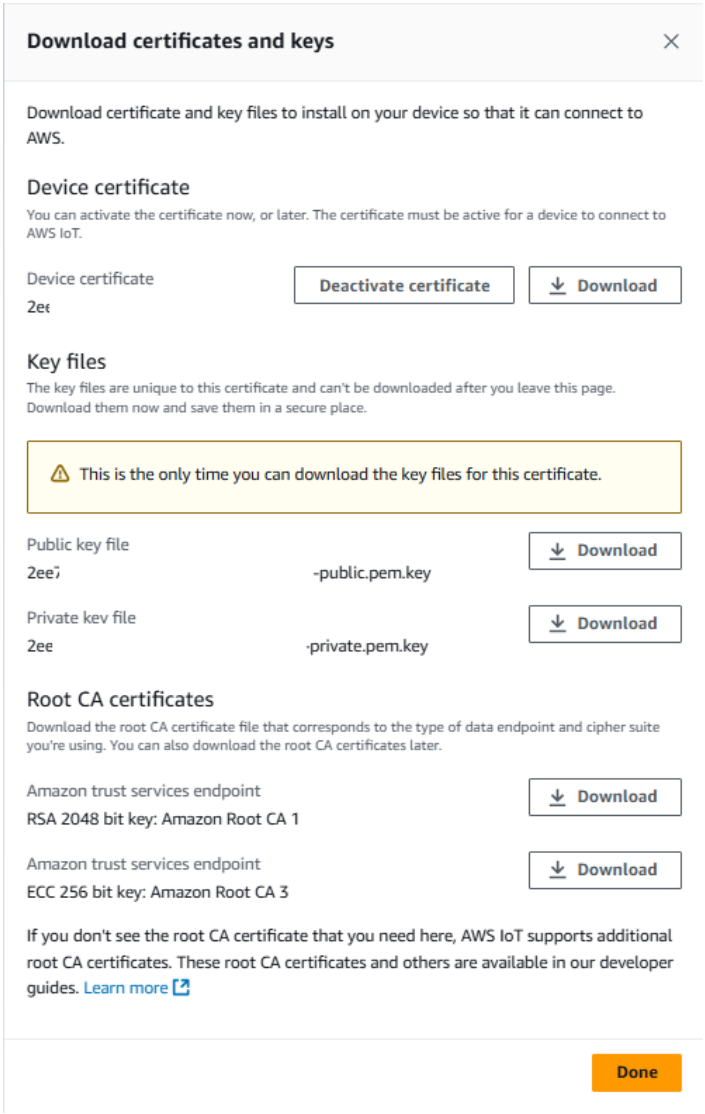
Specify the name of the Thing.



Specify the name of the Thing and select **Auto-generate a new certificate**.



Select the previously created security policy and click **Create Thing**.
After the Thing is created, download all certificates generated for it.



Creating a .pfx with password

The conversion to the `.pfx` format is required to ensure that the MQTT driver can be used across any device, regardless of whether the device is running .NET 8.0 or not. The `.pfx` format bundles the device certificate, private key, and any necessary root certificates into a single, encrypted file. This simplifies secure authentication and allows the driver to establish a trusted connection with the MQTT broker.

> **.NET 8.0**
>
> While the generated `.pfx` file can be used to connect the driver from any device, regardless of whether .NET 8 is used or not, the certificate conversion process requires executing **CertificateConverter.exe** in an environment with **.NET 8.0** installed.

After downloading the certificates for the Thing, use the `CertificateConverter.exe` utility located in the software installation directory: `"\Program Files\<Company>\<Product>\Utilities"`.

This utility converts the certificates generated for the Thing into a single `.pfx` file with a password.

To perform the conversion, execute `CertificateConverter.exe` from the command line with the following four parameters:

- /certificate - Path to the downloaded certificate file with the extension `-certificate.pem.crt;`
- /key - Path to the downloaded private key file with the extension `-private.pem.key;`
- /rootca - Path to the downloaded Amazon Root CA file, typically named `AmazonRootCA1.pem;`
- /password - New password to protect the `.pfx` file that will be generated.

With this setup, open a command prompt in the directory where `CertificateConverter.exe` is located.
Run a command similar to the following:

```
CertificateConverter.exe /certificate:"C:\my_folder\3e4...0e4-certificate.pem.crt" /key:"C:\my_folder\3e4...
0e4-private.pem.key" /rootca:"C:\my_folder\AmazonRootCA1.pem" /password:"12345"
```

After executing the command, the `.pfx` certificate file will be generated in the same folder as the source certificate files, such as `C:\my_folder` in the example above.

Channels Configuration

Protocol Options

Type: Defines how the channel works when connecting to the MQTT Broker.

Publisher: It is an MQTT client that publishes messages read from field equipment. It also can receive commands from Collectors.

TagProperties: Defines the tag properties beyond the **Value**. The names of the properties should be separated by a **comma ' , '**.

TimePublishRate: Indicates the time for publishing, in milliseconds. The messages that will be published to the MQTT Broker are grouped and sent using this period.

StoreAndForward: When this option is enabled, a database file, named as below, will be created at the same path of the solution file.

<SolutionName>.dbCh_N_<Device.Node.Id>_<GroupId>_<NodeId>

Where:

<Device.Node.Id> is the row value from the **ID** column in **Devices / Nodes**.

<GroupId> and **<NodeId>** come from the **Address** column in **Devices / Points.**

E.g.: MyMQTTSolution.dbCh_N_0_MyGroupID_MyNodeID.

All data will be stored in this database before forwarding to the Broker. The data that receives the Broker confirmation will be removed from this database. Then, when the communication with the Broker is lost, the data will stay in the local database until the connection is re-established to the Broker. The data-updating rate to the Broker is the time configured in the **TimePublishRate** field, even after a re-established connection and always using the first-in, first-out (FIFO) method to forward the data.

PublishDBIRTHAfterNBIRTH: Flag that indicates if a DBIRTH message will also be published when an NBIRTH message is requested.

UseParenthesesForArray: Flag that indicates if it should replace brackets by parentheses in the metric name.

PublishUDTsAsSimpleTags: When enabled, publishes User-Defined Types (UDTs) as individual tags rather than as a single complex object.

GroupID: Defines the identifier for the group to which the publisher belongs. This value helps organize and categorize publishers.

ForceMessageTimestamp: When enabled, it enforces that all tags within a BIRTH message have the same timestamp as the main timestamp. This applies exclusively to BIRTH messages. For other message types (e.g., data updates), the original behavior of timestamps remains unchanged. it enforces that all tags within a BIRTH message have the same timestamp as the main timestamp. This applies exclusively to BIRTH messages. For other message types (e.g., data updates), the original behavior of timestamps remains unchanged.

ForceCommandValue: When enabled, the Command received that are mapped to tags, the value is applied always as a new value, even it's the same of previous command. The effect of this configuration is that the Tag receiving the command value, will act as it had its Value Changed, there this value will be send to PLCs when used in WriteOnTagChange AccessTypes, or any other Module in the solution where it is used a Trigger, or tested for Value Change, the event will be processed as well.

Nodes Configuration

Station Configuration

Station syntax: ***<BrokerURL> ; <Port> ; <ClientID> ; <Username> ; <Password> ; <X509Certificate> ; <X509CertificatePassword> ; <NetworkSecurity> ; <WebSocket> ; <CollectorID> ; <QoS> ; <KeepAlive> ; <ReportOutOfRangeByException> ; <NodeID> ; <PayloadFormat>***

Where:

- ***<BrokerURL>****:* MQTT Broker (Server) name. It must be the domain name of your created 'thing' in AWS IoT Core.
- ***<Port>****:* MQTT Broker port. It must be the same *port* configured in the Broker to be listening. Default value is **8883**.
- ***<ClientID>****:* A unique identifier for the client connecting to the MQTT broker. If left empty, a unique identified client will be assigned. This user-defined string can also include the following macros, intermixed with the text:
  - _**Guid**_: Replaced by a dynamically generated Global Unique Identifier (GUID).
  - _**SolutionName**_: Replaced by the name of the solution (without the path or file extension).
  - _**NodeId**_: Replaced by the string defined as the NodeId.
  - _**GroupId**_ : Replaced by the string defined as the GroupId.

- ***<Username>**:* If applicable, is the username defined on MQTT Broker. This field is typically not required when using a `.pfx` certificate with a password.
- ***<Password>**:* Password defined on MQTT Broker. This field is typically not required when using a `.pfx` certificate with a password.
- ***<X509Certificate>**:* Path of `.pfx` certificate created with `CertificateConverter.exe`.
- ***<X509CertificatePassword>**:* Password created in certicate convert step with `CertificateConverter.exe`.
- ***<NetworkSecurity>**:* If the Broker works by using a certificate, this field is mandatory and must be the same as configured in MQTT. Options: None, Ssl2, Ssl3, Tls, Tls11, and Tls12 (*optional*). AWS IoT Core uses Transport Layer Security (TLS) to secure connections from IoT devices. The supported version for secure MQTT communication is **TLS 1.2**. This is the recommended and commonly used version when connecting to AWS IoT Core.
- ***<WebSocket>**:* Flag indicating if the connection to the MQTT Broker should be via WebSocket. Check how the MQTT Broker works before checking it (*optional*).
- ***<CollectorID>**:* If the channel works as Publisher, some Collector is active and other MQTT Brokers are also present in the network (redundancy, scalability, etc.,), this field is used to know that the MQTT Broker must be used. A Collector publishes its ID when connecting to MQTT Broker (optional).
- ***<QoS>**:* Quality of Service (QoS) is an agreement between the sender and the receiver of messages regarding the guarantees of delivering a message. There are 3 QoS levels:

  AtMostOnce: This service level guarantees a best-effort delivery. However, there is no guarantee of delivery, due to the recipient not acknowledging the receipt of the message.

  AtLeastOnce: This service guarantees that a message is delivered at least one time to the receiver. *Default Value*.

  ExactlyOnce: Highest level of service. It is safer, but slower when compared to the others. When the message flow is complete, all parties involved in the communication are sure the message was sent and delivered with success.

  Choosing the Right QoS Level

  Use QoS 0 when:
  - The connection between sender and receiver is stable (e.g., a test client or front-end application connected via a wired connection).
  - Occasional message loss is acceptable (e.g., when sending non-critical data at short intervals).
  - Message queuing is **not** required.

  Use QoS 1 when:
  - Every message must be received, but duplicates are acceptable.
  - The application **does not require** the overhead of QoS 2 but still needs message delivery guarantees.
  - Faster delivery is preferred compared to QoS 2.

  Use QoS 2 when:
  - Receiving **every message exactly once** is critical (e.g., when duplicate messages could cause issues).
  - Applications need the highest level of reliability, despite the additional overhead and latency.

- ***<KeepAlive>**:* Ensures that the connection between the Broker and Client is open and both parties are aware of the connection. Default is *10 seconds*.
- ***<ReportOutOfRangeByException>**:* When enabled, an item is published only when its value falls outside the Tag's ScaleMin / ScaleMax range.
- ***<NodeID>**:* NodeID name that will be used in the payload.
- ***<PayloadFormat>**:* Defines the payload format. The available options are **SparkplugB** and **Flat**.

All nodes of the same channel share all the device points regardless of which node they were configured for. Setting more than one node for the same channel should be used for MQTT Broker redundancy.

When using the Test button in a Linux targeted solution with TLS security, the certificate should be placed in the Solution's directory in both Windows and Linux environments (for Engineering and Runtime). **Do not** use the full file path.

Station Examples

***BrokerURL* =** agehf6lt8nnf1v-ats.iot.us-east-1.amazonaws.com

Port = 8883

Client ID = MyClientID

Username = <Blank>

Password = <Blank>

X509Certificate = C:\my_folder\3e4...0e4-certificate.pem.pfx

X509CertificatePassword = 12345

NetworkSecurity = Tls12

WebSocket = False

CollectorID = <Blank>

QoS = AtLeastOnce

KeepAlive = 10

ReportOutOfRangeByException = False

NodeID = NodeID

PayloadFormat = SparkplugB

Points Configuration

Address

The Address syntax is: **<DeviceID>**

Where:

- <**Device**>: Identifies ID of Device from Edge Node.

SparkblugB Payload Format

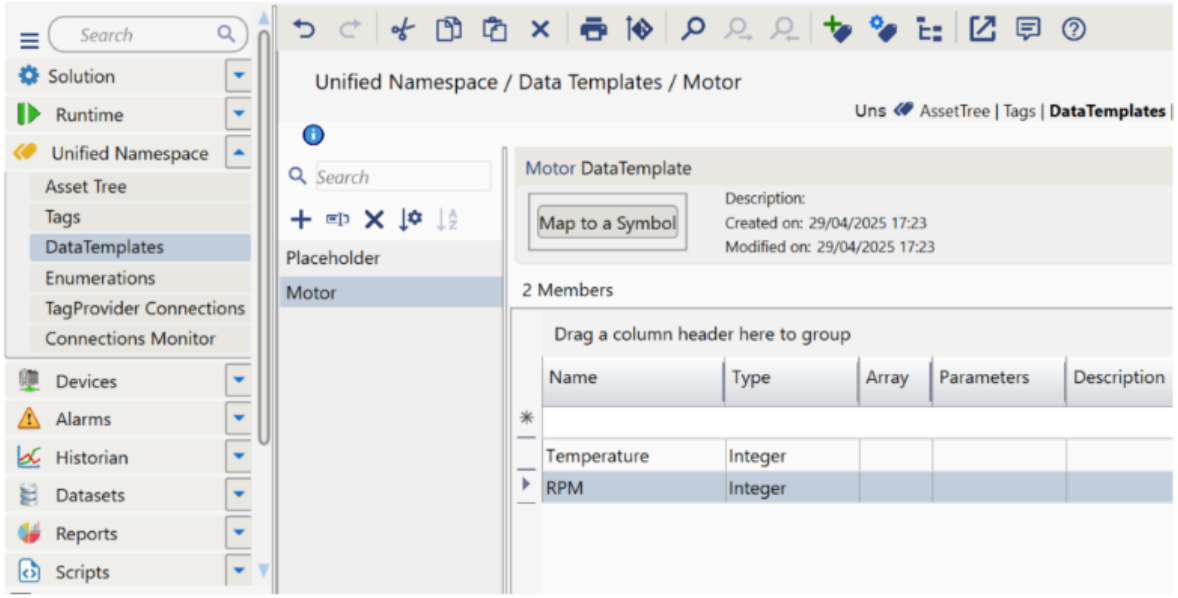
When selected SparkblugB Payload Format, it is **NOT** supported the direct use of simple Tags within the Devices / Points table. Simple Tags refer to variables of common data types such as Integer, Digital, Text, etc.

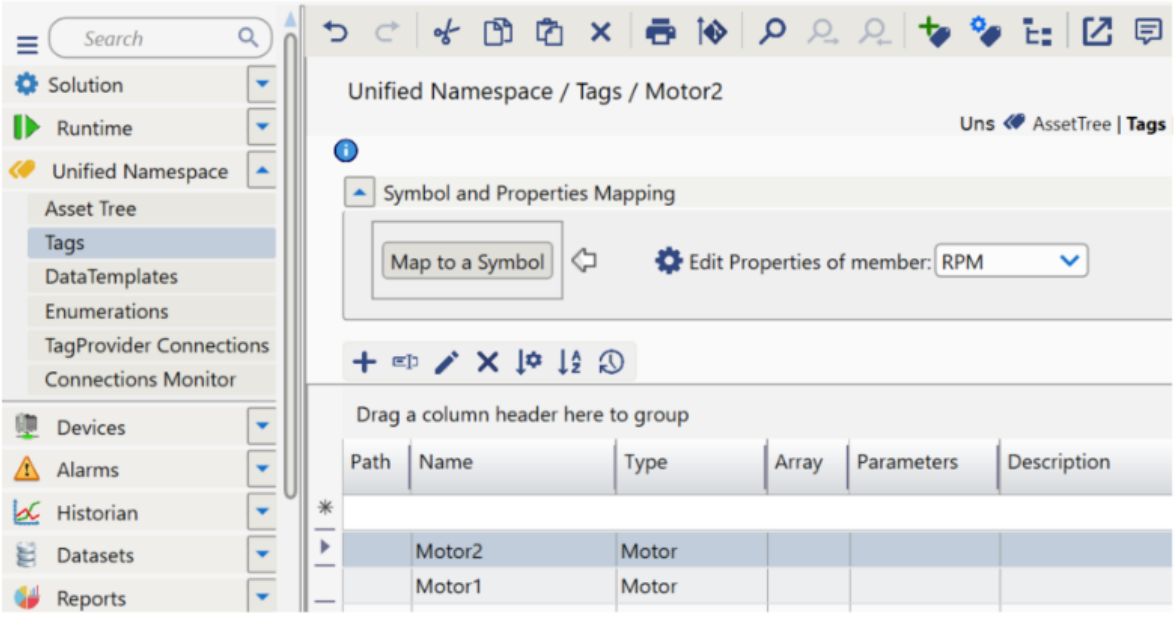
Instead of that, you need to build a DataModel / DataStucture using Templates, and use the Template as your Communication Point. The example below illustrates that.

To ensure proper communication with devices, it is necessary to build a DataModel or DataStructure using Templates. These Templates then serve as the communication points instead of simple Tags.

Example

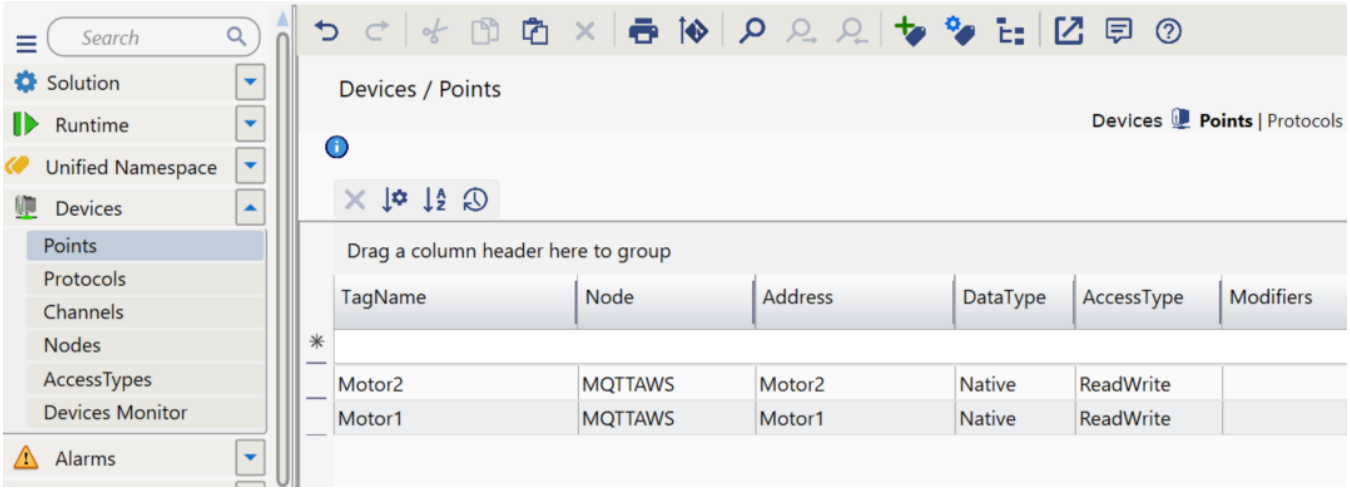
We have a Motor Template with members: Temperature and RPM. From that Template, two Tags are created, Motor1 and Motor2.





Once the Tags are created, we can map them to our DeviceIDs. In the **Devices / Points** tab, add just the main Tags (Motor1 and Motor2) to the table.

Both Tags are mapped to the same **NodeID**, but different **DeviceIDs** and **ReadWrite** AccessType.



In Runtime, the Template members will be automatically expanded and your MQTTspB DBIRTH payload will look like the example below:

▼ spBv1.0/FACTORY1/DBIRTH/Line3/Motor1


```
{
  "timestamp": "1745959292388",
  "metric": [
    {
      "name": "RPM",
      "alias": "1",
      "timestamp": "1745959290489",
      "datatype": 3,
      "intValue": 5
    },
    {
      "name": "Temperature",
      "alias": "2",
      "timestamp": "1745959290489",
      "datatype": 3,
      "intValue": 0
    },
    {
      "name": "Device Control/Rebirth",
      "timestamp": "1745959292411",
      "datatype": 11,
      "booleanValue": false
    }
  ],
  "seq": "1"
}
```


▼ spBv1.0/FACTORY1/DBIRTH/Line3/Motor2


```
{
  "timestamp": "1745959292412",
  "metric": [
    {
      "name": "RPM",
      "alias": "1",
      "timestamp": "1745959290571",
      "datatype": 3,
      "intValue": 0
    },
    {
      "name": "Temperature",
      "alias": "2",
      "timestamp": "1745959290571",
      "datatype": 3,
      "intValue": 0
    },
    {
      "name": "Device Control/Rebirth",
      "timestamp": "1745959292413",
      "datatype": 11,
      "booleanValue": false
    }
  ],
  "seq": "2"
}
```


When changing values in Motors tags, DDATA payloads will be sent like the example below:

▼ spBv1.0/FACTORY1/DDATA/Line3/Motor1


```
{
  "timestamp": "1745960163709",
  "metric": [
    {
      "name": "Temperature",
      "alias": "2",
      "timestamp": "1745960163515",
      "datatype": 3,
      "intValue": 4
    }
  ],
  "seq": "3"
}
```


It is possible to check the datatype codes of Sparkplug in this document.

Flat Payload Format

When the **Flat** Payload Format is selected, both simple Tags and Templates are supported.

1. **JSON Tags / Text Tags with JSON format**:
    - Tags in JSON format or text Tags in JSON format will publish the direct value of the JSON in the topic.
2. **Other Tags**:

- For Tags that do not have a JSON value, the payload will be published in the following format:

```
{
  "name": "Tag.Name",
  "value": <value>,
  "timestamp": <timestamp>,
  "quality": <quality>
}
```

Topic Format:

- The topic published in the **Flat** format will follow the structure:
  `GroupId/NodeId/DeviceId`

Access Type

AccessType defines how the device interacts with the platform.

In this communication protocol the **DeviceControl** (Write Only) AccessType should be used.

Troubleshoot

This section details some errors you might see in your connection to AWS IoT Core via MQTT protocol and its possible causes.

Driver Revision History

MQTTAWS Revision History	
Version	**Notes**
1.0.0.3	Report Out of Range by Exception now uses the tag's ScaleMin/ScaleMax as its range (Min/Max clamps the value, so nothing was published), reports tags without a scale range once in the Trace and no longer delays start-up; a certificate not found at its configured path is also looked up in the machine Certificates folder; fixed topic remapping when the edge node ID changes at runtime.
1.0.0.2	Flat Payload Format added.
1.0.0.1	Sparkblug structure.
1.0.0.0	Initial release on new documentation standards.

In this section...
