# Table of Contents

MSDN forum

Pricing

Pricing calculator

Service updates

Stack Overflow

Technical case studies

Videos

# What is Azure IoT Edge - preview

5/11/2018 • 4 min to read • Edit Online

Azure IoT Edge moves cloud analytics and custom business logic to devices so that your organization can focus on business insights instead of data management. Enable your solution to truly scale by configuring your IoT software, deploying it to devices via standard containers, and monitoring it all from the cloud.

**NOTE**

Azure IoT Edge is available in the free and standard tier of IoT Hub. The free tier is for testing and evaluation only. For more information about the basic and standard tiers, see How to choose the right IoT Hub tier.

Analytics drives business value in IoT solutions, but not all analytics needs to be in the cloud. If you want a device to respond to emergencies as quickly as possible, you can perform anomaly detection on the device itself. Similarly, if you want to reduce bandwidth costs and avoid transferring terabytes of raw data, you can perform data cleaning and aggregation locally. Then send the insights to the cloud.

Azure IoT Edge is made up of three components:

- IoT Edge modules are containers that run Azure services, 3rd party services, or your own code. They are deployed to IoT Edge devices and execute locally on those devices.
- The IoT Edge runtime runs on each IoT Edge device and manages the modules deployed to each device.
- A cloud-based interface enables you to remotely monitor and manage IoT Edge devices.

## IoT Edge modules

IoT Edge modules are units of execution, currently implemented as Docker compatible containers, that run your business logic at the edge. Multiple modules can be configured to communicate with each other, creating a pipeline of data processing. You can develop custom modules or package certain Azure services into modules that provide insights offline and at the edge.

### Artificial Intelligence on the edge

Azure IoT Edge allows you to deploy complex event processing, machine learning, image recognition and other high value AI without writing it in house. Azure services like Azure Functions, Azure Stream Analytics, and Azure Machine Learning can all be run on premises via Azure IoT Edge; however you're not limited to Azure services. Anyone is able to create AI modules and make them available to the community for use.

### Bring your own code

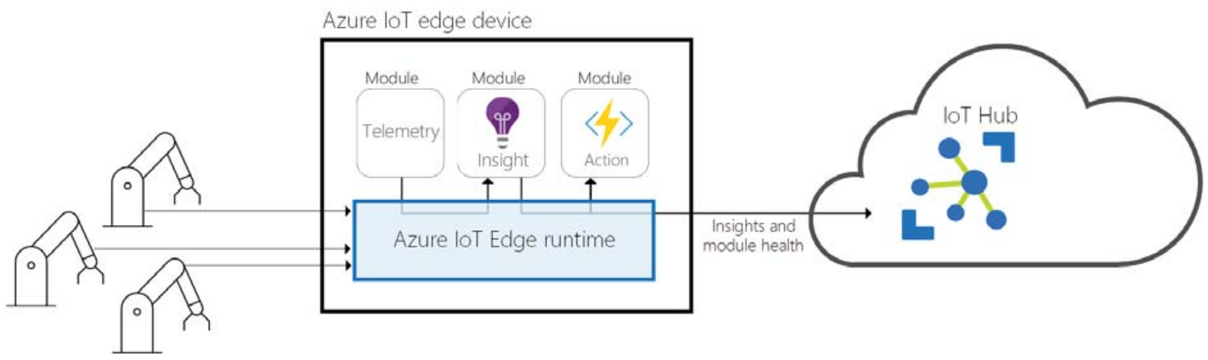
When you want to deploy your own code to your devices, Azure IoT Edge supports that, too. Azure IoT Edge holds to the same programming model as the other Azure IoT services. The same code can be run on a device or in the cloud. Azure IoT Edge supports both Linux and Windows so you can code to the platform of your choice. It supports Java, .NET Core 2.0, Node.js, C, and Python so your developers can code in a language they already know and use existing business logic without writing it from scratch.

## IoT Edge runtime

The Azure IoT Edge runtime enables custom and cloud logic on IoT Edge devices. It sits on the IoT Edge device, and performs management and communication operations. The runtime performs several functions:

- Installs and updates workloads on the device.

- Maintains Azure IoT Edge security standards on the device.
- Ensures that IoT Edge modules are always running.
- Reports module health to the cloud for remote monitoring.
- Facilitates communication between downstream leaf devices and the IoT Edge device.
- Facilitates communication between modules on the IoT Edge device.
- Facilitates communication between the IoT Edge device and the cloud.



How you use an Azure IoT Edge device is completely up to you. The runtime is often used to deploy AI to gateways which aggregate and process data from multiple other on premises devices, however this is just one option. Leaf devices could also be Azure IoT Edge devices, regardless of whether they are connected to a gateway or directly to the cloud.

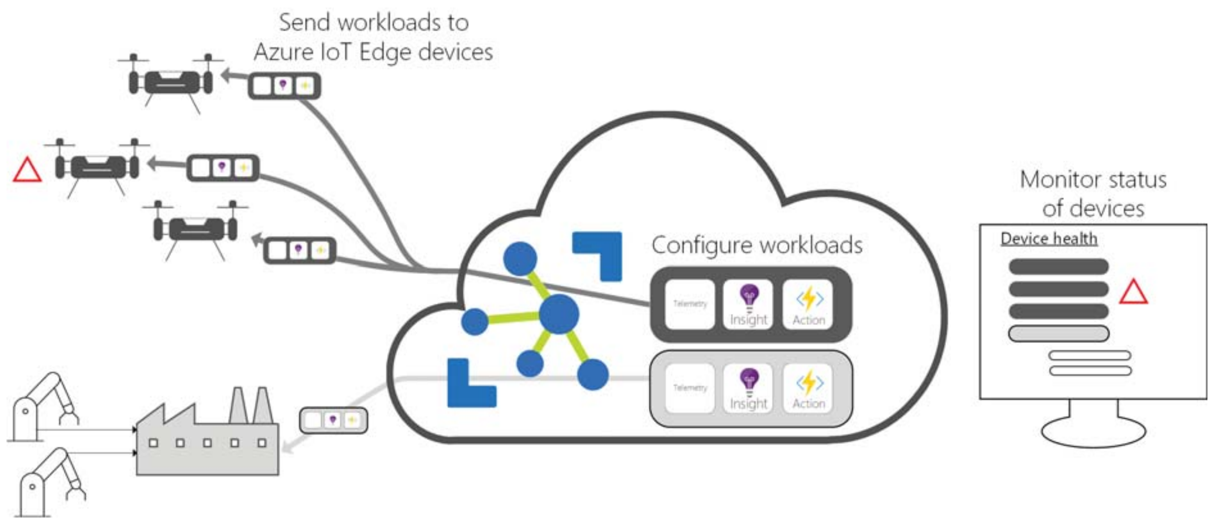
The Azure IoT Edge runtime runs on a large set of IoT devices to enable using the runtime in a wide variety of ways. It supports both Linux and Windows operating systems as well as abstracts hardware details. Use a device smaller than a Raspberry Pi 3 if you're not processing much data or scale up to an industrialized server to run resource intensive workloads.

## IoT Edge cloud interface

Managing the software lifecycle for enterprise devices is complicated. Managing the software lifecycle for millions of heterogenous IoT devices is even more difficult. Workloads must be created and configured for a particular type of device, deployed at scale to the millions of devices in your solution, and monitored to catch any misbehaving devices. These activities can't be done on a per device basis and must be done at scale.

Azure IoT Edge integrates seamlessly with Azure IoT Suite to provide one control plane for your solution's needs. Cloud services allow users to:

- Create and configure a workload to be run on a specific type of device.
- Send a workload to a set of devices.
- Monitor workloads running on devices in the field.



## Next steps

Try out these concepts by deploying IoT Edge on a simulated device.

# Quickstart: Deploy your first IoT Edge module to a Linux or Mac device - preview

5/11/2018 • 4 min to read • Edit Online

Azure IoT Edge moves the power of the cloud to your Internet of Things devices. In this topic, learn how to use the cloud interface to deploy prebuilt code remotely to an IoT Edge device.

If you don't have an active Azure subscription, create a free account before you begin.

## Prerequisites

This quickstart uses your computer or virtual machine like an Internet of Things device. To turn your machine into an IoT Edge device, the following services are required:

- Python pip, to install the IoT Edge runtime.
  - Linux: `sudo apt-get install python-pip`.

    **NOTE**
    On certain distributions (such as Raspbian), you might also need to upgrade certain pip packages and install additional dependencies:

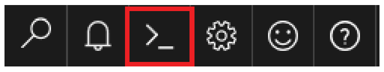
    ```
    sudo pip install --upgrade setuptools pip
    sudo apt-get install python2.7-dev libffi-dev libssl-dev
    ```

  - MacOS: `sudo easy_install pip`.
- Docker, to run the IoT Edge modules
  - Install Docker for Linux and make sure that it's running.
  - Install Docker for Mac and make sure that it's running.

## Create an IoT hub with Azure CLI

Create an IoT hub in your Azure subscription. The free level of IoT Hub works for this quickstart. If you've used IoT Hub in the past and already have a free hub created, you can skip this section and go on to Register an IoT Edge device. Each subscription can only have one free IoT hub.

1. Sign in to the Azure portal.
2. Select the **Cloud Shell** button.



3. Create a resource group. The following code creates a resource group called **IoTEdge** in the **West US** region:

```
az group create --name IoTEdge --location westus
```

4. Create an IoT hub in your new resource group. The following code creates a free **F1** hub called **MyIotHub**

in the resource group **IoTEdge**:

```
az iot hub create --resource-group IoTEdge --name MyIotHub --sku F1
```

## Register an IoT Edge device

Create a device identity for your simulated device so that it can communicate with your IoT hub. Since IoT Edge devices behave and can be managed differently than typical IoT devices, you declare this to be an IoT Edge device from the beginning.

1. In the Azure portal, navigate to your IoT hub.
2. Select **IoT Edge (preview)**.
3. Select **Add IoT Edge device**.
4. Give your simulated device a unique device ID.
5. Select **Save** to add your device.
6. Select your new device from the list of devices.
7. Copy the value for **Connection string--primary key** and save it. You'll use this value to configure the IoT Edge runtime in the next section.

## Install and start the IoT Edge runtime

The IoT Edge runtime is deployed on all IoT Edge devices. It comprises two modules. First, the IoT Edge agent facilitates deployment and monitoring of modules on the IoT Edge device. Second, the IoT Edge hub manages communications between modules on the IoT Edge device, and between the device and IoT Hub.

On the machine where you'll run the IoT Edge device, download the IoT Edge control script:

```
sudo pip install -U azure-iot-edge-runtime-ctl
```

Configure the runtime with your IoT Edge device connection string from the previous section:

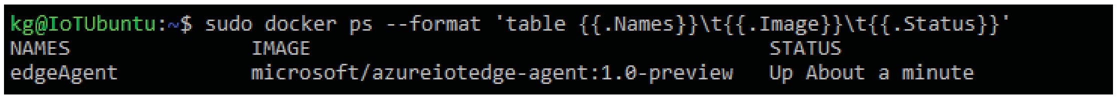
```
sudo iotedgectl setup --connection-string "{device connection string}" --nopass
```

Start the runtime:

```
sudo iotedgectl start
```

Check Docker to see that the IoT Edge agent is running as a module:

```
sudo docker ps
```

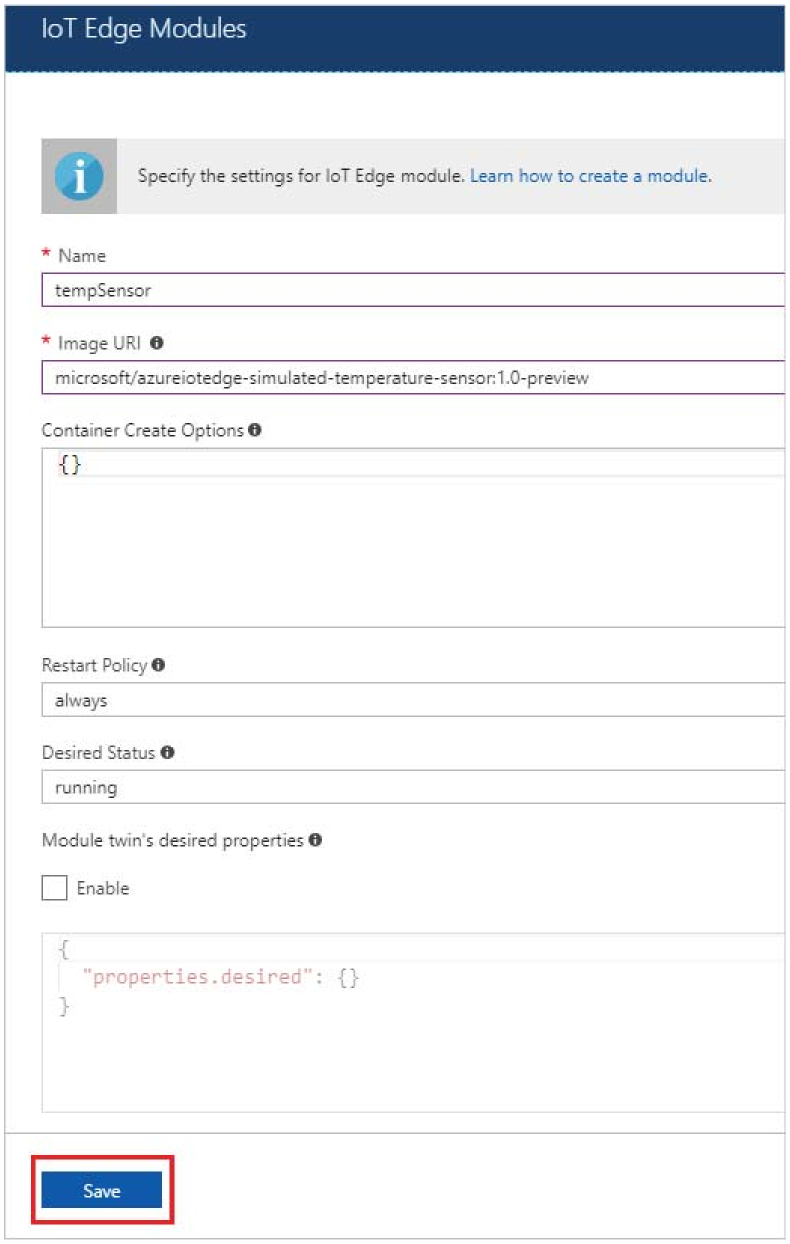


```
kg@IoTUbuntu:~$ sudo docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}'
NAMES               IMAGE                                     STATUS
edgeAgent           microsoft/azureiotedge-agent:1.0-preview   Up About a minute
```

## Deploy a module

One of the key capabilities of Azure IoT Edge is being able to deploy modules to your IoT Edge devices from the cloud. An IoT Edge module is an executable package implemented as a container. In this section, you deploy a module that generates telemetry for your simulated device.

1. In the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. Select **Add IoT Edge Module**.
5. In the **Name** field, enter `tempSensor`.
6. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
7. Leave the other settings unchanged, and select **Save**.



Specify the settings for IoT Edge module. Learn how to create a module.

* Name: tempSensor

* Image URI: microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview

Container Create Options:

```
{}
```

Restart Policy: always

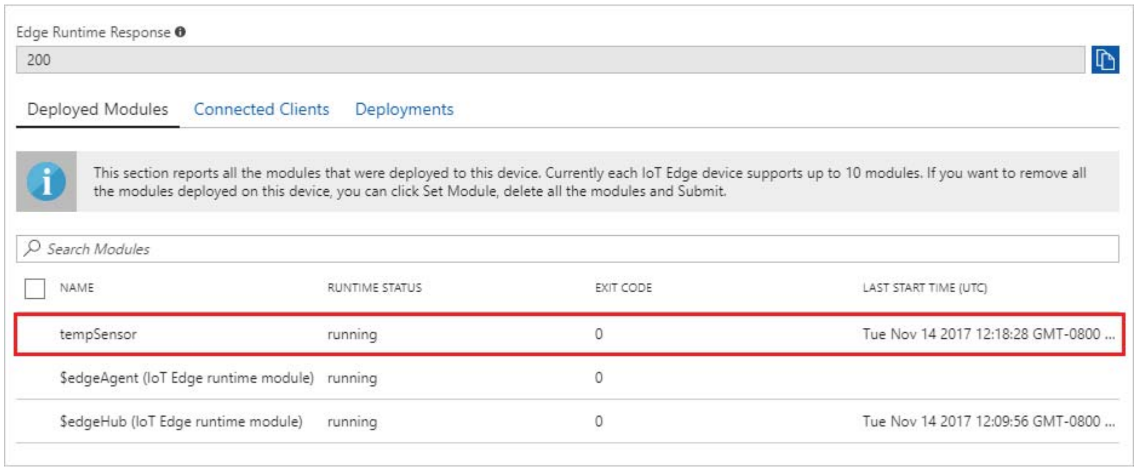
Desired Status: running

Module twin's desired properties

☐ Enable

```
{
  "properties.desired": {}
}
```

8. Back in the **Add modules** step, select **Next**.
9. In the **Specify routes** step, select **Next**.
10. In the **Review template** step, select **Submit**.
11. Return to the device details page and select **Refresh**. You should see the new tempSensor module running along the IoT Edge runtime.


Edge Runtime Response

200

Deployed Modules | Connected Clients | Deployments

This section reports all the modules that were deployed to this device. Currently each IoT Edge device supports up to 10 modules. If you want to remove all the modules deployed on this device, you can click Set Module, delete all the modules and Submit.

Search Modules

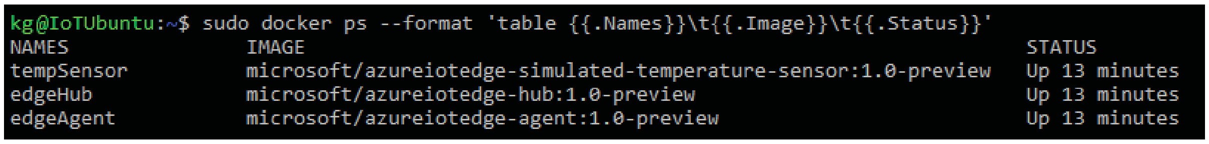
| NAME | RUNTIME STATUS | EXIT CODE | LAST START TIME (UTC) |
|---|---|---|---|
| tempSensor | running | 0 | Tue Nov 14 2017 12:18:28 GMT-0800 ... |
| $edgeAgent (IoT Edge runtime module) | running | 0 | |
| $edgeHub (IoT Edge runtime module) | running | 0 | Tue Nov 14 2017 12:09:56 GMT-0800 ... |

## View generated data

In this quickstart, you created a new IoT Edge device and installed the IoT Edge runtime on it. Then, you used the Azure portal to push an IoT Edge module to run on the device without having to make changes to the device itself. In this case, the module that you pushed creates environmental data that you can use for the tutorials.

Open the command prompt on the computer running your simulated device again. Confirm that the module deployed from the cloud is running on your IoT Edge device:

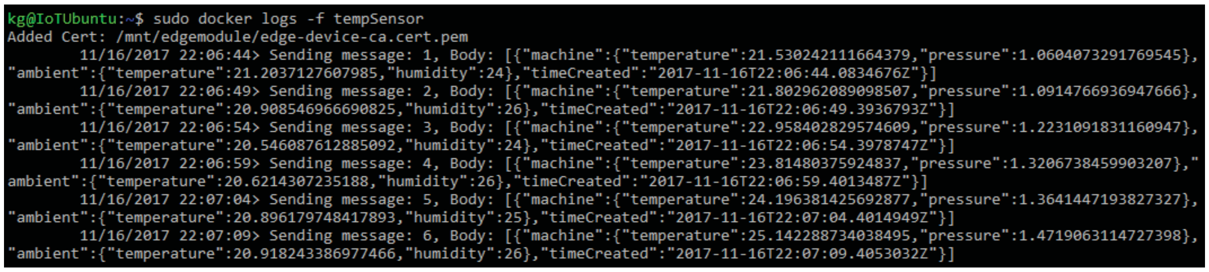
```
sudo docker ps
```



```
kg@IoTUbuntu:~$ sudo docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}'
NAMES               IMAGE                                                           STATUS
tempSensor          microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview   Up 13 minutes
edgeHub             microsoft/azureiotedge-hub:1.0-preview                             Up 13 minutes
edgeAgent           microsoft/azureiotedge-agent:1.0-preview                           Up 13 minutes
```

View the messages being sent from the tempSensor module to the cloud:

```
sudo docker logs -f tempSensor
```



```
kg@IoTUbuntu:~$ sudo docker logs -f tempSensor
Added Cert: /mnt/edgemodule/edge-device-ca.cert.pem
        11/16/2017 22:06:44> Sending message: 1, Body: [{"machine":{"temperature":21.530242111664379,"pressure":1.0604073291769545},
"ambient":{"temperature":21.2037127607985,"humidity":24},"timeCreated":"2017-11-16T22:06:44.0834676Z"}]
        11/16/2017 22:06:49> Sending message: 2, Body: [{"machine":{"temperature":21.802962089098507,"pressure":1.0914766936947666},
"ambient":{"temperature":20.908546966690825,"humidity":26},"timeCreated":"2017-11-16T22:06:49.3936793Z"}]
        11/16/2017 22:06:54> Sending message: 3, Body: [{"machine":{"temperature":22.958402829574609,"pressure":1.2231091831160947},
"ambient":{"temperature":20.546087612885092,"humidity":24},"timeCreated":"2017-11-16T22:06:54.3978747Z"}]
        11/16/2017 22:06:59> Sending message: 4, Body: [{"machine":{"temperature":23.81480375924837,"pressure":1.320673845990320 7},"
ambient":{"temperature":20.6214307235188,"humidity":26},"timeCreated":"2017-11-16T22:06:59.4013487Z"}]
        11/16/2017 22:07:04> Sending message: 5, Body: [{"machine":{"temperature":24.196381425692877,"pressure":1.3641447193827327},
"ambient":{"temperature":20.896179748417893,"humidity":25},"timeCreated":"2017-11-16T22:07:04.4014949Z"}]
        11/16/2017 22:07:09> Sending message: 6, Body: [{"machine":{"temperature":25.142288734038495,"pressure":1.4719063114727398},
"ambient":{"temperature":20.918243386977466,"humidity":26},"timeCreated":"2017-11-16T22:07:09.4053032Z"}]
```

You can also view the telemetry the device is sending by using the IoT Hub explorer tool.

## Clean up resources

If you want to remove the simulated device that you created, along with the Docker containers that were started for each module, use the following command:

```
sudo iotedgectl uninstall
```

When you no longer need the IoT Hub you created, you can use the az iot hub delete command to remove the resource and any devices associated with it:

```
az iot hub delete --name {your iot hub name} --resource-group {your resource group name}
```

## Next steps

You learned how to deploy an IoT Edge module to an IoT Edge device. Now try deploying different types of Azure services as modules, so that you can analyze data at the edge.

- Deploy your own code as a module
- Deploy Azure Function as a module
- Deploy Azure Stream Analytics as a module

# Quickstart: Deploy your first IoT Edge module from the Azure portal to a Windows device - preview

4/9/2018 • 4 min to read • Edit Online

In this quickstart, use the Azure IoT Edge cloud interface to deploy prebuilt code remotely to an IoT Edge device. To accomplish this task, first use your Windows device to simulate an IoT Edge device, then you can deploy a module to it.

If you don't have an active Azure subscription, create a free account before you begin.

## Prerequisites

This tutorial assumes that you're using a computer or virtual machine running Windows to simulate an Internet of Things device. If you're running Windows in a virtual machine, enable nested virtualization and allocate at least 2GB memory.

1. Make sure you're using a supported Windows version:
   - Windows 10
   - Windows Server
2. Install Docker for Windows and make sure it's running.
3. Install Python 2.7 on Windows and make sure you can use the pip command.
4. Run the following command to download the IoT Edge control script.

```
pip install -U azure-iot-edge-runtime-ctl
```

**NOTE**

Azure IoT Edge can run either Windows containers or Linux containers. To use Windows containers, you have to run:

- Windows 10 Fall Creators Update, or
- Windows Server 1709 (Build 16299), or
- Windows IoT Core (Build 16299) on a x64-based device

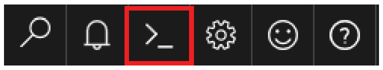
For Windows IoT Core, follow the instructions in Install the IoT Edge runtime on Windows IoT Core. Otherwise, simply configure Docker to use Windows containers, and optionally validate your prerequisites with the following powershell command:

```
Invoke-Expression (Invoke-WebRequest -useb https://aka.ms/iotedgewin)
```

## Create an IoT hub with Azure CLI

Create an IoT hub in your Azure subscription. The free level of IoT Hub works for this quickstart. If you've used IoT Hub in the past and already have a free hub created, you can skip this section and go on to Register an IoT Edge device. Each subscription can only have one free IoT hub.

1. Sign in to the Azure portal.
2. Select the **Cloud Shell** button.



3. Create a resource group. The following code creates a resource group called **IoTEdge** in the **West US** region:

```
az group create --name IoTEdge --location westus
```

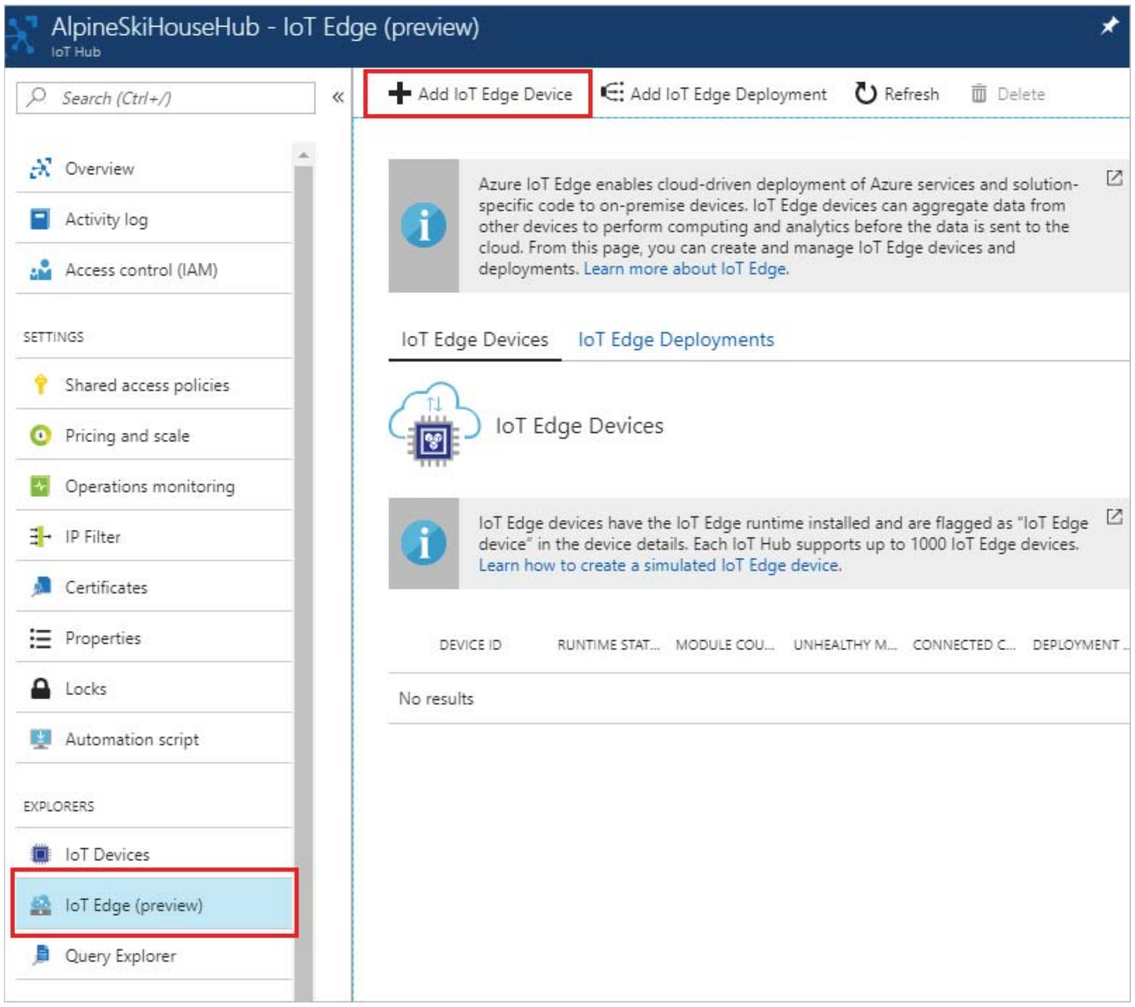
4. Create an IoT hub in your new resource group. The following code creates a free **F1** hub called **MyIotHub** in the resource group **IoTEdge**:

```
az iot hub create --resource-group IoTEdge --name MyIotHub --sku F1
```

## Register an IoT Edge device

Create a device identity for your simulated device so that it can communicate with your IoT hub. Since IoT Edge devices behave and can be managed differently than typical IoT devices, you declare this to be an IoT Edge device from the beginning.

1. In the Azure portal, navigate to your IoT hub.
2. Select **IoT Edge (preview)** then select **Add IoT Edge Device**.



Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. Learn more about IoT Edge.

IoT Edge devices have the IoT Edge runtime installed and are flagged as "IoT Edge device" in the device details. Each IoT Hub supports up to 1000 IoT Edge devices. Learn how to create a simulated IoT Edge device.

No results

3. Give your simulated device a unique device ID.
4. Select **Save** to add your device.

5. Select your new device from the list of devices.
6. Copy the value for **Connection string—primary key** and save it. You'll use this value to configure the IoT Edge runtime in the next section.

## Configure the IoT Edge runtime

The IoT Edge runtime is deployed on all IoT Edge devices. It comprises two modules. First, the IoT Edge agent facilitates deployment and monitoring of modules on the IoT Edge device. Second, the IoT Edge hub manages communications between modules on the IoT Edge device, and between the device and IoT Hub.

Configure the runtime with your IoT Edge device connection string from the previous section.

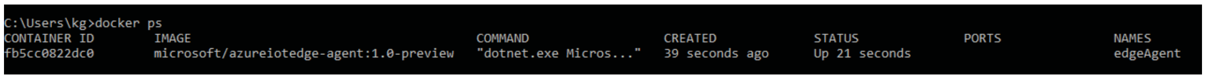
```
iotedgectl setup --connection-string "{device connection string}" --nopass
```

Start the runtime.

```
iotedgectl start
```

Check Docker to see that the IoT Edge agent is running as a module.

```
docker ps
```

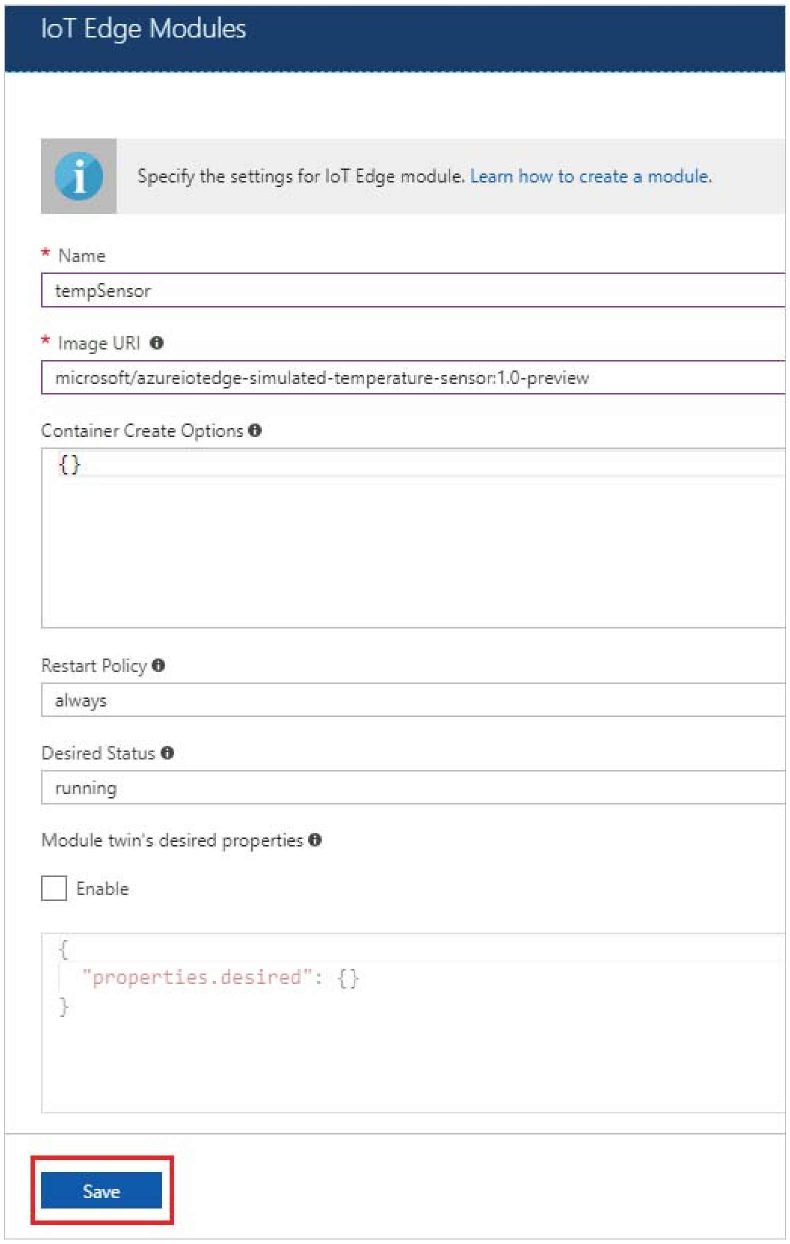


```
C:\Users\kg>docker ps
CONTAINER ID        IMAGE                                        COMMAND                  CREATED             STATUS              PORTS               NAMES
fb5cc0822dc0        microsoft/azureiotedge-agent:1.0-preview     "dotnet.exe Micros..."   39 seconds ago      Up 21 seconds                           edgeAgent
```

## Deploy a module

One of the key capabilities of Azure IoT Edge is being able to deploy modules to your IoT Edge devices from the cloud. An IoT Edge module is an executable package implemented as a container. In this section, you deploy a module that generates telemetry for your simulated device.

1. In the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. Select **Add IoT Edge Module**.
5. In the **Name** field, enter `tempSensor`.
6. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
7. Leave the other settings unchanged, and select **Save**.


IoT Edge Modules

Specify the settings for IoT Edge module. Learn how to create a module.

* Name

tempSensor

* Image URI

microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview

Container Create Options

```
{}
```

Restart Policy

always

Desired Status

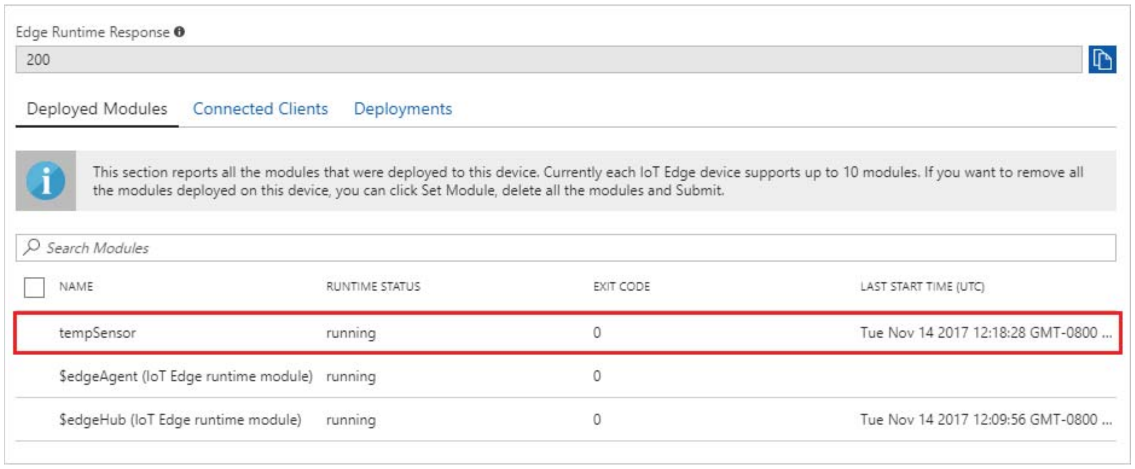
running

Module twin's desired properties

☐ Enable

```
{
  "properties.desired": {}
}
```

8. Back in the **Add modules** step, select **Next**.
9. In the **Specify routes** step, select **Next**.
10. In the **Review template** step, select **Submit**.
11. Return to the device details page and select **Refresh**. You should see the new tempSensor module running along the IoT Edge runtime.


Edge Runtime Response

200

Deployed Modules | Connected Clients | Deployments

This section reports all the modules that were deployed to this device. Currently each IoT Edge device supports up to 10 modules. If you want to remove all the modules deployed on this device, you can click Set Module, delete all the modules and Submit.

Search Modules

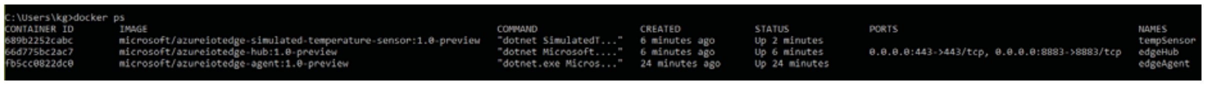
| NAME | RUNTIME STATUS | EXIT CODE | LAST START TIME (UTC) |
|---|---|---|---|
| tempSensor | running | 0 | Tue Nov 14 2017 12:18:28 GMT-0800 ... |
| $edgeAgent (IoT Edge runtime module) | running | 0 | |
| $edgeHub (IoT Edge runtime module) | running | 0 | Tue Nov 14 2017 12:09:56 GMT-0800 ... |

## View generated data

In this quickstart, you created a new IoT Edge device and installed the IoT Edge runtime on it. Then, you used the Azure portal to push an IoT Edge module to run on the device without having to make changes to the device itself. In this case, the module that you pushed creates environmental data that you can use for the tutorials.

Open the command prompt on the computer running your simulated device again. Confirm that the module deployed from the cloud is running on your IoT Edge device.

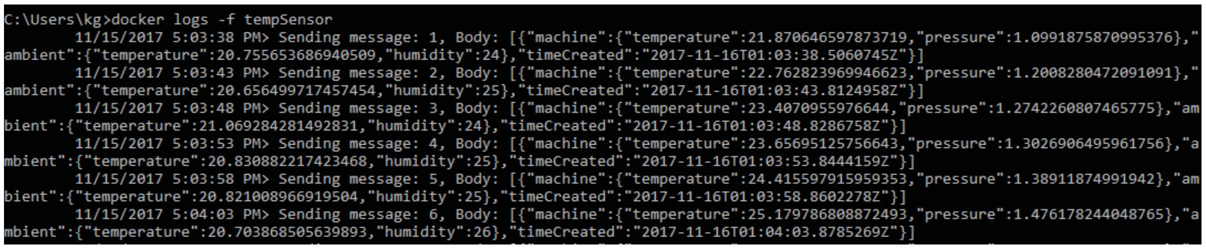
```
docker ps
```



```
C:\Users\kg>docker ps
CONTAINER ID        IMAGE                                                          COMMAND                  CREATED             STATUS              PORTS                                            NAMES
689b2252cabc        microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview   "dotnet SimulatedT..."   6 minutes ago       Up 2 minutes                                                         tempSensor
66d775bc2ac7        microsoft/azureiotedge-hub:1.0-preview                         "dotnet Microsoft...."   6 minutes ago       Up 6 minutes        0.0.0.0:443->443/tcp, 0.0.0.0:8883->8883/tcp     edgeHub
fb5cc0822dc0        microsoft/azureiotedge-agent:1.0-preview                       "dotnet.exe Micros..."   24 minutes ago      Up 24 minutes                                                        edgeAgent
```

View the messages being sent from the tempSensor module to the cloud.

```
docker logs -f tempSensor
```



```
C:\Users\kg>docker logs -f tempSensor
        11/15/2017 5:03:38 PM> Sending message: 1, Body: [{"machine":{"temperature":21.870646597873719,"pressure":1.0991875870995376},"
ambient":{"temperature":20.755653686940509,"humidity":24},"timeCreated":"2017-11-16T01:03:38.5060745Z"}]
        11/15/2017 5:03:43 PM> Sending message: 2, Body: [{"machine":{"temperature":22.762823969946623,"pressure":1.2008280472091091},"
ambient":{"temperature":20.656499717457454,"humidity":25},"timeCreated":"2017-11-16T01:03:43.8124958Z"}]
        11/15/2017 5:03:48 PM> Sending message: 3, Body: [{"machine":{"temperature":23.4070955976644,"pressure":1.2742260807465775},"am
bient":{"temperature":21.069284281492831,"humidity":24},"timeCreated":"2017-11-16T01:03:48.8286758Z"}]
        11/15/2017 5:03:53 PM> Sending message: 4, Body: [{"machine":{"temperature":23.65695125756643,"pressure":1.3026906495961756},"a
mbient":{"temperature":20.830882217423468,"humidity":25},"timeCreated":"2017-11-16T01:03:53.8444159Z"}]
        11/15/2017 5:03:58 PM> Sending message: 5, Body: [{"machine":{"temperature":24.415597915959353,"pressure":1.38911874991942},"am
bient":{"temperature":20.821008966919504,"humidity":25},"timeCreated":"2017-11-16T01:03:58.8602278Z"}]
        11/15/2017 5:04:03 PM> Sending message: 6, Body: [{"machine":{"temperature":25.179786808872493,"pressure":1.476178244048765},"a
mbient":{"temperature":20.703868505639893,"humidity":26},"timeCreated":"2017-11-16T01:04:03.8785269Z"}]
```

You can also view the telemetry the device is sending by using the IoT Hub explorer tool.

## Clean up resources

If you want to remove the simulated device that you created, along with the Docker containers that were started for each module, use the following command:

```
iotedgectl uninstall
```

When you no longer need the IoT Hub you created, you can use the az iot hub delete command to remove the resource and any devices associated with it:

```
az iot hub delete --name {your iot hub name} --resource-group {your resource group name}
```

## Next steps

You learned how to deploy an IoT Edge module to an IoT Edge device. Now try deploying different types of Azure services as modules, so that you can analyze data at the edge.

- Deploy Azure Function as a module
- Deploy Azure Stream Analytics as a module
- Deploy your own code as a module

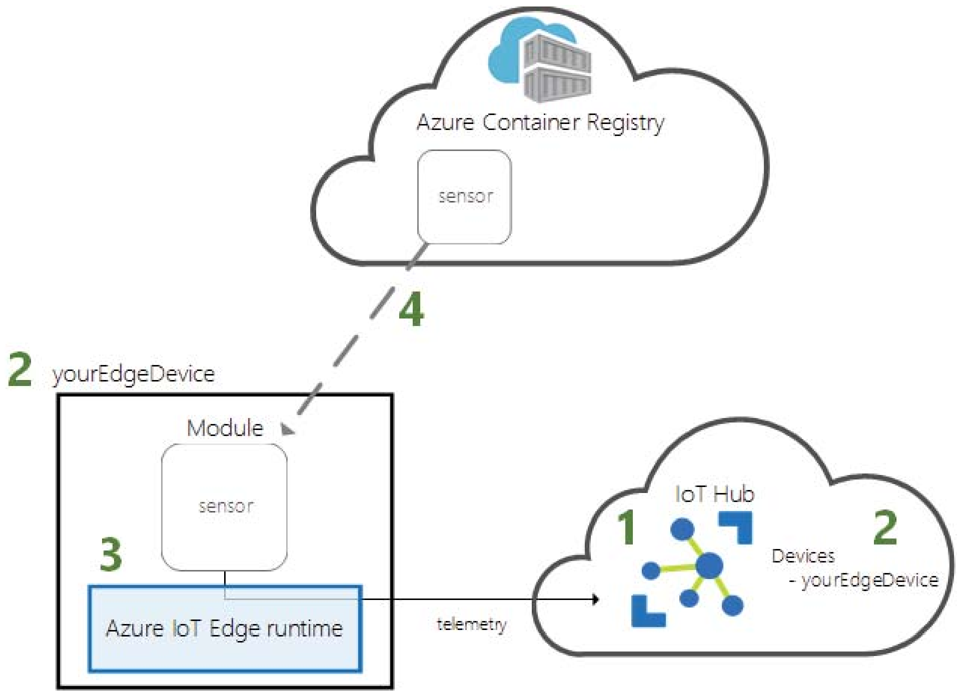
# Deploy Azure IoT Edge on a simulated device in Linux or MacOS - preview

4/9/2018 • 5 min to read • Edit Online

Azure IoT Edge enables you to perform analytics and data processing on your devices, instead of having to push all the data to the cloud. The IoT Edge tutorials demonstrate how to deploy different types of modules, built from Azure services or custom code, but first you need a device to test.

In this tutorial you learn how to:

1. Create an IoT Hub
2. Register an IoT Edge device
3. Start the IoT Edge runtime
4. Deploy a module



The simulated device that you create in this tutorial is a monitor that generates temperature, humidity, and pressure data. The other Azure IoT Edge tutorials build upon the work you do here by deploying modules that analyze the data for business insights.

## Prerequisites

This tutorial uses your computer or virtual machine like an Internet of Things device. To turn your machine into an IoT Edge device, the following services are required:

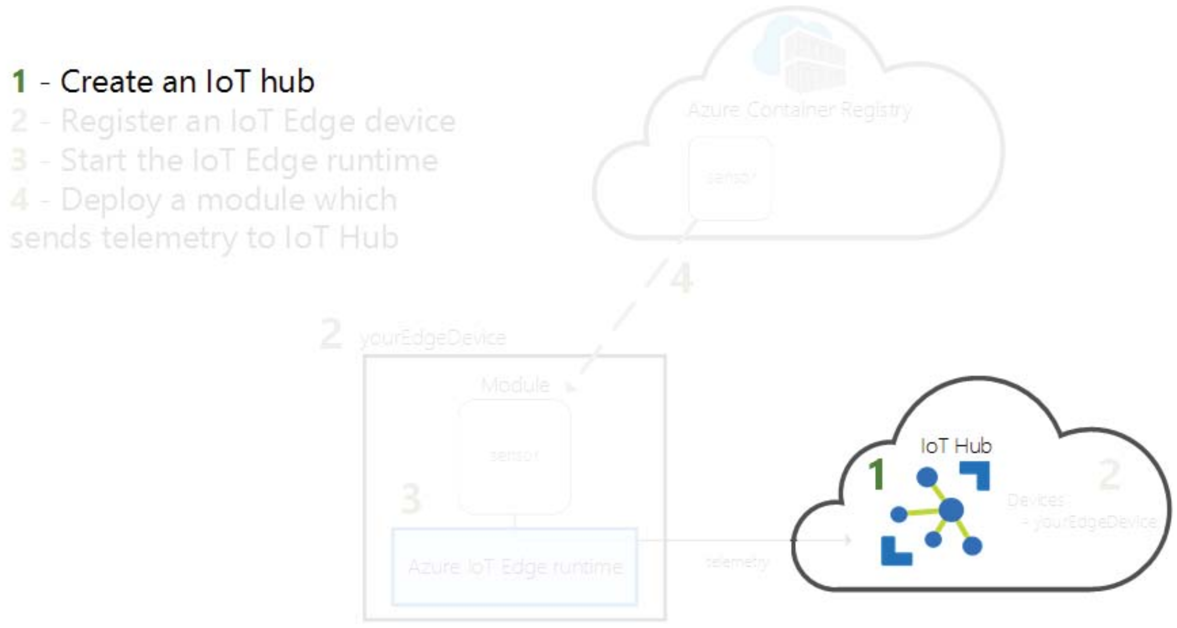
- Python pip, to install the IoT Edge runtime.
  - Linux: `sudo apt-get install python-pip`.
    - *Note that on certain distributions (e.g., Raspbian), you might also need to upgrade certain pip packages and install additional dependencies:*

```
sudo pip install --upgrade setuptools pip sudo apt-get install python2.7-dev libffi-dev
libssl-dev
```

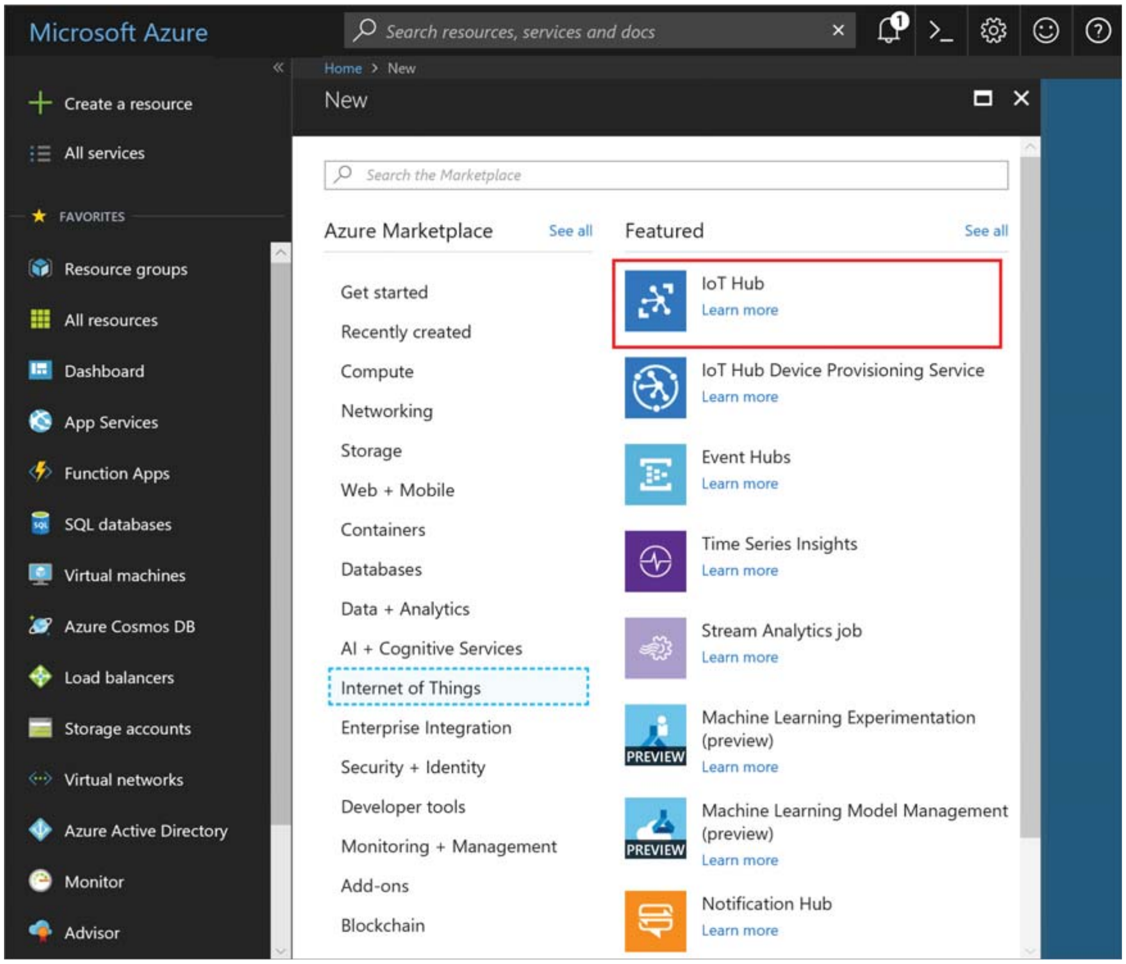
- MacOS: `sudo easy_install pip`.
- Docker, to run the IoT Edge modules
  - Install Docker for Linux and make sure that it's running.
  - Install Docker for Mac and make sure that it's running.

## Create an IoT hub

Start the tutorial by creating your IoT Hub.



1. Sign in to the Azure portal.
2. Select **Create a resource** > **Internet of Things** > **IoT Hub**.

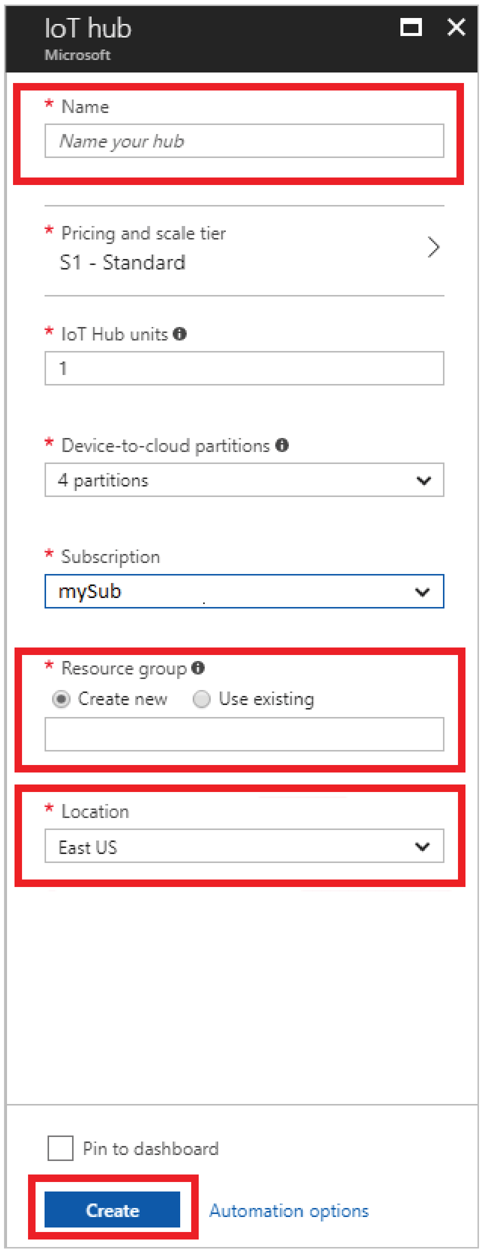


3. In the **IoT hub** pane, enter the following information for your IoT hub:

- **Name**: Create a name for your IoT hub. If the name you enter is valid, a green check mark appears.

> **IMPORTANT**
>
> The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

- **Pricing and scale tier**: For this tutorial, select the **F1 - Free** tier. For more information, see the Pricing and scale tier.
- **Resource group**: Create a resource group to host the IoT hub or use an existing one. For more information, see Use resource groups to manage your Azure resources
- **Location**: Select the closest location to you.
- **Pin to dashboard**: Check this option for easy access to your IoT hub from the dashboard.


IoT hub
Microsoft

* Name
Name your hub

* Pricing and scale tier
S1 - Standard

* IoT Hub units
1

* Device-to-cloud partitions
4 partitions

* Subscription
mySub

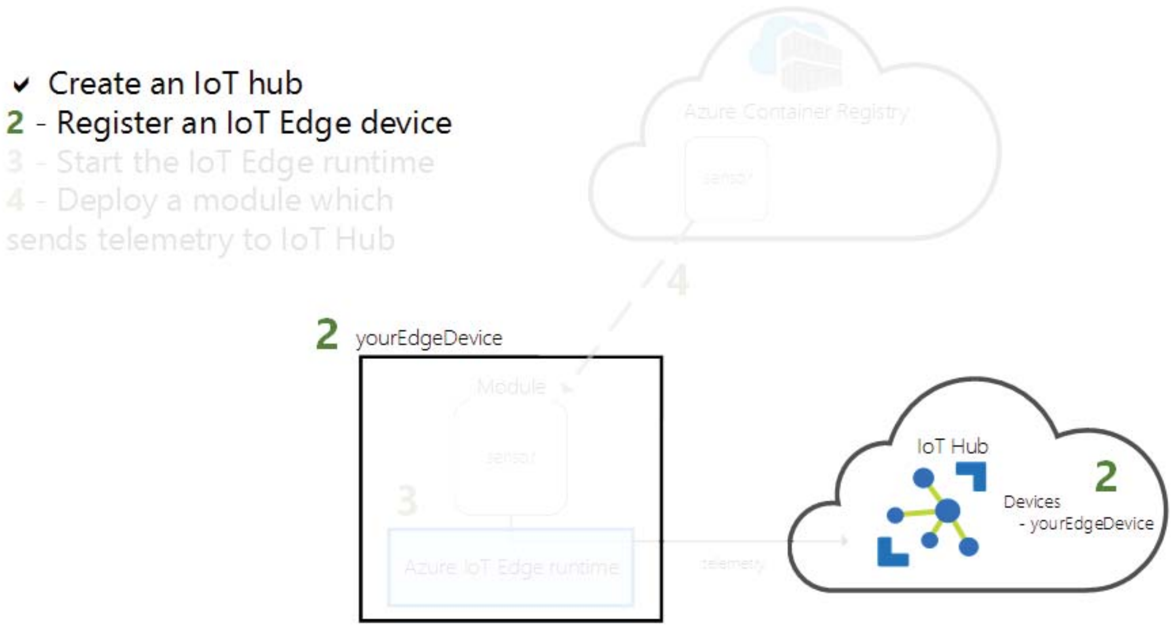
* Resource group
(● Create new) ○ Use existing

* Location
East US

4. Click **Create**. Your IoT hub might take a few minutes to create. You can monitor the progress in the **Notifications** pane.

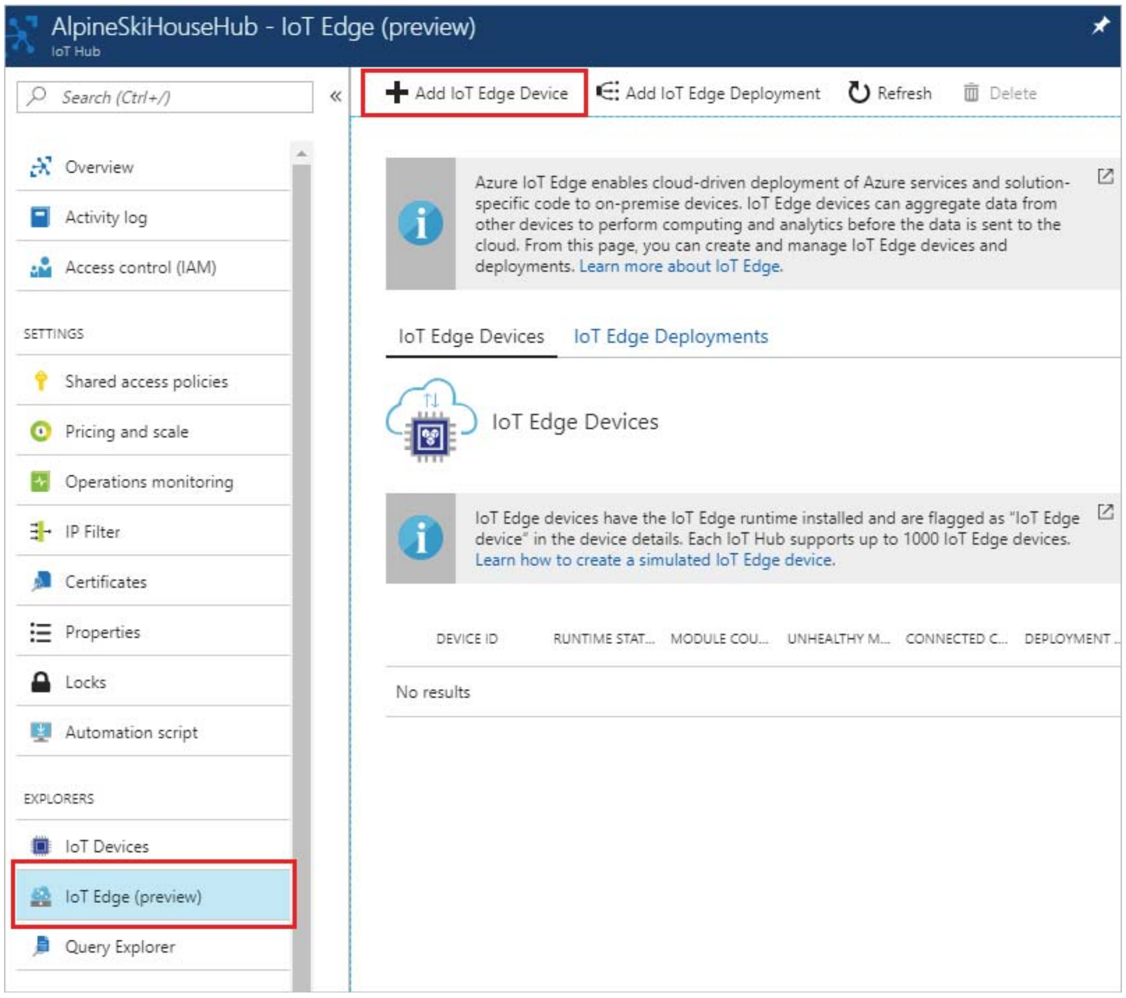
## Register an IoT Edge device

Register an IoT Edge device with your newly created IoT Hub.



Create a device identity for your simulated device so that it can communicate with your IoT hub. Since IoT Edge devices behave and can be managed differently than typical IoT devices, you declare this to be an IoT Edge device from the beginning.

1. In the Azure portal, navigate to your IoT hub.
2. Select **IoT Edge (preview)** then select **Add IoT Edge Device**.



Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. Learn more about IoT Edge.

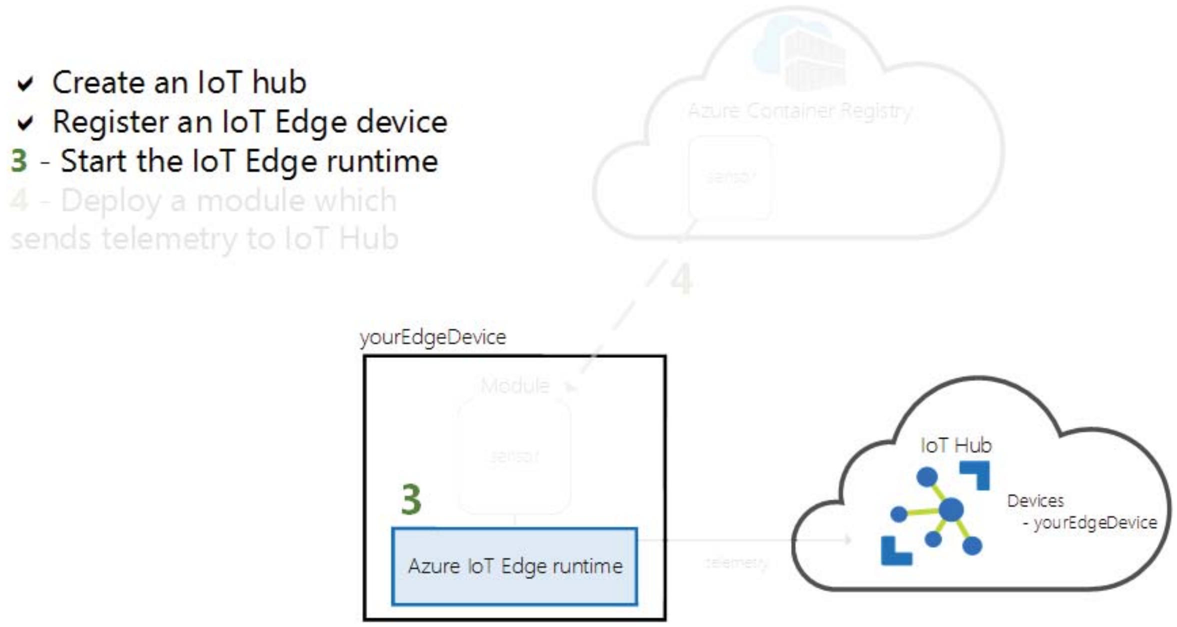
IoT Edge devices have the IoT Edge runtime installed and are flagged as "IoT Edge device" in the device details. Each IoT Hub supports up to 1000 IoT Edge devices. Learn how to create a simulated IoT Edge device.

| DEVICE ID | RUNTIME STAT... | MODULE COU... | UNHEALTHY M... | CONNECTED C... | DEPLOYMENT ... |
|---|---|---|---|---|---|
| No results | | | | | |

3. Give your simulated device a unique device ID.
4. Select **Save** to add your device.
5. Select your new device from the list of devices.
6. Copy the value for **Connection string—primary key** and save it. You'll use this value to configure the IoT Edge runtime in the next section.

## Install and start the IoT Edge runtime

Install and start the Azure IoT Edge runtime on your device.



The IoT Edge runtime is deployed on all IoT Edge devices. It comprises two modules. The **IoT Edge agent** facilitates deployment and monitoring of modules on the IoT Edge device. The **IoT Edge hub** manages communications between modules on the IoT Edge device, and between the device and IoT Hub. When you configure the runtime on your new device, only the IoT Edge agent will start at first. The IoT Edge hub comes later when you deploy a module.

On the machine where you'll run the IoT Edge device, download the IoT Edge control script:

```
sudo pip install -U azure-iot-edge-runtime-ctl
```

Configure the runtime with your IoT Edge device connection string from the previous section:

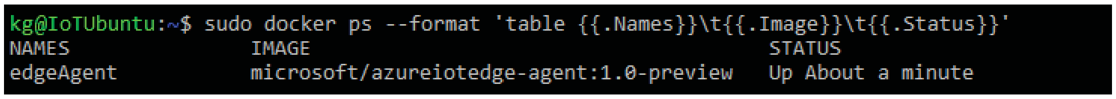
```
sudo iotedgectl setup --connection-string "{device connection string}" --nopass
```

Start the runtime:

```
sudo iotedgectl start
```

Check Docker to see that the IoT Edge agent is running as a module:

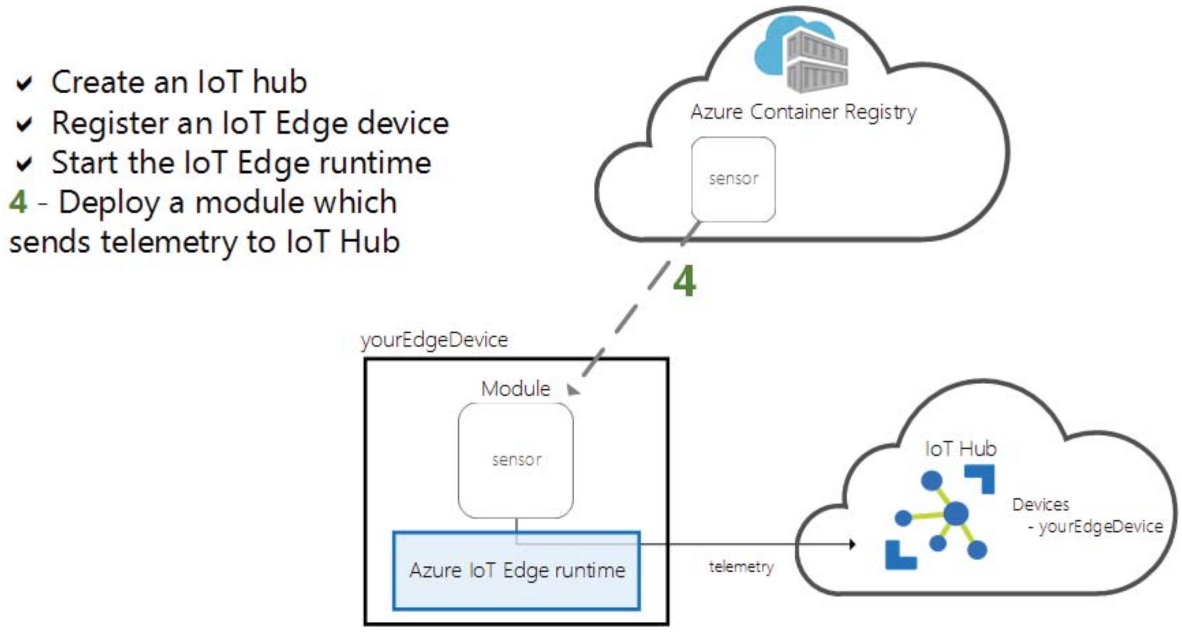
```
sudo docker ps
```



```
kg@IoTUbuntu:~$ sudo docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}'
NAMES               IMAGE                                       STATUS
edgeAgent           microsoft/azureiotedge-agent:1.0-preview    Up About a minute
```

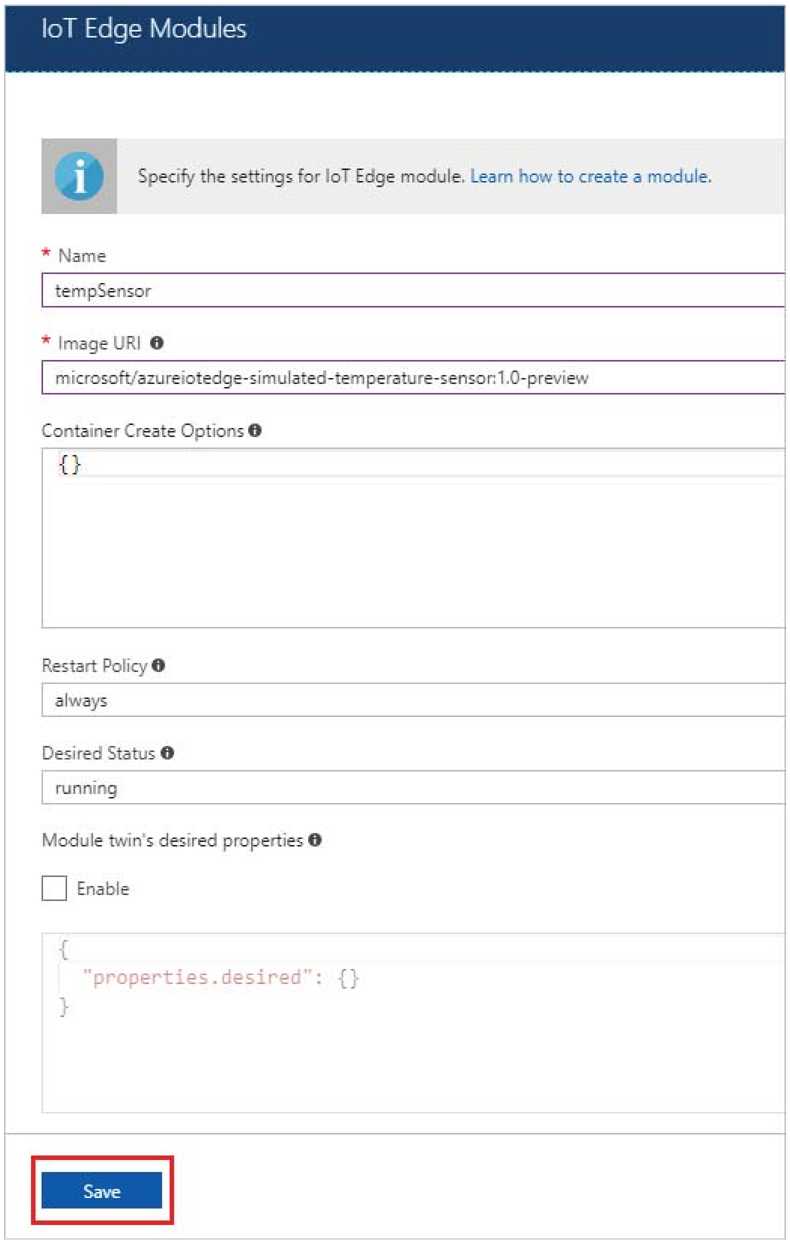
## Deploy a module

Manage your Azure IoT Edge device from the cloud to deploy a module which will send telemetry data to IoT Hub.



One of the key capabilities of Azure IoT Edge is being able to deploy modules to your IoT Edge devices from the cloud. An IoT Edge module is an executable package implemented as a container. In this section, you deploy a module that generates telemetry for your simulated device.

1. In the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. Select **Add IoT Edge Module**.
5. In the **Name** field, enter `tempSensor`.
6. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
7. Leave the other settings unchanged, and select **Save**.


IoT Edge Modules

Specify the settings for IoT Edge module. Learn how to create a module.

* Name
tempSensor

* Image URI
microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview

Container Create Options
```
{}
```

Restart Policy
always

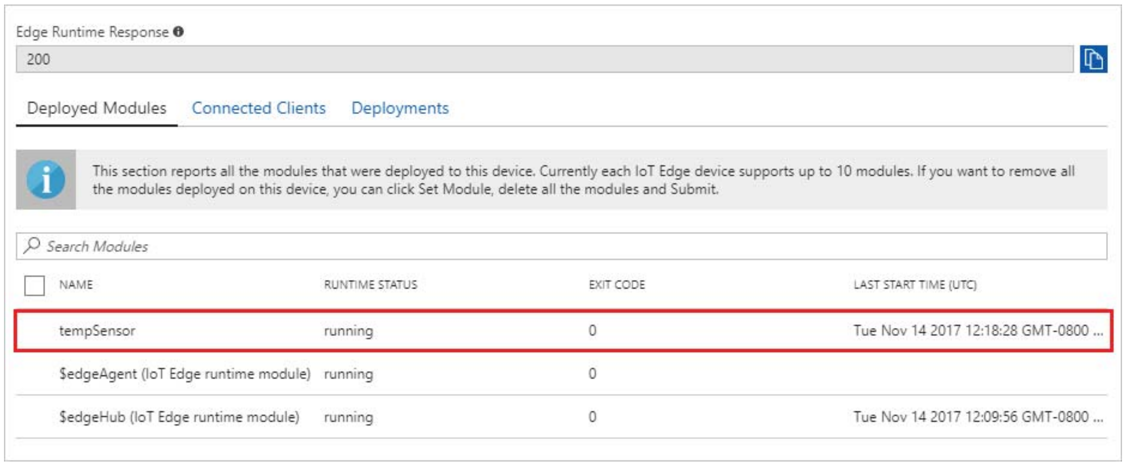
Desired Status
running

Module twin's desired properties

☐ Enable

```
{
  "properties.desired": {}
}
```

8. Back in the **Add modules** step, select **Next**.
9. In the **Specify routes** step, select **Next**.
10. In the **Review template** step, select **Submit**.
11. Return to the device details page and select **Refresh**. You should see the new tempSensor module running along the IoT Edge runtime.


Edge Runtime Response
200

This section reports all the modules that were deployed to this device. Currently each IoT Edge device supports up to 10 modules. If you want to remove all the modules deployed on this device, you can click Set Module, delete all the modules and Submit.

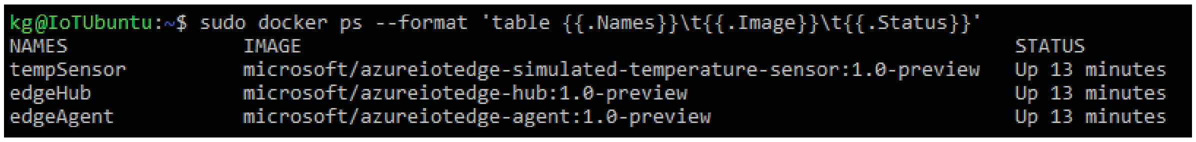
| NAME | RUNTIME STATUS | EXIT CODE | LAST START TIME (UTC) |
|---|---|---|---|
| tempSensor | running | 0 | Tue Nov 14 2017 12:18:28 GMT-0800 ... |
| $edgeAgent (IoT Edge runtime module) | running | 0 | |
| $edgeHub (IoT Edge runtime module) | running | 0 | Tue Nov 14 2017 12:09:56 GMT-0800 ... |

## View generated data

In this tutorial, you created a new IoT Edge device and installed the IoT Edge runtime on it. Then, you used the Azure portal to push an IoT Edge module to run on the device without having to make changes to the device itself. In this case, the module that you pushed creates environmental data that you can use for the tutorials.

Open the command prompt on the computer running your simulated device again. Confirm that the module deployed from the cloud is running on your IoT Edge device:

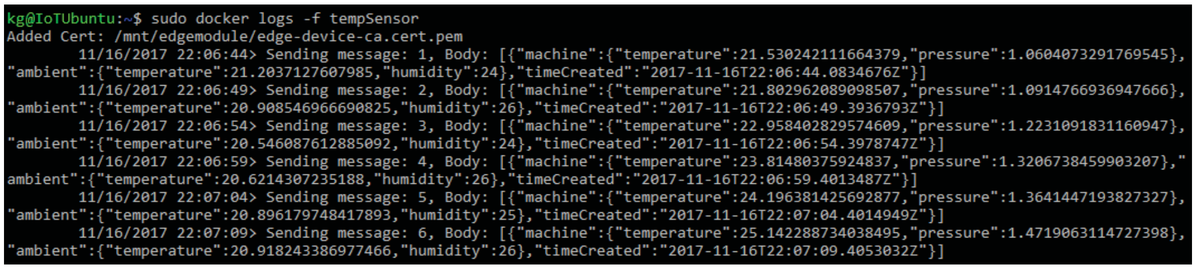
```
sudo docker ps
```



```
kg@IoTUbuntu:~$ sudo docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}'
NAMES               IMAGE                                                          STATUS
tempSensor          microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview   Up 13 minutes
edgeHub             microsoft/azureiotedge-hub:1.0-preview                          Up 13 minutes
edgeAgent           microsoft/azureiotedge-agent:1.0-preview                        Up 13 minutes
```

View the messages being sent from the tempSensor module to the cloud:

```
sudo docker logs -f tempSensor
```



```
kg@IoTUbuntu:~$ sudo docker logs -f tempSensor
Added Cert: /mnt/edgemodule/edge-device-ca.cert.pem
        11/16/2017 22:06:44> Sending message: 1, Body: [{"machine":{"temperature":21.530242111664379,"pressure":1.0604073291769545},
"ambient":{"temperature":21.2037127607985,"humidity":24},"timeCreated":"2017-11-16T22:06:44.0834676Z"}]
        11/16/2017 22:06:49> Sending message: 2, Body: [{"machine":{"temperature":21.802962089098507,"pressure":1.0914766936947666},
"ambient":{"temperature":20.908546966690825,"humidity":26},"timeCreated":"2017-11-16T22:06:49.3936793Z"}]
        11/16/2017 22:06:54> Sending message: 3, Body: [{"machine":{"temperature":22.958402829574609,"pressure":1.2231091831160947},
"ambient":{"temperature":20.546087612885092,"humidity":24},"timeCreated":"2017-11-16T22:06:54.3978747Z"}]
        11/16/2017 22:06:59> Sending message: 4, Body: [{"machine":{"temperature":23.81480375924837,"pressure":1.3206738459903207},"
ambient":{"temperature":20.6214307235188,"humidity":26},"timeCreated":"2017-11-16T22:06:59.4013487Z"}]
        11/16/2017 22:07:04> Sending message: 5, Body: [{"machine":{"temperature":24.196381425692877,"pressure":1.3641447193827327},
"ambient":{"temperature":20.896179748417893,"humidity":25},"timeCreated":"2017-11-16T22:07:04.4014949Z"}]
        11/16/2017 22:07:09> Sending message: 6, Body: [{"machine":{"temperature":25.142288734038495,"pressure":1.4719063114727398},
"ambient":{"temperature":20.918243386977466,"humidity":26},"timeCreated":"2017-11-16T22:07:09.4053032Z"}]
```

You can also view the telemetry the device is sending by using the IoT Hub explorer tool.

## Next steps

In this tutorial, you created a new IoT Edge device and used the Azure IoT Edge cloud interface to deploy code onto the device. Now, you have a simulated device generating raw data about its environment.

This tutorial is the prerequisite for all of the other IoT Edge tutorials. You can continue on to any of the other tutorials to learn how Azure IoT Edge can help you turn this data into business insights at the edge.

Develop and deploy C# code as a module

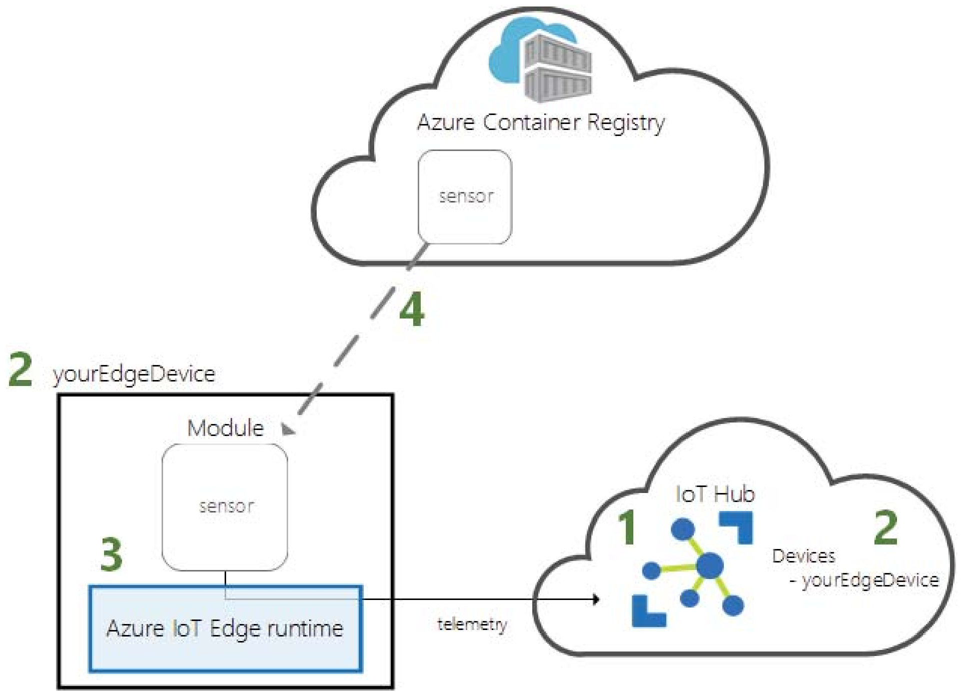
# Deploy Azure IoT Edge on a simulated device in Windows - preview

4/9/2018 • 5 min to read • Edit Online

Azure IoT Edge enables you to perform analytics and data processing on your devices, instead of having to push all the data to the cloud. The IoT Edge tutorials demonstrate how to deploy different types of modules, built from Azure services or custom code, but first you need a device to test.

In this tutorial you learn how to:

1. Create an IoT Hub
2. Register an IoT Edge device
3. Start the IoT Edge runtime
4. Deploy a module



The simulated device that you create in this tutorial is a monitor on a wind turbine that generates temperature, humidity, and pressure data. You're interested in this data because your turbines perform at different levels of efficiency depending on the weather conditions. The other Azure IoT Edge tutorials build upon the work you do here by deploying modules that analyze the data for business insights.

## Prerequisites

This tutorial assumes that you're using a computer or virtual machine running Windows to simulate an Internet of Things device.

**TIP**

If you're running Windows in a virtual machine, enable nested virtualization and allocate at least 2GB memory.

1. Make sure you're using a supported Windows version:

- Windows 10
- Windows Server

2. Install Docker for Windows and make sure it's running.
3. Install Python 2.7 on Windows and make sure you can use the pip command.
4. Run the following command to download the IoT Edge control script.

```
pip install -U azure-iot-edge-runtime-ctl
```

**NOTE**

Azure IoT Edge can run either Windows containers or Linux containers. If you're running one of the following Windows versions, you can use Windows containers:

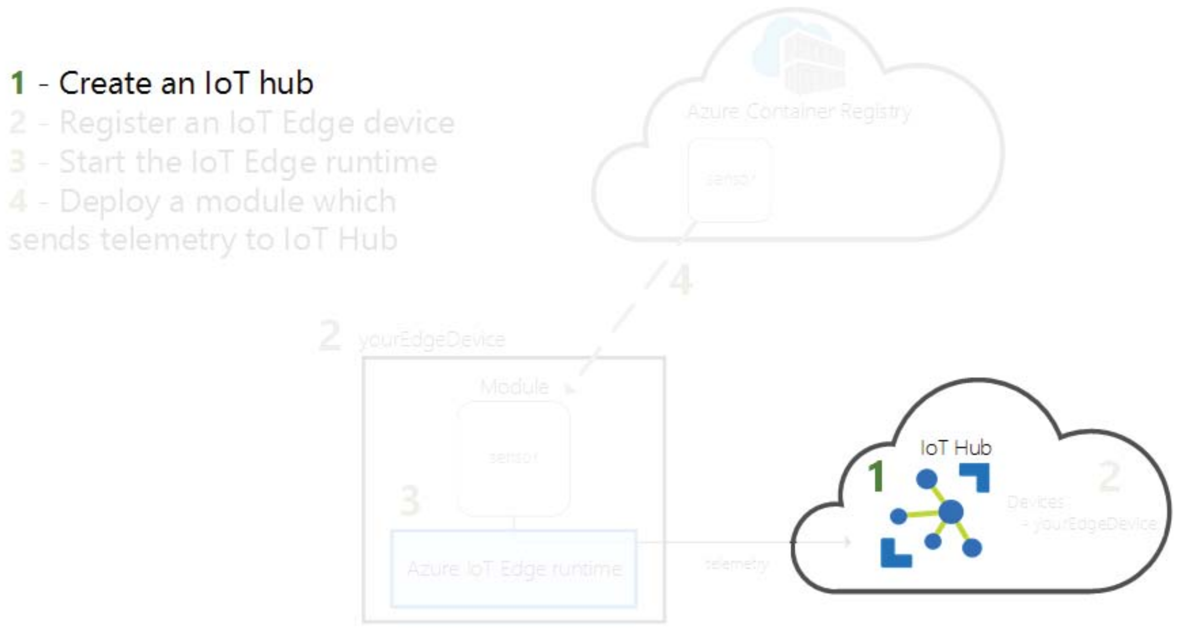
- Windows 10 Fall Creators Update
- Windows Server 1709 (Build 16299)
- Windows IoT Core (Build 16299) on a x64-based device

For Windows IoT Core, follow the instructions in Install the IoT Edge runtime on Windows IoT Core. Otherwise, simply configure Docker to use Windows containers. Use the following command to validate your prerequisites:

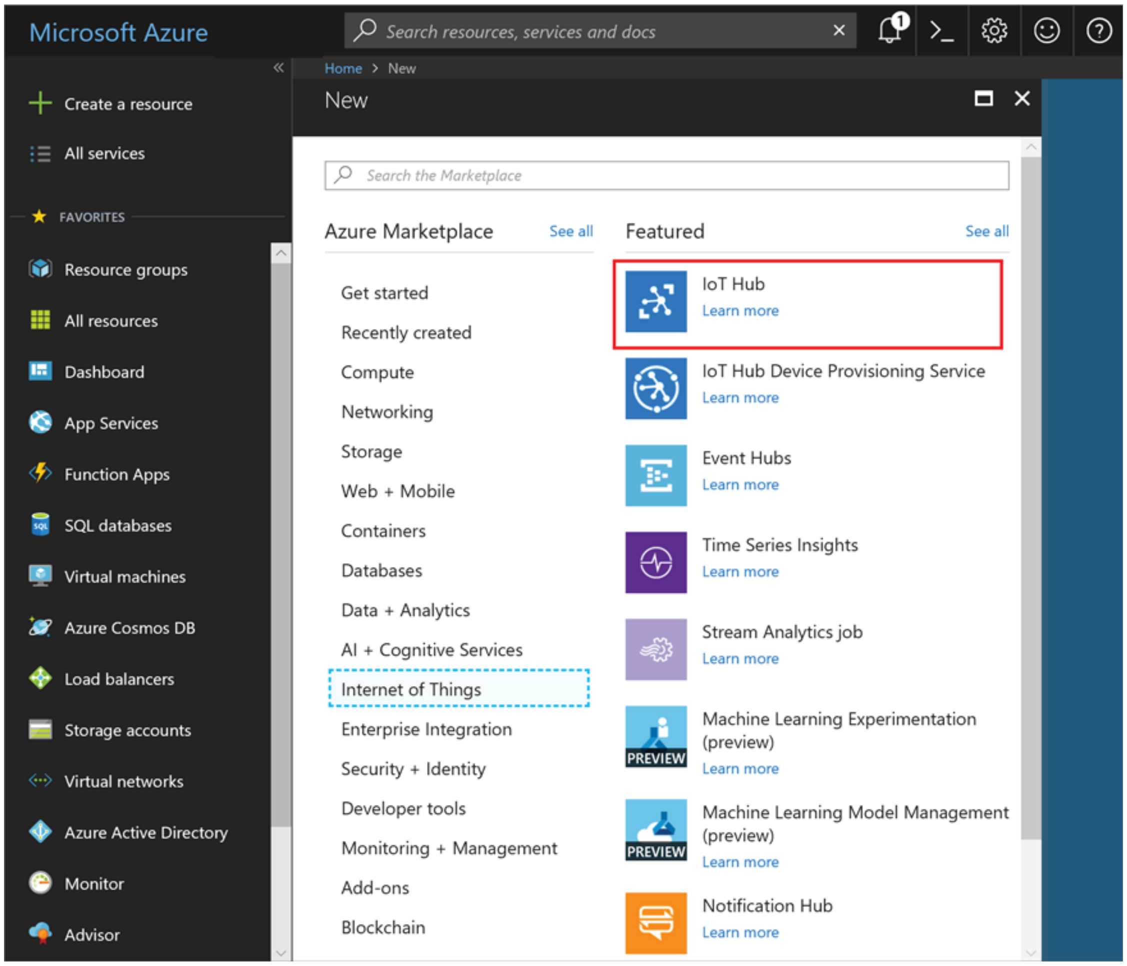
```
Invoke-Expression (Invoke-WebRequest -useb https://aka.ms/iotedgewin)
```

## Create an IoT hub

Start the tutorial by creating your IoT Hub.



1. Sign in to the Azure portal.
2. Select **Create a resource** > **Internet of Things** > **IoT Hub**.



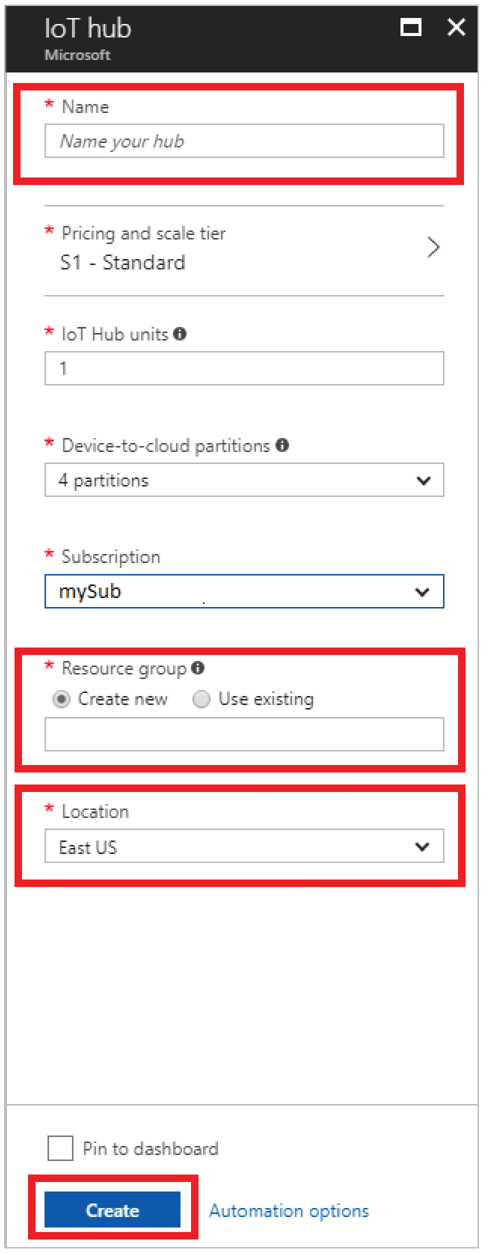
3. In the **IoT hub** pane, enter the following information for your IoT hub:

- **Name**: Create a name for your IoT hub. If the name you enter is valid, a green check mark appears.

**IMPORTANT**

The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.

- **Pricing and scale tier**: For this tutorial, select the **F1 - Free** tier. For more information, see the Pricing and scale tier.
- **Resource group**: Create a resource group to host the IoT hub or use an existing one. For more information, see Use resource groups to manage your Azure resources
- **Location**: Select the closest location to you.
- **Pin to dashboard**: Check this option for easy access to your IoT hub from the dashboard.


IoT hub
Microsoft

* Name
Name your hub

* Pricing and scale tier
S1 - Standard

* IoT Hub units
1

* Device-to-cloud partitions
4 partitions

* Subscription
mySub

* Resource group
[✓ Create new] Use existing

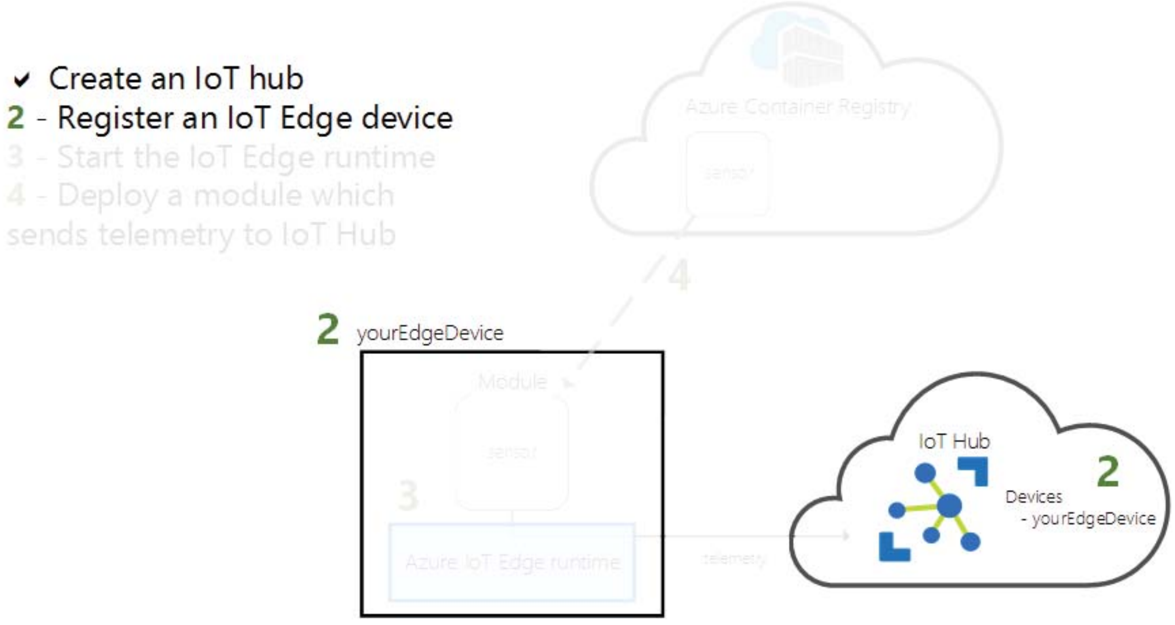
* Location
East US

Pin to dashboard

4. Click **Create**. Your IoT hub might take a few minutes to create. You can monitor the progress in the **Notifications** pane.

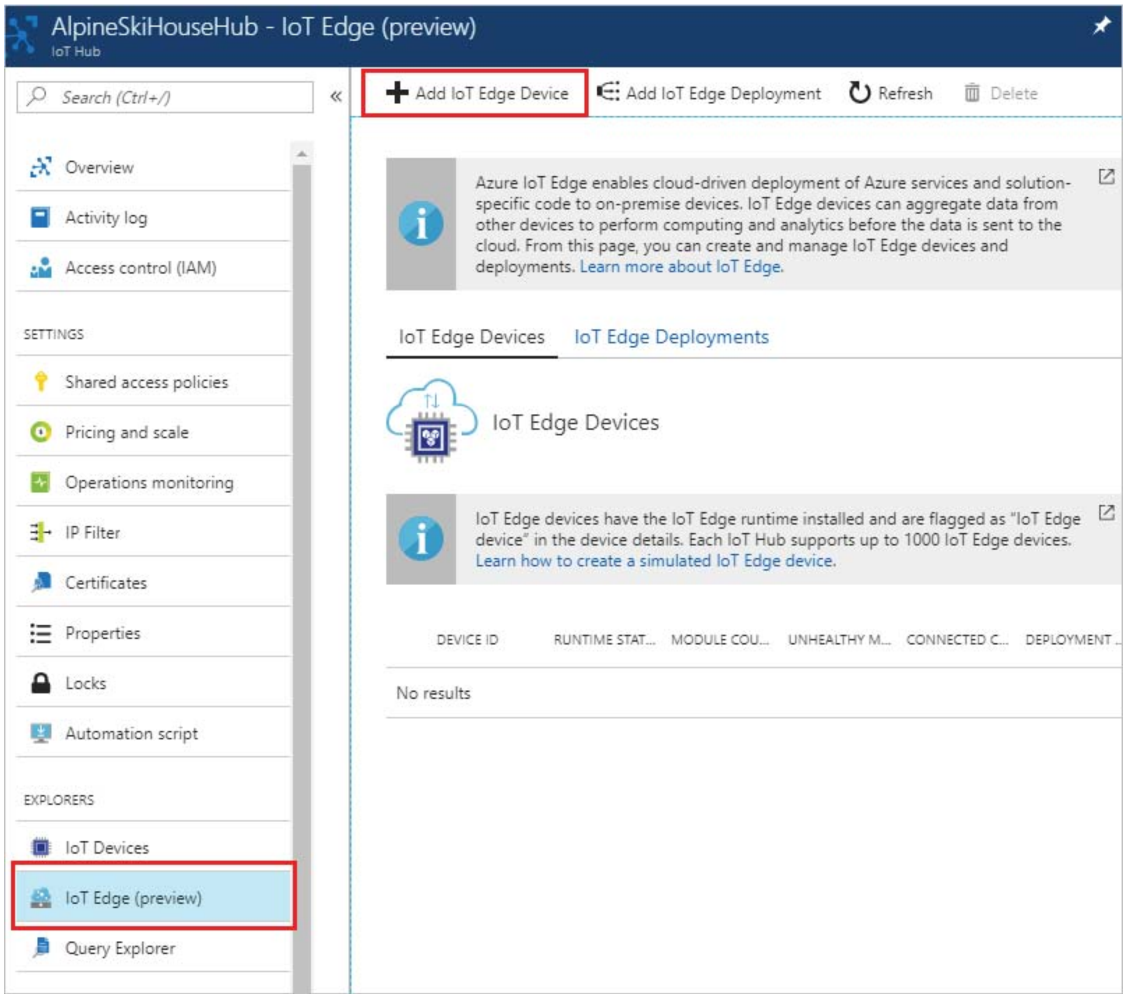
## Register an IoT Edge device

Register an IoT Edge device with your newly created IoT Hub.



Create a device identity for your simulated device so that it can communicate with your IoT hub. Since IoT Edge devices behave and can be managed differently than typical IoT devices, you declare this to be an IoT Edge device from the beginning.

1. In the Azure portal, navigate to your IoT hub.
2. Select **IoT Edge (preview)** then select **Add IoT Edge Device**.



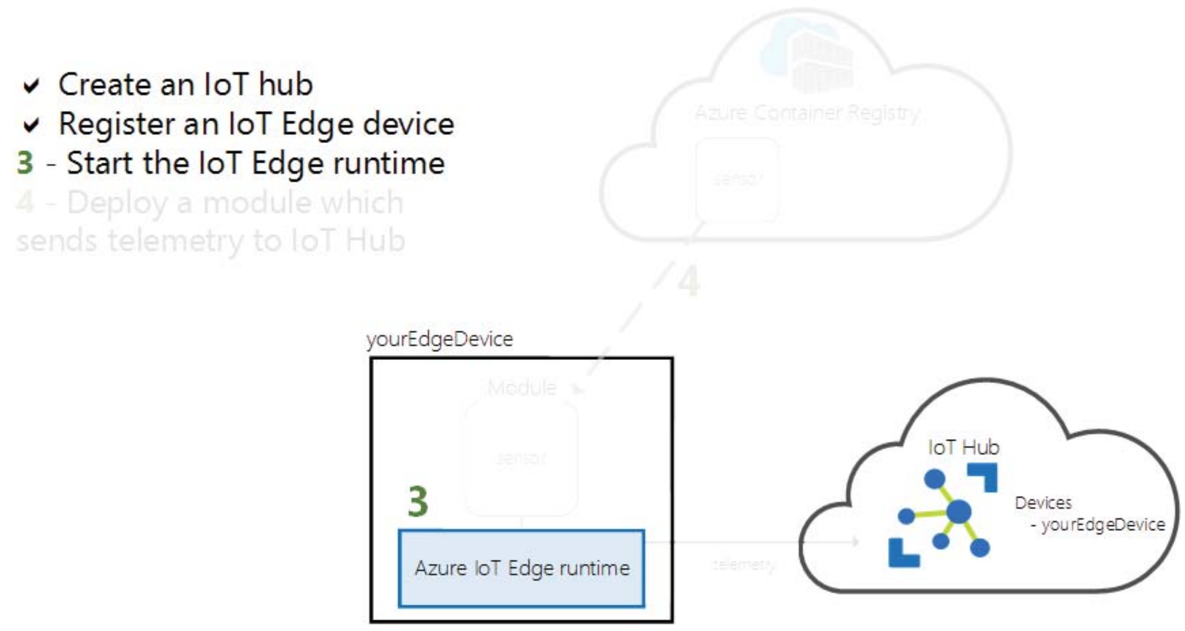
Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. Learn more about IoT Edge.

IoT Edge devices have the IoT Edge runtime installed and are flagged as "IoT Edge device" in the device details. Each IoT Hub supports up to 1000 IoT Edge devices. Learn how to create a simulated IoT Edge device.

3. Give your simulated device a unique device ID.
4. Select **Save** to add your device.
5. Select your new device from the list of devices.
6. Copy the value for **Connection string—primary key** and save it. You'll use this value to configure the IoT Edge runtime in the next section.

## Configure the IoT Edge runtime

Install and start the Azure IoT Edge runtime on your device.



The IoT Edge runtime is deployed on all IoT Edge devices. It comprises two modules. The **IoT Edge agent** facilitates deployment and monitoring of modules on the IoT Edge device. The **IoT Edge hub** manages communications between modules on the IoT Edge device, and between the device and IoT Hub. When you configure the runtime on your new device, only the IoT Edge agent will start at first. The IoT Edge hub comes later when you deploy a module.

Configure the runtime with your IoT Edge device connection string from the previous section.

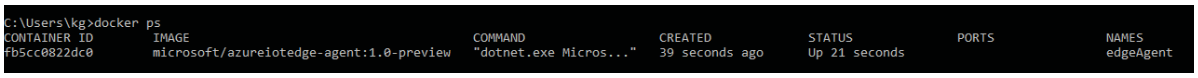
```
iotedgectl setup --connection-string "{device connection string}" --nopass
```

Start the runtime.

```
iotedgectl start
```

Check Docker to see that the IoT Edge agent is running as a module.

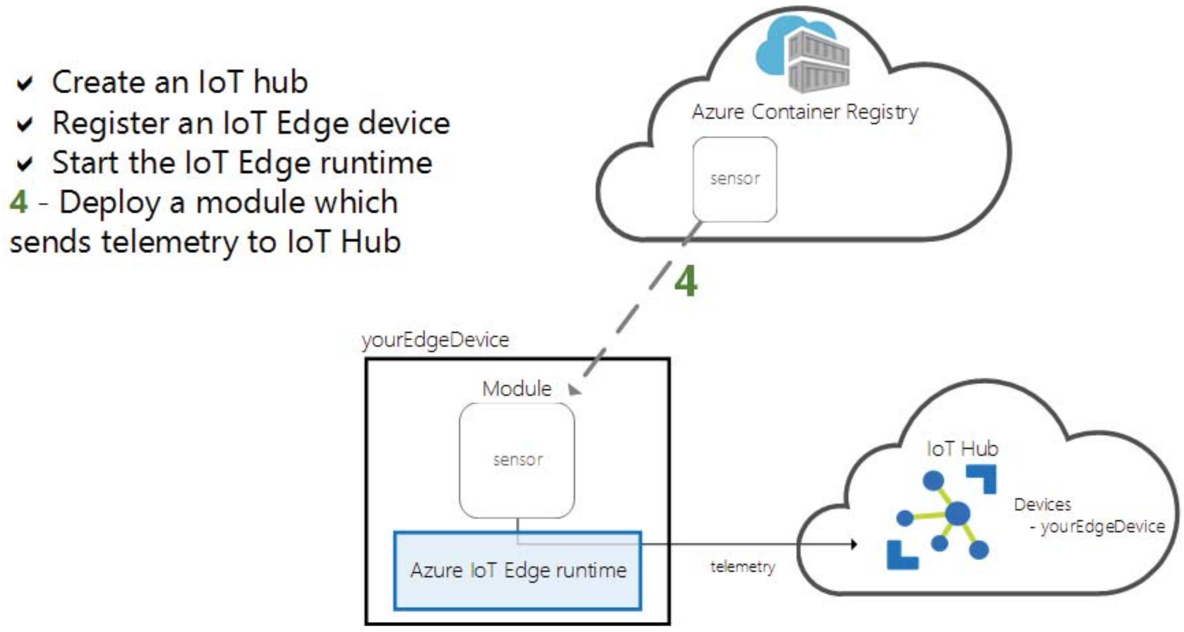
```
docker ps
```



```
C:\Users\kg>docker ps
CONTAINER ID        IMAGE                                       COMMAND                  CREATED             STATUS              PORTS               NAMES
fb5cc0822dc0        microsoft/azureiotedge-agent:1.0-preview    "dotnet.exe Micros..."   39 seconds ago      Up 21 seconds                           edgeAgent
```

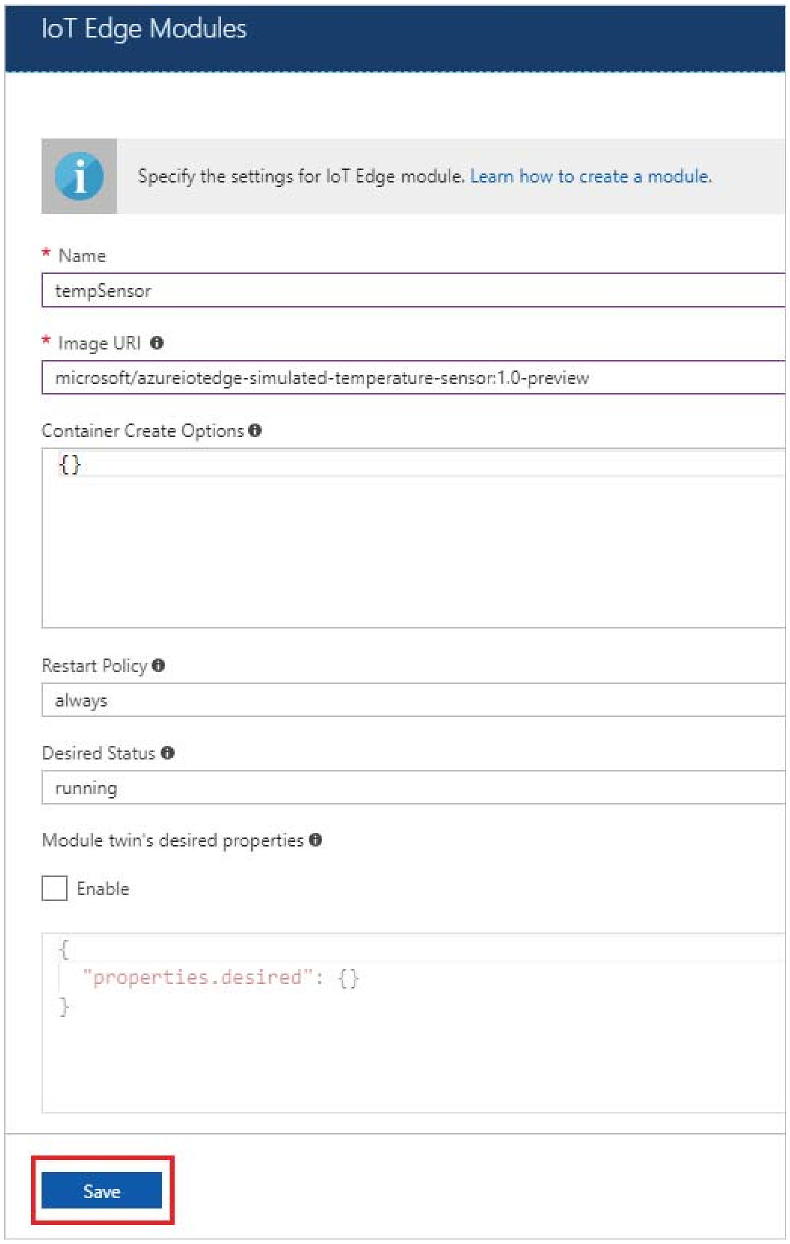
## Deploy a module

Manage your Azure IoT Edge device from the cloud to deploy a module which will send telemetry data to IoT Hub.



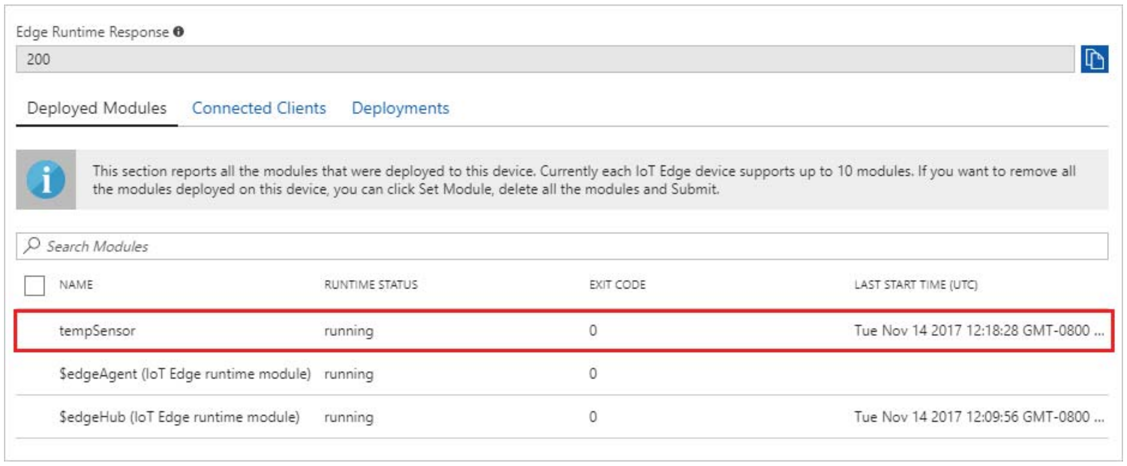
One of the key capabilities of Azure IoT Edge is being able to deploy modules to your IoT Edge devices from the cloud. An IoT Edge module is an executable package implemented as a container. In this section, you deploy a module that generates telemetry for your simulated device.

1. In the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. Select **Add IoT Edge Module**.
5. In the **Name** field, enter `tempSensor`.
6. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
7. Leave the other settings unchanged, and select **Save**.


IoT Edge Modules
Specify the settings for IoT Edge module. Learn how to create a module.

* Name
tempSensor
* Image URI
microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview
Container Create Options
```
{}
```
Restart Policy
always
Desired Status
running
Module twin's desired properties
Enable
```
{
  "properties.desired": {}
}
```

8. Back in the **Add modules** step, select **Next**.
9. In the **Specify routes** step, select **Next**.
10. In the **Review template** step, select **Submit**.
11. Return to the device details page and select **Refresh**. You should see the new tempSensor module running along the IoT Edge runtime.


Edge Runtime Response
200

This section reports all the modules that were deployed to this device. Currently each IoT Edge device supports up to 10 modules. If you want to remove all the modules deployed on this device, you can click Set Module, delete all the modules and Submit.

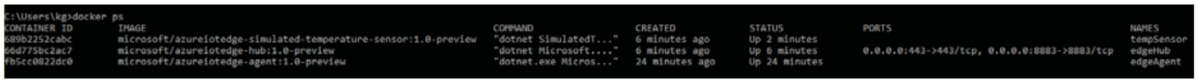
| NAME | RUNTIME STATUS | EXIT CODE | LAST START TIME (UTC) |
|---|---|---|---|
| tempSensor | running | 0 | Tue Nov 14 2017 12:18:28 GMT-0800 ... |
| $edgeAgent (IoT Edge runtime module) | running | 0 | |
| $edgeHub (IoT Edge runtime module) | running | 0 | Tue Nov 14 2017 12:09:56 GMT-0800 ... |

## View generated data

In this tutorial, you created a new IoT Edge device and installed the IoT Edge runtime on it. Then, you used the Azure portal to push an IoT Edge module to run on the device without having to make changes to the device itself. In this case, the module that you pushed creates environmental data that you can use for the tutorials.

Open the command prompt on the computer running your simulated device again. Confirm that the module deployed from the cloud is running on your IoT Edge device.

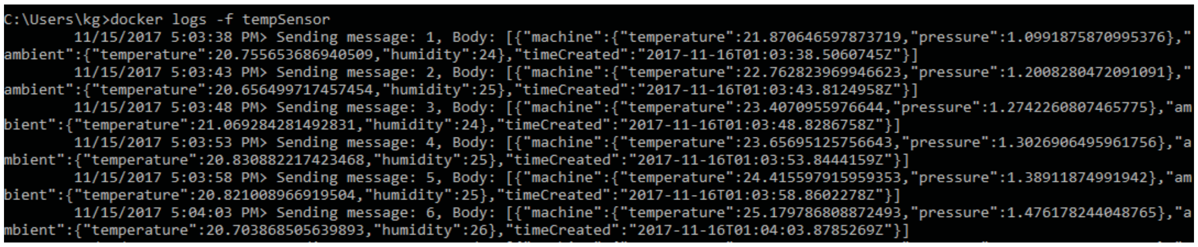
```
docker ps
```



```
C:\Users\kg>docker ps
CONTAINER ID        IMAGE                                                         COMMAND                  CREATED             STATUS              PORTS                                            NAMES
689b2252cabc        microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview   "dotnet SimulatedT..."   6 minutes ago       Up 2 minutes                                                         tempSensor
66d775bc2ac7        microsoft/azureiotedge-hub:1.0-preview                        "dotnet Microsoft...."   6 minutes ago       Up 6 minutes        0.0.0.0:443->443/tcp, 0.0.0.0:8883->8883/tcp   edgeHub
fb5cc0822dc0        microsoft/azureiotedge-agent:1.0-preview                      "dotnet.exe Micros..."   24 minutes ago      Up 24 minutes                                                        edgeAgent
```

View the messages being sent from the tempSensor module to the cloud.

```
docker logs -f tempSensor
```



```
C:\Users\kg>docker logs -f tempSensor
        11/15/2017 5:03:38 PM> Sending message: 1, Body: [{"machine":{"temperature":21.870646597873719,"pressure":1.0991875870995376},"
ambient":{"temperature":20.755653686940509,"humidity":24},"timeCreated":"2017-11-16T01:03:38.5060745Z"}]
        11/15/2017 5:03:43 PM> Sending message: 2, Body: [{"machine":{"temperature":22.762823969946623,"pressure":1.2008280472091091},"
ambient":{"temperature":20.656499717457454,"humidity":25},"timeCreated":"2017-11-16T01:03:43.8124958Z"}]
        11/15/2017 5:03:48 PM> Sending message: 3, Body: [{"machine":{"temperature":23.4070955976644,"pressure":1.2742260807465775},"am
bient":{"temperature":21.069284281492831,"humidity":24},"timeCreated":"2017-11-16T01:03:48.8286758Z"}]
        11/15/2017 5:03:53 PM> Sending message: 4, Body: [{"machine":{"temperature":23.65695125756643,"pressure":1.3026906495961756},"a
mbient":{"temperature":20.830882217423468,"humidity":25},"timeCreated":"2017-11-16T01:03:53.8444159Z"}]
        11/15/2017 5:03:58 PM> Sending message: 5, Body: [{"machine":{"temperature":24.415597915959353,"pressure":1.38911874991942},"am
bient":{"temperature":20.821008966919504,"humidity":25},"timeCreated":"2017-11-16T01:03:58.8602278Z"}]
        11/15/2017 5:04:03 PM> Sending message: 6, Body: [{"machine":{"temperature":25.179786808872493,"pressure":1.476178244048765},"a
mbient":{"temperature":20.703868505639893,"humidity":26},"timeCreated":"2017-11-16T01:04:03.8785269Z"}]
```

You can also view the telemetry the device is sending by using the IoT Hub explorer tool.

## Next steps

In this tutorial, you created a new IoT Edge device and used the Azure IoT Edge cloud interface to deploy code onto the device. Now, you have a simulated device generating raw data about its environment.

This tutorial is the prerequisite for all of the other IoT Edge tutorials. You can continue on to any of the other tutorials to learn how Azure IoT Edge can help you turn this data into business insights at the edge.

Develop and deploy C# code as a module

# Develop and deploy a C# IoT Edge module to your simulated device - preview

5/8/2018 • 9 min to read • Edit Online

You can use IoT Edge modules to deploy code that implements your business logic directly to your IoT Edge devices. This tutorial walks you through creating and deploying an IoT Edge module that filters sensor data. You'll use the simulated IoT Edge device that you created in the Deploy Azure IoT Edge on a simulated device in Windows or Linux tutorials. In this tutorial, you learn how to:

- Use Visual Studio Code to create an IoT Edge module based on .NET core 2.0
- Use Visual Studio Code and Docker to create a docker image and publish it to your registry
- Deploy the module to your IoT Edge device
- View generated data

The IoT Edge module that you create in this tutorial filters the temperature data generated by your device. It only sends messages upstream if the temperature is above a specified threshold. This type of analysis at the edge is useful for reducing the amount of data communicated to and stored in the cloud.

## Prerequisites

- The Azure IoT Edge device that you created in the quickstart or first tutorial.
- The primary key connection string for the IoT Edge device.
- Visual Studio Code.
- Azure IoT Edge extension for Visual Studio Code.
- C# for Visual Studio Code (powered by OmniSharp) extension.
- Docker on the same computer that has Visual Studio Code. The Community Edition (CE) is sufficient for this tutorial.
- .NET Core 2.0 SDK.

## Create a container registry

In this tutorial, you use the Azure IoT Edge extension for VS Code to build a module and create a **container image** from the files. Then you push this image to a **registry** that stores and manages your images. Finally, you deploy your image from your registry to run on your IoT Edge device.

You can use any Docker-compatible registry for this tutorial. Two popular Docker registry services available in the cloud are Azure Container Registry and Docker Hub. This tutorial uses Azure Container Registry.

1. In the Azure portal, select **Create a resource** > **Containers** > **Azure Container Registry**.
2. Give your registry a name, choose a subscription, choose a resource group, and set the SKU to **Basic**.
3. Select **Create**.
4. Once your container registry is created, navigate to it and select **Access keys**.
5. Toggle **Admin user** to **Enable**.
6. Copy the values for **Login server**, **Username**, and **Password**. You'll use these values later in the tutorial when you publish the Docker image to your registry, and when you add the registry credentials to the Edge runtime.

## Create an IoT Edge module project

The following steps show you how to create an IoT Edge module based on .NET core 2.0 using Visual Studio Code and the Azure IoT Edge extension.

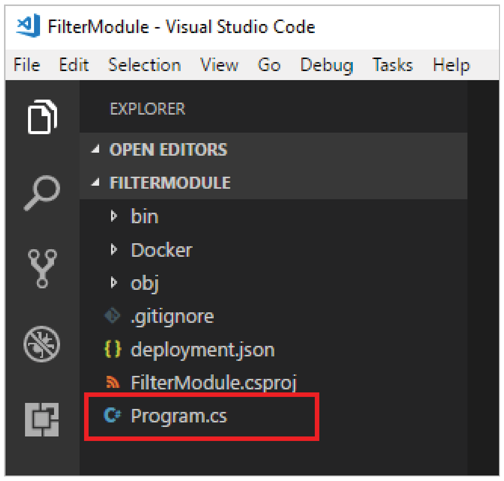
1. In Visual Studio Code, select **View** > **Integrated Terminal** to open the VS Code integrated terminal.
2. In the integrated terminal, enter the following command to install (or update) the **AzureIoTEdgeModule** template in dotnet:

   ```
   dotnet new -i Microsoft.Azure.IoT.Edge.Module
   ```

3. Create a project for the new module. The following command creates the project folder, **FilterModule**, with your container repository. The second parameter should be in the form of `<your container registry name>.azurecr.io` if you are using Azure container registry. Enter the following command in the current working folder:

   ```
   dotnet new aziotedgemodule -n FilterModule -r <your container registry address>/filtermodule
   ```

4. Select **File** > **Open Folder**.
5. Browse to the **FilterModule** folder and click **Select Folder** to open the project in VS Code.
6. In VS Code explorer, click **Program.cs** to open it.



7. At the top of the **FilterModule** namespace, add three `using` statements for types used later on:

   ```
   using System.Collections.Generic;     // for KeyValuePair<>
   using Microsoft.Azure.Devices.Shared; // for TwinCollection
   using Newtonsoft.Json;                // for JsonConvert
   ```

8. Add the `temperatureThreshold` variable to the **Program** class. This variable sets the value that the measured temperature must exceed in order for the data to be sent to IoT Hub.

   ```
   static int temperatureThreshold { get; set; } = 25;
   ```

9. Add the `MessageBody`, `Machine`, and `Ambient` classes to the **Program** class. These classes define the expected schema for the body of incoming messages.

```
class MessageBody
{
    public Machine machine {get;set;}
    public Ambient ambient {get; set;}
    public string timeCreated {get; set;}
}
class Machine
{
   public double temperature {get; set;}
   public double pressure {get; set;}
}
class Ambient
{
   public double temperature {get; set;}
   public int humidity {get; set;}
}
```

10. In the **Init** method, the code creates and configures a **DeviceClient** object. This object allows the module to connect to the local Azure IoT Edge runtime to send and receive messages. The connection string used in the **Init** method is supplied to the module by IoT Edge runtime. After creating the **DeviceClient**, the code reads the TemperatureThreshold from the Module Twin's desired properties and registers a callback for receiving messages from the IoT Edge hub via the **input1** endpoint. Replace the `SetInputMessageHandlerAsync` method with a new one, and add a `SetDesiredPropertyUpdateCallbackAsync` method for desired properties updates. To make this change, replace the last line of the **Init** method with the following code:

```
// Register callback to be called when a message is received by the module
// await ioTHubModuleClient.SetImputMessageHandlerAsync("input1", PipeMessage, iotHubModuleClient);

// Read TemperatureThreshold from Module Twin Desired Properties
var moduleTwin = await ioTHubModuleClient.GetTwinAsync();
var moduleTwinCollection = moduleTwin.Properties.Desired;
try {
    temperatureThreshold = moduleTwinCollection["TemperatureThreshold"];
} catch(ArgumentOutOfRangeException e) {
    Console.WriteLine("Property TemperatureThreshold not exist");
}

// Attach callback for Twin desired properties updates
await ioTHubModuleClient.SetDesiredPropertyUpdateCallbackAsync(onDesiredPropertiesUpdate, null);

// Register callback to be called when a message is received by the module
await ioTHubModuleClient.SetInputMessageHandlerAsync("input1", FilterMessages, ioTHubModuleClient);
```

11. Add the `onDesiredPropertiesUpdate` method to the **Program** class. This method receives updates on the desired properties from the module twin, and updates the **temperatureThreshold** variable to match. All modules have their own module twin, which lets you configure the code running inside a module directly from the cloud.

```
static Task onDesiredPropertiesUpdate(TwinCollection desiredProperties, object userContext)
{
    try
    {
        Console.WriteLine("Desired property change:");
        Console.WriteLine(JsonConvert.SerializeObject(desiredProperties));

        if (desiredProperties["TemperatureThreshold"]!=null)
            temperatureThreshold = desiredProperties["TemperatureThreshold"];

    }
    catch (AggregateException ex)
    {
        foreach (Exception exception in ex.InnerExceptions)
        {
            Console.WriteLine();
            Console.WriteLine("Error when receiving desired property: {0}", exception);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine("Error when receiving desired property: {0}", ex.Message);
    }
    return Task.CompletedTask;
}
```

12. Replace the `PipeMessage` method with the `FilterMessages` method. This method is called whenever the module receives a message from the IoT Edge hub. It filters out messages that report temperatures below the temperature threshold set via the module twin. It also adds the **MessageType** property to the message with the value set to **Alert**.

```
static async Task<MessageResponse> FilterMessages(Message message, object userContext)
{
    var counterValue = Interlocked.Increment(ref counter);

    try {
        DeviceClient deviceClient = (DeviceClient)userContext;

        var messageBytes = message.GetBytes();
        var messageString = Encoding.UTF8.GetString(messageBytes);
        Console.WriteLine($"Received message {counterValue}: [{messageString}]");

        // Get message body
        var messageBody = JsonConvert.DeserializeObject<MessageBody>(messageString);

        if (messageBody != null && messageBody.machine.temperature > temperatureThreshold)
        {
            Console.WriteLine($"Machine temperature {messageBody.machine.temperature} " +
                $"exceeds threshold {temperatureThreshold}");
            var filteredMessage = new Message(messageBytes);
            foreach (KeyValuePair<string, string> prop in message.Properties)
            {
                filteredMessage.Properties.Add(prop.Key, prop.Value);
            }

            filteredMessage.Properties.Add("MessageType", "Alert");
            await deviceClient.SendEventAsync("output1", filteredMessage);
        }

        // Indicate that the message treatment is completed
        return MessageResponse.Completed;
    }
    catch (AggregateException ex)
    {
        foreach (Exception exception in ex.InnerExceptions)
        {
            Console.WriteLine();
            Console.WriteLine("Error in sample: {0}", exception);
        }
        // Indicate that the message treatment is not completed
        var deviceClient = (DeviceClient)userContext;
        return MessageResponse.Abandoned;
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine("Error in sample: {0}", ex.Message);
        // Indicate that the message treatment is not completed
        DeviceClient deviceClient = (DeviceClient)userContext;
        return MessageResponse.Abandoned;
    }
}
```

13. Save this file.

## Create a Docker image and publish it to your registry

1. Sign in to Docker by entering the following command in the VS Code integrated terminal:

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

To find the user name, password and login server to use in this command, go to the Azure portal. From **All resources**, click the tile for your Azure container registry to open its properties, then click **Access keys**. Copy the values in the **Username**, **password**, and **Login server** fields.

2. In VS Code explorer, Right-click the **module.json** file and click **Build and Push IoT Edge module Docker image**. In the pop-up dropdown box at the top of the VS Code window, select your container platform, either **amd64** for Linux container or **windows-amd64** for Windows container. VS Code then builds your code, containerize the `FilterModule.dll` and push it to the container registry you specified.

3. You can get the full container image address with tag in the VS Code integrated terminal. For more infomation about the build and push definition, you can refer to the `module.json` file.

## Add registry credentials to Edge runtime

Add the credentials for your registry to the Edge runtime on the computer where you are running your Edge device. These credentials give the runtime access to pull the container.

- For Windows, run the following command:

```
iotedgectl login --address <your container registry address> --username <username> --password
<password>
```

- For Linux, run the following command:

```
sudo iotedgectl login --address <your container registry address> --username <username> --password
<password>
```

## Run the solution

1. In the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. Check that the **tempSensor** module is automatically populated. If it's not, use the following steps to add it:
   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `tempSensor`.
   c. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
   d. Leave the other settings unchanged and click **Save**.
5. Add the **filterModule** module that you created in the previous sections.

   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `filterModule`.
   c. In the **Image URI** field, enter your image address; for example `<your container registry address>/filtermodule:0.0.1-amd64`. The full image address can be found from the previous section.
   d. Check the **Enable** box so that you can edit the module twin.
   e. Replace the JSON in the text box for the module twin with the following JSON:

   ```
   {
      "properties.desired":{
         "TemperatureThreshold":25
      }
   }
   ```

   f. Click **Save**.

6. Click **Next**.
7. In the **Specify Routes** step, copy the JSON below into the text box. Modules publish all messages to the Edge runtime. Declarative rules in the runtime define where the messages flow. In this tutorial, you need two routes. The first route transports messages from the temperature sensor to the filter module via the "input1" endpoint, which is the endpoint that you configured with the **FilterMessages** handler. The second route transports messages from the filter module to IoT Hub. In this route, `upstream` is a special destination that tells Edge Hub to send messages to IoT Hub.

```
{
   "routes":{
      "sensorToFilter":"FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/filterModule/inputs/input1\")",
      "filterToIoTHub":"FROM /messages/modules/filterModule/outputs/output1 INTO $upstream"
   }
}
```

8. Click **Next**.
9. In the **Review Template** step, click **Submit**.
10. Return to the IoT Edge device details page and click **Refresh**. You should see the new **filtermodule** running along with the **tempSensor** module and the **IoT Edge runtime**.

## View generated data

To monitor device to cloud messages sent from your IoT Edge device to your IoT hub:

1. Configure the Azure IoT Toolkit extension with connection string for your IoT hub:
   a. Open the VS Code explorer by selecting **View** > **Explorer**.
   b. In the explorer, click **IOT HUB DEVICES** and then click **...**. Click **Set IoT Hub Connection String** and enter the connection string for the IoT hub that your IoT Edge device connects to in the pop-up window.

      To find the connection string, click the tile for your IoT hub in the Azure portal and then click **Shared access policies**. In **Shared access policies**, click the **iothubowner** policy and copy the IoT Hub connection string in the **iothubowner** window.
2. To monitor data arriving at the IoT hub, select **View** > **Command Palette** and search for the **IoT: Start monitoring D2C message** menu command.
3. To stop monitoring data, use the **IoT: Stop monitoring D2C message** menu command.

## Next steps

In this tutorial, you created an IoT Edge module that contains code to filter raw data generated by your IoT Edge device. You can continue on to either of the following tutorials to learn about other ways that Azure IoT Edge can help you turn data into business insights at the edge.

Deploy Azure Function as a module Deploy Azure Stream Analytics as a module

# Develop and deploy a Python IoT Edge module to your simulated device - preview

4/26/2018 • 8 min to read • Edit Online

You can use IoT Edge modules to deploy code that implements your business logic directly to your IoT Edge devices. This tutorial walks you through creating and deploying an IoT Edge module that filters sensor data. You'll use the simulated IoT Edge device that you created in the Deploy Azure IoT Edge on a simulated device in Windows or Linux tutorials. In this tutorial, you learn how to:

- Use Visual Studio Code to create an IoT Edge Python module
- Use Visual Studio Code and Docker to create a docker image and publish it to your registry
- Deploy the module to your IoT Edge device
- View generated data

The IoT Edge module that you create in this tutorial filters the temperature data generated by your device. It only sends messages upstream if the temperature is above a specified threshold. This type of analysis at the edge is useful for reducing the amount of data communicated to and stored in the cloud.

**IMPORTANT**

Currently the Python module can only be run in amd64 Linux containers; it cannot run in Windows containers or ARM-based containers.

## Prerequisites

- The Azure IoT Edge device that you created in the quickstart or first tutorial.
- The primary key connection string for the IoT Edge device.
- Visual Studio Code.
- Azure IoT Edge extension for Visual Studio Code.
- Python extension for Visual Studio Code.
- Docker on the same computer that has Visual Studio Code. The Community Edition (CE) is sufficient for this tutorial.
- Python.
- Pip for installing Python packages (typically included with your Python installation).

## Create a container registry

In this tutorial, you use the Azure IoT Edge extension for VS Code to build a module and create a **container image** from the files. Then you push this image to a **registry** that stores and manages your images. Finally, you deploy your image from your registry to run on your IoT Edge device.

You can use any Docker-compatible registry for this tutorial. Two popular Docker registry services available in the cloud are Azure Container Registry and Docker Hub. This tutorial uses Azure Container Registry.

1. In the Azure portal, select **Create a resource** > **Containers** > **Azure Container Registry**.
2. Give your registry a name, choose a subscription, choose a resource group, and set the SKU to **Basic**.
3. Select **Create**.
4. Once your container registry is created, navigate to it and select **Access keys**.

5. Toggle **Admin user** to **Enable**.
6. Copy the values for **Login server**, **Username**, and **Password**. You'll use these values later in the tutorial.

## Create an IoT Edge module project

The following steps show you how to create an IoT Edge Python module using Visual Studio Code and the Azure IoT Edge extension.

1. In Visual Studio Code, select **View** > **Integrated Terminal** to open the VS Code integrated terminal.
2. In the integrated terminal, enter the following command to install (or update) **cookiecutter** (we suggest doing this either into a virtual environment or as a user install as shown below):

   ```
   pip install --upgrade --user cookiecutter
   ```

3. Create a project for the new module. The following command creates the project folder, **FilterModule**, with your container repository. The parameter of `image_repository` should be in the form of `<your container registry name>.azurecr.io/filtermodule` if you are using Azure container registry. Enter the following command in the current working folder:

   ```
   cookiecutter --no-input https://github.com/Azure/cookiecutter-azure-iot-edge-module
   module_name=FilterModule image_repository=<your container registry address>/filtermodule
   ```

4. Select **File** > **Open Folder**.
5. Browse to the **FilterModule** folder and click **Select Folder** to open the project in VS Code.
6. In VS Code explorer, click **main.py** to open it.
7. At the top of the **FilterModule** namespace, import the `json` library:

   ```
   import json
   ```

8. Add the `TEMPERATURE_THRESHOLD`, `RECEIVE_CALLBACKS`, and `TWIN_CALLBACKS` under the global counters. The temperature threshold sets the value that the measured temperature must exceed in order for the data to be sent to IoT Hub.

   ```
   TEMPERATURE_THRESHOLD = 25
   TWIN_CALLBACKS = RECEIVE_CALLBACKS = 0
   ```

9. Update the function `receive_message_callback` with below content.

```
# receive_message_callback is invoked when an incoming message arrives on the specified
# input queue (in the case of this sample, "input1").  Because this is a filter module,
# we will forward this message onto the "output1" queue.
def receive_message_callback(message, hubManager):
    global RECEIVE_CALLBACKS
    global TEMPERATURE_THRESHOLD
    message_buffer = message.get_bytearray()
    size = len(message_buffer)
    message_text = message_buffer[:size].decode('utf-8')
    print("    Data: <<<{}>>> & Size={:d}".format(message_text, size))
    map_properties = message.properties()
    key_value_pair = map_properties.get_internals()
    print("    Properties: {}".format(key_value_pair))
    RECEIVE_CALLBACKS += 1
    print("    Total calls received: {:d}".format(RECEIVE_CALLBACKS))
    data = json.loads(message_text)
    if "machine" in data and "temperature" in data["machine"] and data["machine"]["temperature"] >
TEMPERATURE_THRESHOLD:
        map_properties.add("MessageType", "Alert")
        print("Machine temperature {} exceeds threshold {}".format(data["machine"]["temperature"],
TEMPERATURE_THRESHOLD))
    hubManager.forward_event_to_output("output1", message, 0)
    return IoTHubMessageDispositionResult.ACCEPTED
```

10. Add a new function `device_twin_callback`. This function will be invoked when the desired properties are updated.

```
# device_twin_callback is invoked when twin's desired properties are updated.
def device_twin_callback(update_state, payload, user_context):
    global TWIN_CALLBACKS
    global TEMPERATURE_THRESHOLD
    print("\nTwin callback called with:\nupdateStatus = {}\npayload = {}\ncontext =
{}".format(update_state, payload, user_context))
    data = json.loads(payload)
    if "desired" in data and "TemperatureThreshold" in data["desired"]:
        TEMPERATURE_THRESHOLD = data["desired"]["TemperatureThreshold"]
    if "TemperatureThreshold" in data:
        TEMPERATURE_THRESHOLD = data["TemperatureThreshold"]
    TWIN_CALLBACKS += 1
    print("Total calls confirmed: {:d}\n".format(TWIN_CALLBACKS))
```

11. In class `HubManager`, add a new line to the `__init__` method to initialize the `device_twin_callback` function you just added.

```
# sets the callback when a twin's desired properties are updated.
self.client.set_device_twin_callback(device_twin_callback, self)
```

12. Save this file.

## Create a Docker image and publish it to your registry

1. Sign in to Docker by entering the following command in the VS Code integrated terminal:

```
docker login -u <username> -p <password> <Login server>
```

   Use the user name, password, and login server that you copied from your Azure container registry when you created it.

2. In VS Code explorer, Right-click the **module.json** file and click **Build and Push IoT Edge module Docker**

**image**. In the pop-up dropdown box at the top of the VS Code window, select your container platform, for example, **amd64** for Linux container. VS Code containerize the `main.py` and required dependencies, then push it to the container registry you specified. It might take several minutes for your first time to build the image.

3. You can get the full container image address with tag in the VS Code integrated terminal. For more infomation about the build and push definition, you can refer to the `module.json` file.

## Add registry credentials to Edge runtime

Add the credentials for your registry to the Edge runtime on the computer where you are running your Edge device. These credentials give the runtime access to pull the container.

- For Windows, run the following command:

```
iotedgectl login --address <your container registry address> --username <username> --password <password>
```

- For Linux, run the following command:

```
sudo iotedgectl login --address <your container registry address> --username <username> --password
<password>
```

## Run the solution

1. In the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. Check that the **tempSensor** module is automatically populated. If it's not, use the following steps to add it:
   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `tempSensor`.
   c. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
   d. Leave the other settings unchanged and click **Save**.
5. Add the **filterModule** module that you created in the previous sections.
   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `filterModule`.
   c. In the **Image URI** field, enter your image address; for example `<your container registry address>/filtermodule:0.0.1-amd64`. The full image address can be found from the previous section.
   d. Check the **Enable** box so that you can edit the module twin.
   e. Replace the JSON in the text box for the module twin with the following JSON:

   ```
   {
      "properties.desired":{
         "TemperatureThreshold":25
      }
   }
   ```

   f. Click **Save**.
6. Click **Next**.

7. In the **Specify Routes** step, copy the JSON below into the text box. Modules publish all messages to the Edge runtime. Declarative rules in the runtime define where the messages flow. In this tutorial, you need two routes. The first route transports messages from the temperature sensor to the filter module via the "input1" endpoint, which is the endpoint that you configured with the **FilterMessages** handler. The second route transports messages from the filter module to IoT Hub. In this route, `upstream` is a special destination that tells Edge Hub to send messages to IoT Hub.

   ```
   {
      "routes":{
         "sensorToFilter":"FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
   BrokeredEndpoint(\"/modules/filterModule/inputs/input1\")",
         "filterToIoTHub":"FROM /messages/modules/filterModule/outputs/output1 INTO $upstream"
      }
   }
   ```

8. Click **Next**.

9. In the **Review Template** step, click **Submit**.
10. Return to the IoT Edge device details page and click **Refresh**. You should see the new **filtermodule** running along with the **tempSensor** module and the **IoT Edge runtime**.

## View generated data

To monitor device to cloud messages sent from your IoT Edge device to your IoT hub:

1. Configure the Azure IoT Toolkit extension with connection string for your IoT hub:

   a. Open the VS Code explorer by selecting **View** > **Explorer**.
   b. In the explorer, click **IOT HUB DEVICES** and then click **...**. Click **Set IoT Hub Connection String** and enter the connection string for the IoT hub that your IoT Edge device connects to in the pop-up window.

      To find the connection string, click the tile for your IoT hub in the Azure portal and then click **Shared access policies**. In **Shared access policies**, click the **iothubowner** policy and copy the IoT Hub connection string in the **iothubowner** window.

2. To monitor data arriving at the IoT hub, select **View** > **Command Palette** and search for the **IoT: Start monitoring D2C message** menu command.

3. To stop monitoring data, use the **IoT: Stop monitoring D2C message** menu command.

## Next steps

In this tutorial, you created an IoT Edge module that contains code to filter raw data generated by your IoT Edge device. You can continue on to either of the following tutorials to learn about other ways that Azure IoT Edge can help you turn data into business insights at the edge.

Deploy Azure Function as a module Deploy Azure Stream Analytics as a module

# Deploy Azure Stream Analytics as an IoT Edge module - preview

4/9/2018 • 5 min to read • Edit Online

IoT devices can produce large quantities of data. To reduce the amount of uploaded data or to eliminate the round-trip latency of an actionable insight, the data must sometimes be analyzed or processed before it reaches the cloud.

Azure IoT Edge takes advantage of pre-built Azure service IoT Edge modules for quick deployment. Azure Stream Analytics is one such module. You can create an Azure Stream Analytics job from its portal and then go to the Azure IoT Hub portal to deploy it as an IoT Edge module.

Azure Stream Analytics provides a richly structured query syntax for data analysis both in the cloud and on IoT Edge devices. For more information about Azure Stream Analytics on IoT Edge, see Azure Stream Analytics documentation.

This tutorial walks you through creating an Azure Stream Analytics job and deploying it on an IoT Edge device. Doing so lets you process a local telemetry stream directly on the device and generate alerts that drive immediate action on the device.

The tutorial presents two modules:

- A simulated temperature sensor module (tempSensor) that generates temperature data from 20 to 120 degrees, incremented by 1 every 5 seconds.
- A Stream Analytics module that resets the tempSensor when the 30-second average reaches 70. In a production environment, you might use this functionality to shut off a machine or take preventative measures when the temperature reaches dangerous levels.

In this tutorial, you learn how to:

- Create an Azure Stream Analytics job to process data on the edge.
- Connect the new Azure Stream Analytics job with other IoT Edge modules.
- Deploy the Azure Stream Analytics job to an IoT Edge device.

## Prerequisites

- An IoT hub.
- The device that you created and configured in the quickstart or in the articles about deploying Azure IoT Edge on a simulated device in Windows or in Linux. You need to know the device connection key and the device ID.
- Docker running on your IoT Edge device.
  - Install Docker on Windows.
  - Install Docker on Linux.
- Python 2.7.x on your IoT Edge device.
  - Install Python 2.7 on Windows.
  - Most Linux distributions, including Ubuntu, already have Python 2.7 installed. To ensure that pip is installed, use the following command: `sudo apt-get install python-pip`.

## Create an Azure Stream Analytics job

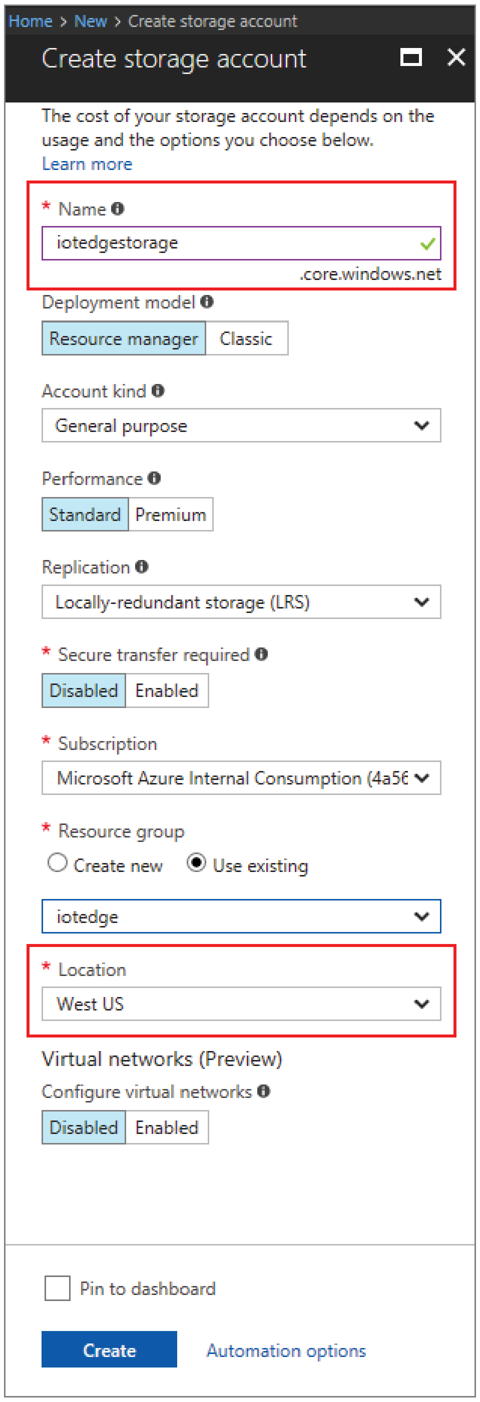
In this section, you create an Azure Stream Analytics job to take data from your IoT hub, query the sent telemetry data from your device, and then forward the results to an Azure Blob storage container. For more information, see the "Overview" section of the Stream Analytics documentation.

### Create a storage account

An Azure Storage account is required to provide an endpoint to be used as an output in your Azure Stream Analytics job. The example in this section uses the Blob storage type. For more information, see the "Blobs" section of the Azure Storage documentation.

1. In the Azure portal, go to **Create a resource**, enter **Storage account** in the search box, and then select **Storage account - blob, file, table, queue**.

2. In the **Create storage account** pane, enter a name for your storage account, select the same location where your IoT hub is stored, and then select **Create**. Note the name for later use.



Home > New > Create storage account

Create storage account

The cost of your storage account depends on the usage and the options you choose below.
Learn more

* Name: iotedgestorage .core.windows.net

Deployment model: [✓ Resource manager] Classic

Account kind: General purpose

Performance: [✓ Standard] Premium

Replication: Locally-redundant storage (LRS)

* Secure transfer required: [✓ Disabled] Enabled

* Subscription: Microsoft Azure Internal Consumption (4a5€

* Resource group: Create new [✓ Use existing]

iotedge

* Location: West US

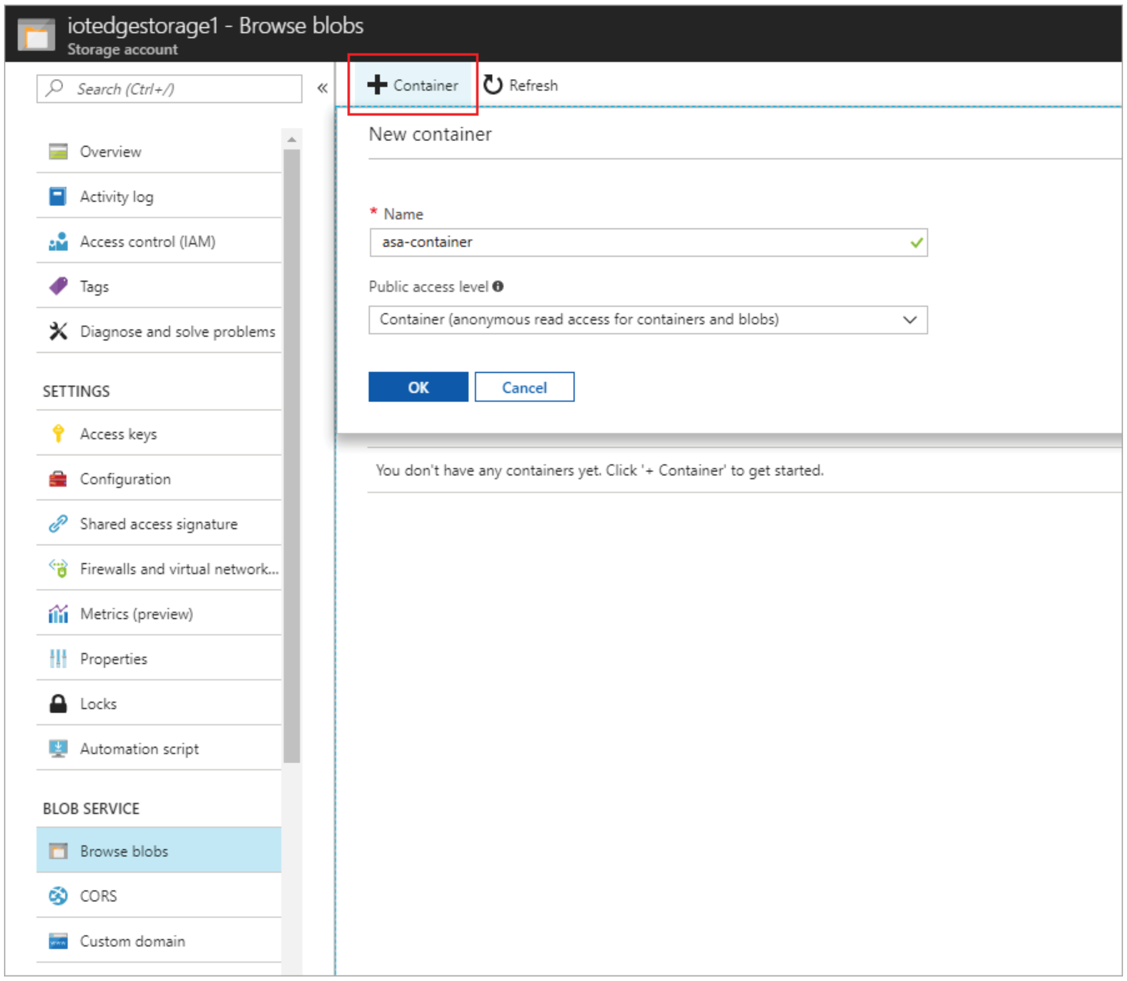
Virtual networks (Preview)

Configure virtual networks: [✓ Disabled] Enabled

Pin to dashboard

3. Go to the storage account that you just created, and then select **Browse blobs**.

4. Create a new container for the Azure Stream Analytics module to store data, set the access level to **Container**, and then select **OK**.



New container

* Name: asa-container

Public access level: Container (anonymous read access for containers and blobs)

You don't have any containers yet. Click '+ Container' to get started.

### Create a Stream Analytics job

1. In the Azure portal, go to **Create a resource** > **Internet of Things**, and then select **Stream Analytics Job**.

2. In the **New Stream Analytics Job** pane, do the following:

   a. In the **Job name** box, type a job name.

   b. Under **Hosting environment**, select **Edge**.

   c. In the remaining fields, use the default values.

   **NOTE**

   Currently, Azure Stream Analytics jobs on IoT Edge aren't supported in the West US 2 region.

3. Select **Create**.

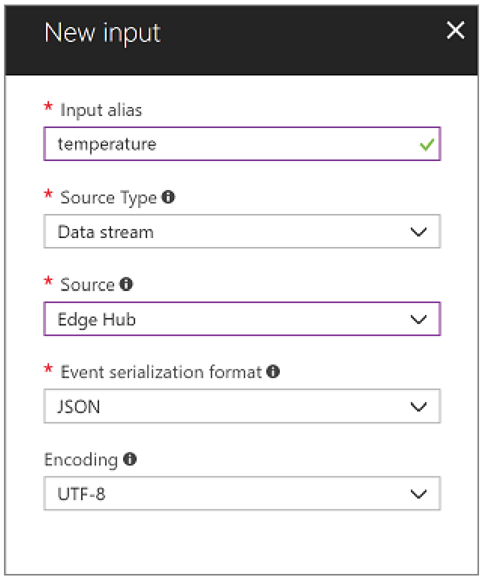
4. In the created job, under **Job Topology**, select **Inputs**, and then select **Add**.

5. In the **New input** pane, do the following:

   a. In the **Input alias** box, enter **temperature**.

   b. In the **Source Type** box, select **Data stream**.

c. In the remaining fields, use the default values.


New input

* Input alias
temperature

* Source Type
Data stream

* Source
Edge Hub

* Event serialization format
JSON

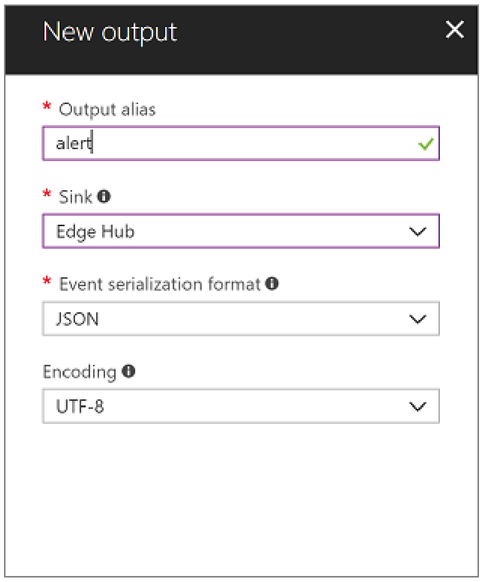
Encoding
UTF-8

6. Select **Create**.
7. Under **Job Topology**, select **Outputs**, and then select **Add**.
8. In the **New output** pane, do the following:

a. In the **Output alias** box, type **alert**.

b. In the remaining fields, use the default values.

c. Select **Create**.


New output

* Output alias
alert

* Sink
Edge Hub

* Event serialization format
JSON

Encoding
UTF-8

9. Under **Job Topology**, select **Query**, and then replace the default text with the following query:

```
SELECT
    'reset' AS command
INTO
   alert
FROM
   temperature TIMESTAMP BY timeCreated
GROUP BY TumblingWindow(second,30)
HAVING Avg(machine.temperature) > 70
```

10. Select **Save**.

## Deploy the job

You are now ready to deploy the Azure Stream Analytics job on your IoT Edge device.

1. In the Azure portal, in your IoT hub, go to **IoT Edge (preview)**, and then open the details page for your IoT Edge device.

2. Select **Set modules**.
   If you previously deployed the tempSensor module on this device, it might autopopulate. If it does not, add the module by doing the following:

   a. Select **Add IoT Edge Module**.

   b. For the name, type **tempSensor**.

   c. For the image URI, enter **microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview**.

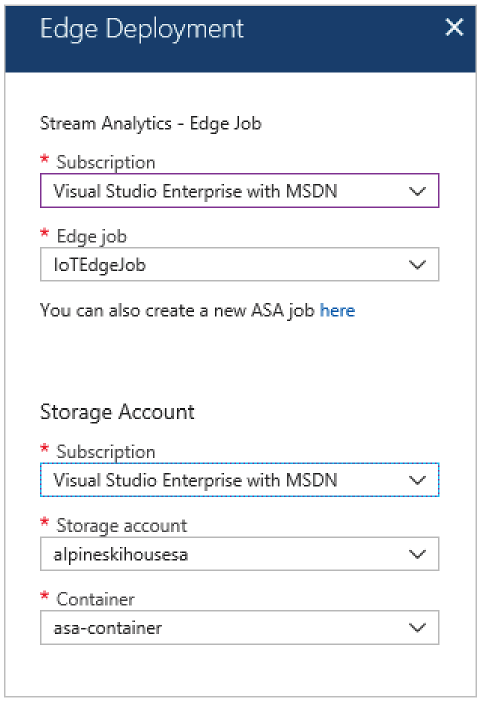
   d. Leave the other settings unchanged.

   e. Select **Save**.

3. To add your Azure Stream Analytics Edge job, select **Import Azure Stream Analytics IoT Edge Module**.

4. Select your subscription and the Azure Stream Analytics Edge job that you created.

5. Select your subscription and the storage account that you created, and then select **Save**.



Edge Deployment

Stream Analytics - Edge Job

* Subscription: Visual Studio Enterprise with MSDN

* Edge job: IoTEdgeJob

You can also create a new ASA job here

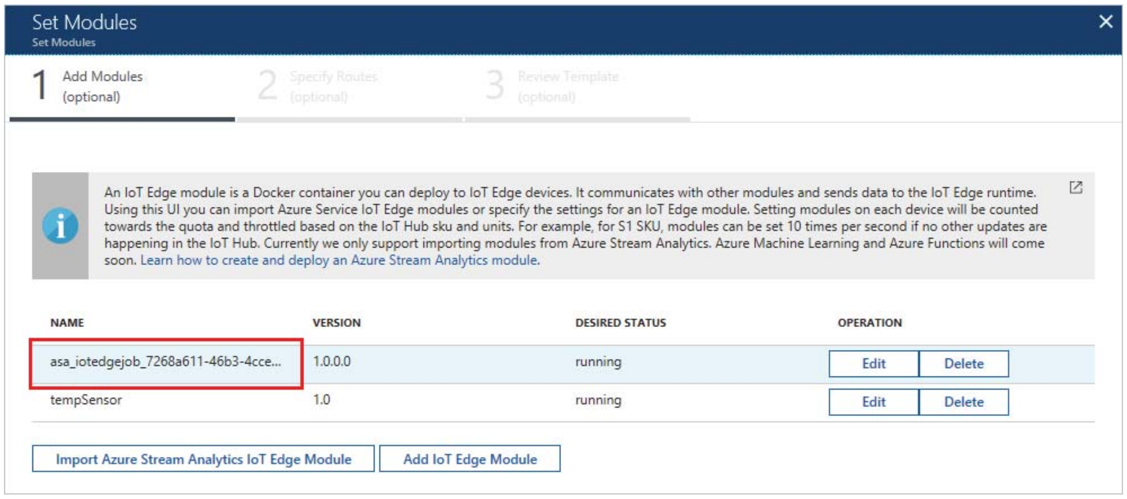
Storage Account

* Subscription: Visual Studio Enterprise with MSDN

* Storage account: alpineskihousesa

* Container: asa-container

6. Copy the name of your Azure Stream Analytics module.



An IoT Edge module is a Docker container you can deploy to IoT Edge devices. It communicates with other modules and sends data to the IoT Edge runtime. Using this UI you can import Azure Service IoT Edge modules or specify the settings for an IoT Edge module. Setting modules on each device will be counted towards the quota and throttled based on the IoT Hub sku and units. For example, for S1 SKU, modules can be set 10 times per second if no other updates are happening in the IoT Hub. Currently we only support importing modules from Azure Stream Analytics. Azure Machine Learning and Azure Functions will come soon. Learn how to create and deploy an Azure Stream Analytics module.

| NAME | VERSION | DESIRED STATUS | OPERATION |
|---|---|---|---|
| asa_iotedgejob_7268a611-46b3-4cce... | 1.0.0.0 | running | Edit Delete |
| tempSensor | 1.0 | running | Edit Delete |

7. To configure routes, select **Next**.

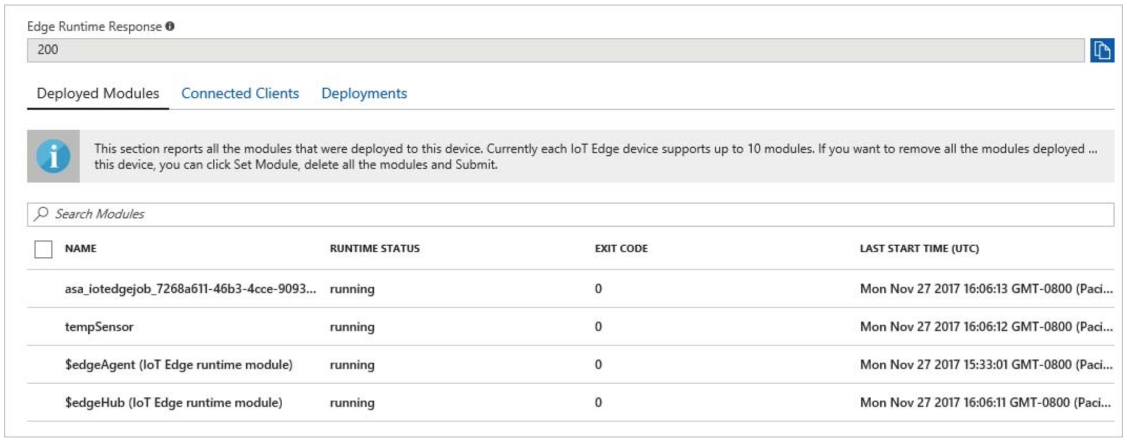
8. Copy the following code to **Routes**. Replace *{moduleName}* with the module name that you copied:

```
{
    "routes": {
      "telemetryToCloud": "FROM /messages/modules/tempSensor/* INTO $upstream",
      "alertsToCloud": "FROM /messages/modules/{moduleName}/* INTO $upstream",
      "alertsToReset": "FROM /messages/modules/{moduleName}/* INTO
BrokeredEndpoint(\"/modules/tempSensor/inputs/control\")",
      "telemetryToAsa": "FROM /messages/modules/tempSensor/* INTO
BrokeredEndpoint(\"/modules/{moduleName}/inputs/temperature\")"
    }
}
```

9. Select **Next**.

10. In the **Review Template** step, select **Submit**.

11. Return to the device details page, and then select **Refresh**.
    You should see the new Stream Analytics module running, along with the IoT Edge agent module and the IoT Edge hub.



This section reports all the modules that were deployed to this device. Currently each IoT Edge device supports up to 10 modules. If you want to remove all the modules deployed ... this device, you can click Set Module, delete all the modules and Submit.

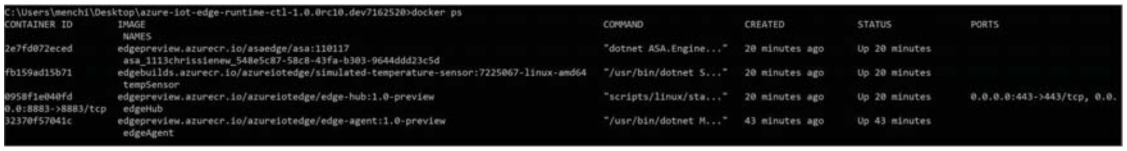
| NAME | RUNTIME STATUS | EXIT CODE | LAST START TIME (UTC) |
|---|---|---|---|
| asa_iotedgejob_7268a611-46b3-4cce-9093... | running | 0 | Mon Nov 27 2017 16:06:13 GMT-0800 (Paci... |
| tempSensor | running | 0 | Mon Nov 27 2017 16:06:12 GMT-0800 (Paci... |
| $edgeAgent (IoT Edge runtime module) | running | 0 | Mon Nov 27 2017 15:33:01 GMT-0800 (Paci... |
| $edgeHub (IoT Edge runtime module) | running | 0 | Mon Nov 27 2017 16:06:11 GMT-0800 (Paci... |

## View data

Now you can go to your IoT Edge device to check out the interaction between the Azure Stream Analytics module and the tempSensor module.

1. Check that all the modules are running in Docker:

```
docker ps
```

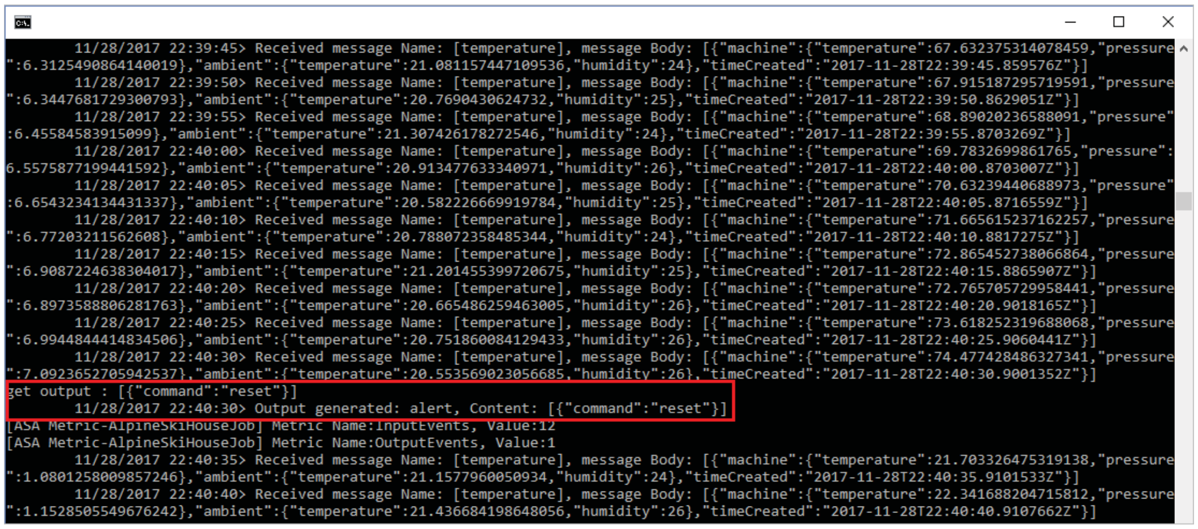


```
C:\Users\menchi\Desktop\azure-iot-edge-runtime-ctl-1.0.0rc10.dev7162520>docker ps
CONTAINER ID        IMAGE                                                                     COMMAND                  CREATED             STATUS              PORTS
                     NAMES
2e7fd072eced        edgepreview.azurecr.io/asaedge/asa:110117                                 "dotnet ASA.Engine..."   20 minutes ago      Up 20 minutes
                     asa_1113chrissienew_548e5c87-58c8-43fa-b303-9644ddd23c5d
fb159ad15b71        edgebuilds.azurecr.io/azureiotedge/simulated-temperature-sensor:7225067-linux-amd64   "/usr/bin/dotnet S..."   20 minutes ago      Up 20 minutes
                     tempSensor
0958f1e040fd        edgepreview.azurecr.io/azureiotedge/edge-hub:1.0-preview                  "scripts/linux/sta..."   20 minutes ago      Up 20 minutes       0.0.0.0:443->443/tcp, 0.0.
0.0:8883->8883/tcp   edgeHub
32370f57041c        edgepreview.azurecr.io/azureiotedge/edge-agent:1.0-preview                "/usr/bin/dotnet M..."   43 minutes ago      Up 43 minutes
                     edgeAgent
```

2. View all system logs and metrics data. Use the Stream Analytics module name:

```
docker logs -f {moduleName}
```

You should be able to watch the machine's temperature gradually rise until it reaches 70 degrees for 30 seconds. Then the Stream Analytics module triggers a reset, and the machine temperature drops back to 21.



```
        11/28/2017 22:39:45> Received message Name: [temperature], message Body: [{"machine":{"temperature":67.632375314078459,"pressure
":6.3125490864140019},"ambient":{"temperature":21.081157447109536,"humidity":24},"timeCreated":"2017-11-28T22:39:45.8595765Z"}]
        11/28/2017 22:39:50> Received message Name: [temperature], message Body: [{"machine":{"temperature":67.915187295719591,"pressure
":6.3447681729300793},"ambient":{"temperature":20.7690430624732,"humidity":25},"timeCreated":"2017-11-28T22:39:50.8629051Z"}]
        11/28/2017 22:39:55> Received message Name: [temperature], message Body: [{"machine":{"temperature":68.89020236588091,"pressure"
:6.45584583915099},"ambient":{"temperature":21.307426178272546,"humidity":24},"timeCreated":"2017-11-28T22:39:55.8703269Z"}]
        11/28/2017 22:40:00> Received message Name: [temperature], message Body: [{"machine":{"temperature":69.7832699861765,"pressure":
6.5575877199441592},"ambient":{"temperature":20.913477633340971,"humidity":26},"timeCreated":"2017-11-28T22:40:00.8703007Z"}]
        11/28/2017 22:40:05> Received message Name: [temperature], message Body: [{"machine":{"temperature":70.63239440688973,"pressure"
:6.6543234134431337},"ambient":{"temperature":20.582226669919784,"humidity":25},"timeCreated":"2017-11-28T22:40:05.8716559Z"}]
        11/28/2017 22:40:10> Received message Name: [temperature], message Body: [{"machine":{"temperature":71.665615237162257,"pressure
":6.77203211562608},"ambient":{"temperature":20.788072358485344,"humidity":24},"timeCreated":"2017-11-28T22:40:10.8817275Z"}]
        11/28/2017 22:40:15> Received message Name: [temperature], message Body: [{"machine":{"temperature":72.865452738066864,"pressure
":6.9087224638304017},"ambient":{"temperature":21.201455399720675,"humidity":25},"timeCreated":"2017-11-28T22:40:15.8865907Z"}]
        11/28/2017 22:40:20> Received message Name: [temperature], message Body: [{"machine":{"temperature":72.765705729958441,"pressure
":6.8973588806281763},"ambient":{"temperature":20.665486259463005,"humidity":26},"timeCreated":"2017-11-28T22:40:20.9018165Z"}]
        11/28/2017 22:40:25> Received message Name: [temperature], message Body: [{"machine":{"temperature":73.618252319688068,"pressure
":6.9944844414834506},"ambient":{"temperature":20.751860084129433,"humidity":26},"timeCreated":"2017-11-28T22:40:25.9060441Z"}]
        11/28/2017 22:40:30> Received message Name: [temperature], message Body: [{"machine":{"temperature":74.477428486327341,"pressure
":7.09236527059425371,"ambient":{"temperature":20.553569023056685,"humidity":26},"timeCreated":"2017-11-28T22:40:30.9001352Z"}]
get output : [{"command":"reset"}]
        11/28/2017 22:40:30> Output generated: alert, Content: [{"command":"reset"}]
[ASA Metric-AlpineSkiHouseJob] Metric Name:InputEvents, Value:12
[ASA Metric-AlpineSkiHouseJob] Metric Name:OutputEvents, Value:1
        11/28/2017 22:40:35> Received message Name: [temperature], message Body: [{"machine":{"temperature":21.703326475319138,"pressure
":1.0801258009857246},"ambient":{"temperature":21.1577960050934,"humidity":24},"timeCreated":"2017-11-28T22:40:35.9101533Z"}]
        11/28/2017 22:40:40> Received message Name: [temperature], message Body: [{"machine":{"temperature":22.341688204715812,"pressure
":1.1528505549676242},"ambient":{"temperature":21.436684198648056,"humidity":26},"timeCreated":"2017-11-28T22:40:40.9107662Z"}]
```

## Next steps

In this tutorial, you configured an Azure storage container and a Streaming Analytics job to analyze data from your IoT Edge device. You then loaded a custom Azure Stream Analytics module to move data from your device, through the stream, into a blob for download. To see how Azure IoT Edge can create more solutions for your business, continue on to the other tutorials.

Deploy an Azure Machine Learning model as a module

# Deploy Azure Machine Learning as an IoT Edge module - preview

5/7/2018 • 4 min to read • Edit Online

You can use IoT Edge modules to deploy code that implements your business logic directly to your IoT Edge devices. This tutorial walks you through deploying an Azure Machine Learning module that predicts when a device fails based on sensor data on the simulated IoT Edge device that you created in the Deploy Azure IoT Edge on a simulated device on Windows or Linux tutorials.

In this tutorial, you learn how to:

- Create an Azure Machine Learning module
- Push a module container to an Azure container registry
- Deploy an Azure Machine Learning module to your IoT Edge device
- View generated data

The Azure Machine Learning module that you create in this tutorial reads the environmental data generated by your device and labels the messages as anomalous or not.

## Prerequisites

- The Azure IoT Edge device that you created in the quickstart or first tutorial.
- The IoT Hub connection string for the IoT hub that your IoT Edge device connects to.
- An Azure Machine Learning account. To create an account, follow the instructions in Create Azure Machine Learning accounts and install Azure Machine Learning Workbench. You do not need to install the workbench application for this tutorial.
- Module Management for Azure ML on your machine. To set up your environment and create an account, follow the instructions in Model management setup.

The Azure Machine Learning module does not support ARM processors.

## Create the Azure ML container

In this section, you download the trained model files and convert them into an Azure ML container.

On the machine running Module Management for Azure ML, download and save iot_score.py and model.pkl from the Azure ML IoT Toolkit on GitHub. These files define the trained machine learning model that you will deploy to your Iot Edge device.

Use the trained model to create a container that can be deployed to IoT Edge devices. Use the following command to:

- Register your model.
- Create a manifest.
- Create a Docker container image named *machinelearningmodule*.
- Deploy the image to your Azure Kubernetes Service (AKS) cluster.

```
az ml service create realtime --model-file model.pkl -f iot_score.py -n machinelearningmodule -r python
```

**View the container repository**

Check that your container image was successfully created and stored in the Azure container repository that is associated with your machine learning environment.

1. On the Azure portal, go to **All Services** and Select **Container registries**.
2. Select your registry. The name should start with **mlcr** and it belongs to the resource group, location, and subscription that you used to set up Module Management.
3. Select **Access keys**
4. Copy the **Login server**, **Username**, and **Password**. You need these to access the registry from your Edge devices.
5. Select **Repositories**
6. Select **machinelearningmodule**
7. You now have the full image path of the container. Take note of this image path for the next section. It should look like this: **<registry_name>.azureacr.io/machinelearningmodule:1**

## Add registry credentials to your Edge device

Add the credentials for your registry to the Edge runtime on the computer where you are running your Edge device. This command gives the runtime access to pull the container.

Linux:

```
sudo iotedgectl login --address <registry-login-server> --username <registry-username> --password <registry-
password>
```

Windows:

```
iotedgectl login --address <registry-login-server> --username <registry-username> --password <registry-
password>
```

## Run the solution

1. On the Azure portal, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set modules**.
4. If you've previously deployed the tempSensor module to your IoT Edge device, it may autopopulate. If it's not already in your list of modules, add it.
   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `tempSensor`.
   c. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
   d. Select **Save**.
5. Add the machine learning module that you created.
   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `machinelearningmodule`
   c. In the **Image** field, enter your image address; for example `<registry_name>.azurecr.io/machinelearningmodule:1`.
   d. Select **Save**.
6. Back in the **Add Modules** step, select **Next**.
7. In the **Specify Routes** step, copy the JSON below into the text box. The first route transports messages from the temperature sensor to the machine learning module via the "amlInput" endpoint, which is the

endpoint that all Azure Machine Learning modules use. The second route transports messages from the machine learning module to IoT Hub. In this route, ''amlOutput'' is the endpoint that all Azure Machine Learning modules use to output data, and ''$upstream'' denotes IoT Hub.

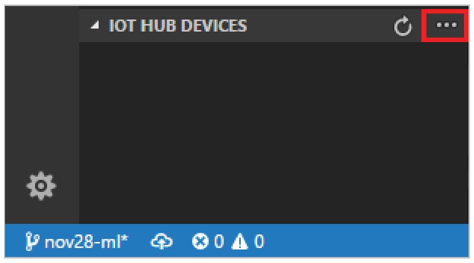
```
{
    "routes": {
        "sensorToMachineLearning":"FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/machinelearningmodule/inputs/amlInput\")",
        "machineLearningToIoTHub": "FROM /messages/modules/machinelearningmodule/outputs/amlOutput INTO
$upstream"
    }
}
```

8. Select **Next**.
9. In the **Review Template** step, select **Submit**.
10. Return to the device details page and select **Refresh**. You should see the new **machinelearningmodule** running along with the **tempSensor** module and the IoT Edge runtime modules.

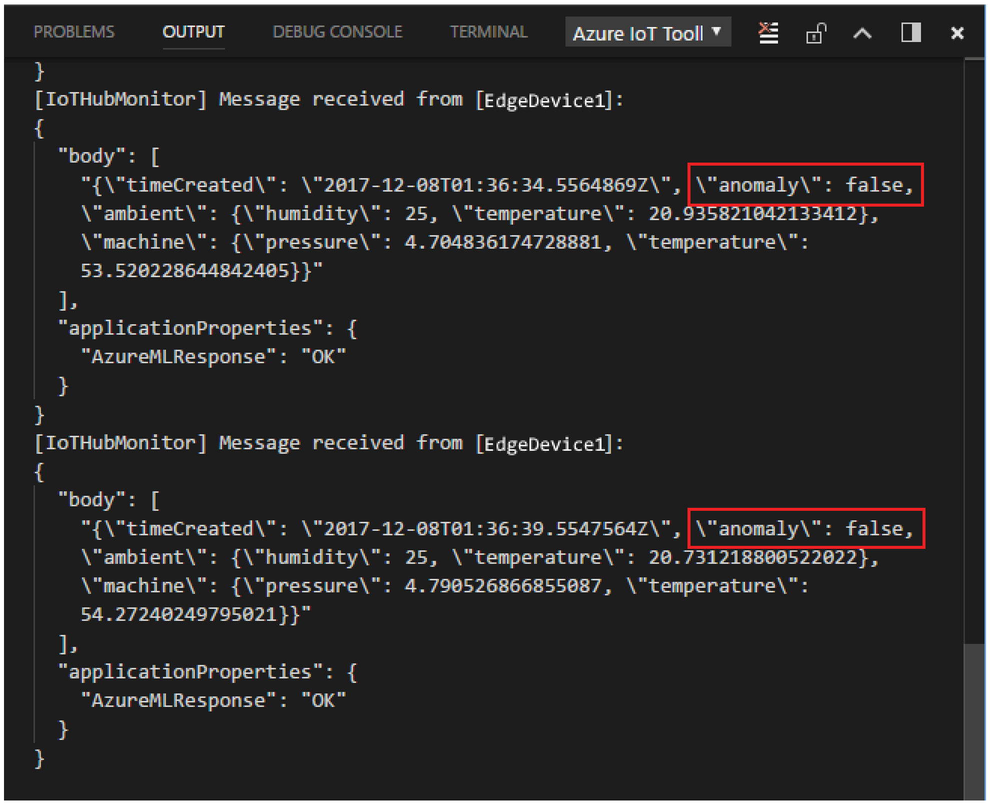
## View generated data

You can view the device-to-cloud messages that your IoT Edge device sends by using the IoT Hub explorer or the Azure IoT Toolkit extension for Visual Studio Code.

1. In Visual Studio Code, select **IoT Hub Devices**.
2. Select **...** then select **Set IoT Hub Connection String** from the menu.



3. In the text box that opens at the top of the page, enter the iothubowner connection string for your IoT Hub. Your IoT Edge device should appear in the IoT Hub Devices list.
4. Select **...** again then select **Start monitoring D2C message**.
5. Observe the messages coming from tempSensor every five seconds. The message body contains a property called **anomaly** which the machinelearningmodule provides with a true or false value. The **AzureMLResponse** property contains the value "OK" if the model ran successfully.



```
}
[IoTHubMonitor] Message received from [EdgeDevice1]:
{
  "body": [
    "{\"timeCreated\": \"2017-12-08T01:36:34.5564869Z\", \"anomaly\": false,
    \"ambient\": {\"humidity\": 25, \"temperature\": 20.935821042133412},
    \"machine\": {\"pressure\": 4.704836174728881, \"temperature\":
    53.520228644842405}}"
  ],
  "applicationProperties": {
    "AzureMLResponse": "OK"
  }
}
[IoTHubMonitor] Message received from [EdgeDevice1]:
{
  "body": [
    "{\"timeCreated\": \"2017-12-08T01:36:39.5547564Z\", \"anomaly\": false,
    \"ambient\": {\"humidity\": 25, \"temperature\": 20.731218800522022},
    \"machine\": {\"pressure\": 4.790526866855087, \"temperature\":
    54.27240249795021}}"
  ],
  "applicationProperties": {
    "AzureMLResponse": "OK"
  }
}
```

## Next steps

In this tutorial, you deployed an IoT Edge module powered by Azure Machine Learning. You can continue on to any of the other tutorials to learn about other ways that Azure IoT Edge can help you turn data into business insights at the edge.

Deploy an Azure Function as a module

# Deploy Azure Function as an IoT Edge module - preview

4/9/2018 • 7 min to read • Edit Online

You can use Azure Functions to deploy code that implements your business logic directly to your IoT Edge devices. This tutorial walks you through creating and deploying an Azure Function that filters sensor data on the simulated IoT Edge device that you created in the Deploy Azure IoT Edge on a simulated device on Windows or Linux tutorials. In this tutorial, you learn how to:

- Use Visual Studio Code to create an Azure Function
- Use VS Code and Docker to create a Docker image and publish it to your registry
- Deploy the module to your IoT Edge device
- View generated data

The Azure Function that you create in this tutorial filters the temperature data generated by your device and only sends messages upstream to Azure IoT Hub when the temperature is above a specified threshold.

## Prerequisites

- The Azure IoT Edge device that you created in the quickstart or previous tutorial.
- Visual Studio Code.
- C# for Visual Studio Code (powered by OmniSharp) extension.
- Azure IoT Edge extension for Visual Studio Code.
- Docker. The Community Edition (CE) for your platform is sufficient for this tutorial.
- .NET Core 2.0 SDK.

## Create a container registry

In this tutorial, you use the Azure IoT Edge extension for VS Code to build a module and create a **container image** from the files. Then you push this image to a **registry** that stores and manages your images. Finally, you deploy your image from your registry to run on your IoT Edge device.

You can use any Docker-compatible registry for this tutorial. Two popular Docker registry services available in the cloud are Azure Container Registry and Docker Hub. This tutorial uses Azure Container Registry.

1. In the Azure portal, select **Create a resource** > **Containers** > **Azure Container Registry**.
2. Give your registry a name, choose a subscription, choose a resource group, and set the SKU to **Basic**.
3. Select **Create**.
4. Once your container registry is created, navigate to it and select **Access keys**.
5. Toggle **Admin user** to **Enable**.
6. Copy the values for **Login server**, **Username**, and **Password**. You'll use these values later in the tutorial.

## Create a function project

The following steps show you how to create an IoT Edge function using Visual Studio Code and the Azure IoT Edge extension.

1. Open Visual Studio Code.
2. To open the VS Code integrated terminal, select **View** > **Integrated Terminal**.

3. To install (or update) the **AzureIoTEdgeFunction** template in dotnet, run the following command in the integrated terminal:

   ```
   dotnet new -i Microsoft.Azure.IoT.Edge.Function
   ```

4. Create a project for the new module. The following command creates the project folder, **FilterFunction**, with your container repository. The second parameter should be in the form of `<your container registry name>.azurecr.io` if you are using Azure container registry. Enter the following command in the current working folder:

   ```
   dotnet new aziotedgefunction -n FilterFunction -r <your container registry address>/filterfunction
   ```

5. Select **File** > **Open Folder**, then browse to the **FilterFunction** folder and open the project in VS Code.

6. In VS Code explorer, expand the **EdgeHubTrigger-Csharp** folder, then open the **run.csx** file.
7. Replace the contents of the file with the following code:

```
#r "Microsoft.Azure.Devices.Client"
#r "Newtonsoft.Json"

using System.IO;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;

// Filter messages based on the temperature value in the body of the message and the temperature
threshold value.
public static async Task Run(Message messageReceived, IAsyncCollector<Message> output, TraceWriter
log)
{
     const int temperatureThreshold = 25;
     byte[] messageBytes = messageReceived.GetBytes();
     var messageString = System.Text.Encoding.UTF8.GetString(messageBytes);

     if (!string.IsNullOrEmpty(messageString))
     {
         // Get the body of the message and deserialize it
         var messageBody = JsonConvert.DeserializeObject<MessageBody>(messageString);

         if (messageBody != null && messageBody.machine.temperature > temperatureThreshold)
         {
             // Send the message to the output as the temperature value is greater than the threashold
             var filteredMessage = new Message(messageBytes);
             // Copy the properties of the original message into the new Message object
             foreach (KeyValuePair<string, string> prop in messageReceived.Properties)
             {
                 filteredMessage.Properties.Add(prop.Key, prop.Value);                }
             // Add a new property to the message to indicate it is an alert
             filteredMessage.Properties.Add("MessageType", "Alert");
             // Send the message
             await output.AddAsync(filteredMessage);
             log.Info("Received and transferred a message with temperature above the threshold");
         }
     }
 }

 //Define the expected schema for the body of incoming messages
 class MessageBody
 {
     public Machine machine {get;set;}
     public Ambient ambient {get; set;}
     public string timeCreated {get; set;}
 }
 class Machine
 {
    public double temperature {get; set;}
    public double pressure {get; set;}
 }
 class Ambient
 {
    public double temperature {get; set;}
    public int humidity {get; set;}
 }
```

8. Save the file.

## Create a Docker image and publish it to your registry

1. Sign in to Docker by entering the following command in the VS Code integrated terminal:

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

To find the user name, password and login server to use in this command, go to the Azure portal. From **All resources**, click the tile for your Azure container registry to open its properties, then click **Access keys**. Copy the values in the **Username**, **password**, and **Login server** fields.

2. Open **module.json**. Optionally, you can update the `"version"` to eg. **"1.0"**. Also the name of the repository is shown which you entered in the `-r` parameter of `dotnet new aziotedgefunction`.

3. Save the **module.json** file.

4. In VS Code explorer, Right-click the **module.json** file and click **Build and Push IoT Edge module Docker image**. In the pop-up dropdown box at the top of the VS Code window, select your container platform, either **amd64** for Linux container or **windows-amd64** for Windows container. VS Code then containerizes your function codes and push it to the container registry you specified.

5. You can get the full container image address with tag in the VS Code integrated terminal. For more infomation about the build and push definition, you can refer to the `module.json` file.

## Add registry credentials to your Edge device

Add the credentials for your registry to the Edge runtime on the computer where you are running your Edge device. This gives the runtime access to pull the container.

- For Windows, run the following command:

```
iotedgectl login --address <your container registry address> --username <username> --password
<password>
```

- For Linux, run the following command:

```
sudo iotedgectl login --address <your container registry address> --username <username> --password
<password>
```

## Run the solution

1. In the **Azure portal**, navigate to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set Modules**.
4. If you've already deployed the **tempSensor** module to this device, it may be automatically populated. If not, follow these steps to add it:
   a. Select **Add IoT Edge Module**.
   b. In the **Name** field, enter `tempSensor`.
   c. In the **Image URI** field, enter `microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview`.
   d. Leave the other settings unchanged and click **Save**.
5. Add the **filterFunction** module.
   a. Select **Add IoT Edge Module** again.
   b. In the **Name** field, enter `filterFunction`.
   c. In the **Image URI** field, enter your image address; for example `<your container registry address>/filterfunction:0.0.1-amd64`. The full image address can be found from the previous section.
   d. Click **Save**.
6. Click **Next**.

7. In the **Specify Routes** step, copy the JSON below into the text box. The first route transports messages from the temperature sensor to the filter module via the "input1" endpoint. The second route transports messages from the filter module to IoT Hub. In this route, `$upstream` is a special destination that tells Edge Hub to send messages to IoT Hub.

```
{
   "routes":{
      "sensorToFilter":"FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/filterFunction/inputs/input1\")",
      "filterToIoTHub":"FROM /messages/modules/filterFunction/outputs/* INTO $upstream"
   }
}
```

8. Click **Next**.
9. In the **Review Template** step, click **Submit**.
10. Return to the IoT Edge device details page and click **Refresh**. You should see the new **filterfunction** module running along with the **tempSensor** module and the **IoT Edge runtime**.

## View generated data

To monitor device to cloud messages sent from your IoT Edge device to your IoT hub:

1. Configure the Azure IoT Toolkit extension with connection string for your IoT hub:
   a. In the Azure portal, navigate to your IoT hub and select **Shared access policies**.
   b. Select **iothubowner** then copy the value of **Connection string-primary key**.
   c. In the VS Code explorer, click **IOT HUB DEVICES** and then click **...**.
   d. Select **Set IoT Hub Connection String** and enter the Iot Hub connection string in the pop-up window.
2. To monitor data arriving at the IoT hub, select the **View** > **Command Palette...** and search for **IoT: Start monitoring D2C message**.
3. To stop monitoring data, use the **IoT: Stop monitoring D2C message** command in the Command Palette.

## Next steps

In this tutorial, you created an Azure Function that contains code to filter raw data generated by your IoT Edge device. To keep exploring Azure IoT Edge, learn how to use an IoT Edge device as a gateway.

Create an IoT Edge gateway device

# Create an IoT Edge device that acts as a transparent gateway - preview

4/24/2018 • 5 min to read • Edit Online

This article provides detailed instructions for using an IoT Edge device as a transparent gateway. For the rest of this article, the term *IoT Edge gateway* refers to an IoT Edge device used as a transparent gateway. For more detailed information, see How an IoT Edge device can be used as a gateway, which gives a conceptual overview.

**NOTE**

Currently:

- If the gateway is disconnected from IoT Hub, downstream devices cannot authenticate with the gateway.
- IoT Edge devices cannot connect to IoT Edge gateways.

## Understand the Azure IoT device SDK

The Edge hub that is installed in all IoT Edge devices exposes the following primitives to downstream devices:

- device-to-cloud and cloud-to-device messages
- direct methods
- device twin operations

Currently, downstream devices are not able to use file upload when connecting through an IoT Edge gateway.

When you connect devices to an IoT Edge gateway using the Azure IoT device SDK, you need to:

- Set up the downstream device with a connection string referring to the gateway device hostname; and
- Make sure that the downstream device trusts the certificate used to accept the connection by the gateway device.

When you install the Azure IoT Edge runtime using the control script, a certificate is created for the Edge hub, as you did in the tutorial Install IoT Edge on a simulated device on Windows and Linux. This certificate is used by the Edge hub to accept incoming TLS connections, and has to be trusted by the downstream device when connecting to the gateway device.

You can create any certificate infrastructure that enables the trust required for your device-gateway topology. In this article, we assume the same certificate setup that you would use to enable X.509 CA security in IoT Hub, which involves an X.509 CA certificate associated to a specific IoT hub (the *IoT hub owner CA*), and a series of certificates, signed with this CA, installed in the IoT Edge devices.

**IMPORTANT**

Currently, IoT Edge devices and downstream devices can only use SAS tokens to authenticate with IoT Hub. The certificates will be used only to validate the TLS connection between the leaf and gateway device.

Our configuration uses **IoT hub owner CA** as both:

- A signing certificate for the setup of the IoT Edge runtime on all IoT Edge devices; and
- A public key certificate installed in downstream devices.

This results in a solution that enables all devices to use any IoT Edge device as a gateway, as long as they are connected to the same IoT hub.

## Create the certificates for test scenarios

You can use the sample Powershell and Bash scripts described in Managing CA Certificate Sample to generate a self-signed **IoT hub owner CA** and device certificates signed with it.

> **IMPORTANT**
>
> This sample is meant only for test purposes. For production scenarios, refer to Secure your IoT deployment for the Azure IoT guidelines on how to secure your IoT solution, and provision your certificate accordingly.

1. Clone the Microsoft Azure IoT SDKs and libraries for C from GitHub:

   ```
   git clone -b modules-preview https://github.com/Azure/azure-iot-sdk-c.git
   ```

2. To install the certificate scripts, follow the instructions in **Step 1 - Initial Setup** of Managing CA Certificate Sample.

3. To generate the **IoT hub owner CA**, follow the instructions in **Step 2 - Create the certificate chain**. This file is used by the downstream devices to validate the connection.
4. To generate a certificate for your gateway device, use either the Bash or PowerShell instructions:

**Bash**

Create the new device certificate. **DO NOT** name the `myGatewayCAName` to be the same as your gateway host's name. Doing so will cause client certification against these certs to fail.

```
./certGen.sh create_edge_device_certificate myGatewayCAName
```

New files are created: .\certs\new-edge-device.* contains the public key and PFX, and .\private\new-edge-device.key.pem contains the device's private key.

In the `certs` directory, run the following command to get the full chain of the device public key:

```
cd ./certs
cat ./new-edge-device.cert.pem ./azure-iot-test-only.intermediate.cert.pem ./azure-iot-test-
only.root.ca.cert.pem > ./new-edge-device-full-chain.cert.pem
```

**Powershell**

Create the new device certificate:

```
New-CACertsEdgeDevice myGateway
```

New myEdgeDevice* files are created, which contain the public key, private key, and PFX of this certificate.

When prompted to enter a password during the signing process, enter "1234".

## Configure a gateway device

In order to configure your IoT Edge device as a gateway you just need to configure to use the device certificate created in the previous section.

We assume the following file names from the sample scripts above:

| OUTPUT | FILE NAME |
|---|---|
| Device certificate | `certs/new-edge-device.cert.pem` |
| Device private key | `private/new-edge-device.cert.pem` |
| Device certificate chain | `certs/new-edge-device-full-chain.cert.pem` |
| IoT hub owner CA | `certs/azure-iot-test-only.root.ca.cert.pem` |

Provide the device and certificate information to the IoT Edge runtime.

In Linux, using the Bash output:

```
sudo iotedgectl setup --connection-string {device connection string} \
     --edge-hostname {gateway hostname, e.g. mygateway.contoso.com} \
     --device-ca-cert-file {full path}/certs/new-edge-device.cert.pem \
     --device-ca-chain-cert-file {full path}/certs/new-edge-device-full-chain.cert.pem \
     --device-ca-private-key-file {full path}/private/new-edge-device.key.pem \
     --owner-ca-cert-file {full path}/certs/azure-iot-test-only.root.ca.cert.pem
```

In Windows, using the PowerShell output:

```
iotedgectl setup --connection-string {device connection string}
     --edge-hostname {gateway hostname, e.g. mygateway.contoso.com}
     --device-ca-cert-file {full path}/certs/new-edge-device.cert.pem
     --device-ca-chain-cert-file {full path}/certs/new-edge-device-full-chain.cert.pem
     --device-ca-private-key-file {full path}/private/new-edge-device.key.pem
     --owner-ca-cert-file {full path}/RootCA.pem
```

By default the sample scripts do not set a passphrase to the device private key. If you set a passphrase, add the following parameter: `--device-ca-passphrase {passphrase}`.

The script prompts you to set a passphrase for the Edge Agent certificate. Restart the IoT Edge runtime after this command:

```
iotedgectl restart
```

## Configure a downstream device

A downstream device can be any application using the Azure IoT device SDK, such as the simple one described in Connect your device to your IoT hub using .NET.

First, a downstream device application has to trust the **IoT hub owner CA** certificate in order to validate the TLS connections to the gateway devices. This step can usually be performed in two ways: at the OS level, or (for certain languages) at the application level.

For instance, for .NET applications, you can add the following snippet to trust a certificate in PEM format stored in path `certPath`. Depending on which version of the script you used, the path references either `certs/azure-iot-test-only.root.ca.cert.pem` (Bash) or `RootCA.pem` (Powershell).

```
using System.Security.Cryptography.X509Certificates;

...

X509Store store = new X509Store(StoreName.Root, StoreLocation.CurrentUser);
store.Open(OpenFlags.ReadWrite);
store.Add(new X509Certificate2(X509Certificate2.CreateFromCertFile(certPath)));
store.Close();
```

Performing this step at the OS level is different between Windows and across Linux distributions.

The second step is to initialize the IoT Hub device sdk with a connection string referring to the hostname of the gateway device. This is done by appending the `GatewayHostName` property to your device connection string. For instance, here is a sample device connection string for a device, to which we appended the `GatewayHostName` property:

```
HostName=yourHub.azure-devices-
int.net;DeviceId=yourDevice;SharedAccessKey=2BUaYca45uBS/O1AsawsuQslH4GX+SPkrytydWNdFxc=;GatewayHostName=mygat
eway.contoso.com
```

These two steps enable your device application to connect to the gateway device.

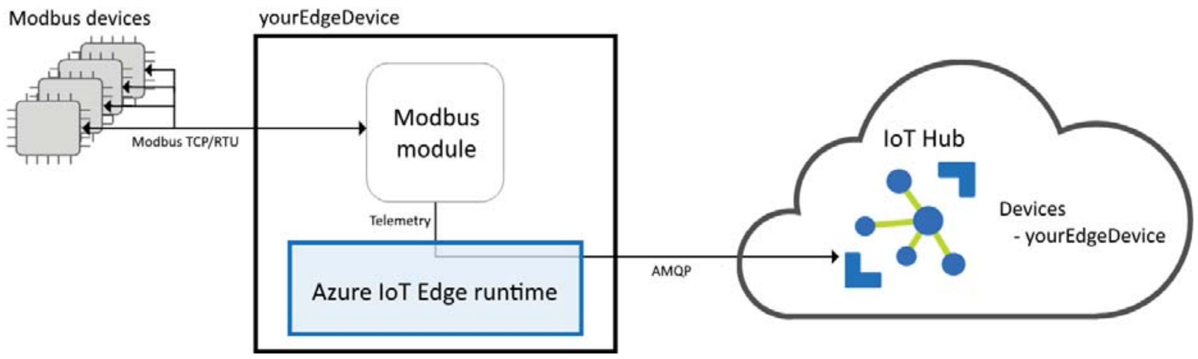
## Next steps

Understand the requirements and tools for developing IoT Edge modules.

# Connect Modbus TCP devices through an IoT Edge device gateway - preview

1/4/2018 • 2 min to read • Edit Online

If you want to connect IoT devices that use Modbus TCP or RTU protocols to an Azure IoT hub, use an IoT Edge device as a gateway. The gateway device reads data from your Modbus devices, then communicates that data to the cloud using a supported protocol.



This article covers how to create your own container image for a Modbus module (or you can use a prebuilt sample) and then deploy it to the IoT Edge device that will act as your gateway.

This article assumes that you're using Modbus TCP protocol. For more information about how to configure the module to support Modbus RTU, refer to the Azure IoT Edge Modbus module project on Github.

## Prerequisites

- An Azure IoT Edge device. For a walkthrough on how to set up one, see Deploy Azure IoT Edge on a simulated device in Windows or Linux.
- The primary key connection string for the IoT Edge device.
- A physical or simulated Modbus device that supports Modbus TCP.

## Prepare a Modbus container

If you want to test the Modbus gateway functionality, Microsoft has a sample module that you can use. To use the sample module, go to the Run the solution section and enter the following as the Image URI:

```
microsoft/azureiotedge-modbus-tcp:1.0-preview
```

If you want to create your own module and customize it for your environment, there is an open source Azure IoT Edge Modbus module project on Github. Follow the guidance in that project to create your own container image. If you create your own container image, refer to Develop and deploy a C# IoT Edge module for instructions on publishing container images to a registry, and deploying a custom module to your device.

## Run the solution

1. On the Azure portal, go to your IoT hub.
2. Go to **IoT Edge (preview)** and select your IoT Edge device.
3. Select **Set modules**.

4. Add the Modbus module:

   a. Select **Add IoT Edge module**.
   b. In the **Name** field, enter "modbus".
   c. In the **Image** field, enter the image URI of the sample container: `microsoft/azureiotedge-modbus-tcp:1.0-preview`.
   d. Check the **Enable** box to update the module twin's desired properties.
   e. Copy the following JSON into the text box. Change the value of **SlaveConnection** to the IPv4 address of your Modbus device.

   ```
   {
     "properties.desired":{
       "PublishInterval":"2000",
       "SlaveConfigs":{
         "Slave01":{
           "SlaveConnection":"<IPV4 address>",
           "HwId":"PowerMeter-0a:01:01:01:01:01",
           "Operations":{
             "Op01":{
               "PollingInterval": "1000",
               "UnitId":"1",
               "StartAddress":"400001",
               "Count":"2",
               "DisplayName":"Voltage"
             }
           }
         }
       }
     }
   }
   ```

   f. Select **Save**.

5. Back in the **Add Modules** step, select **Next**.
6. In the **Specify Routes** step, copy the following JSON into the text box. This route sends all messages collected by the Modbus module to IoT Hub. In this route, ''modbusOutput'' is the endpoint that Modbus module use to output data, and ''upstream'' is a special destination that tells Edge Hub to send messages to IoT Hub.

   ```
   {
    "routes": {
      "modbusToIoTHub":"FROM /messages/modules/modbus/outputs/modbusOutput INTO $upstream"
    }
   }
   ```

7. Select **Next**.
8. In the **Review Template** step, select **Submit**.
9. Return to the device details page and select **Refresh**. You should see the new **modbus** running along with the IoT Edge runtime.

## View data

View the data coming through the modbus module:

```
docker logs -f modbus
```

You can also view the telemetry the device is sending by using the IoT Hub explorer tool.

## Next steps

- To learn more about how IoT Edge devices can act as gateways, see Create an IoT Edge device that acts as a transparent gateway
- For more information about how IoT Edge modules work, see Understand Azure IoT Edge modules

# Deploy modules to an IoT Edge device using IoT extension for Azure CLI 2.0

4/25/2018 • 3 min to read • Edit Online

Azure CLI 2.0 is an open-source cross platform command-line tool for managing Azure resources such as IoT Edge. Azure CLI 2.0 is available on Windows, Linux, and MacOS.

Azure CLI 2.0 enables you to manage Azure IoT Hub resources, device provisioning service instances, and linked-hubs out of the box. The new IoT extension enriches Azure CLI 2.0 with features such as device management and full IoT Edge capability.

In this article, you set up Azure CLI 2.0 and the IoT extension. Then you learn how to deploy modules to an IoT Edge device using the available CLI commands.

## Prerequisites

- An Azure account. If you don't have one yet, you can create a free account today.
- Python 2.7x or Python 3.x.
- Azure CLI 2.0 in your environment. At a minimum, your Azure CLI 2.0 version must be 2.0.24 or above. Use `az --version` to validate. This version supports az extension commands and introduces the Knack command framework. One simple way to install on Windows is to download and install the MSI.
- The IoT extension for Azure CLI 2.0:
  1. Run `az extension add --name azure-cli-iot-ext`.
  2. After installation, use `az extension list` to validate the currently installed extensions or `az extension show --name azure-cli-iot-ext` to see details about the IoT extension.
  3. To remove the extension, use `az extension remove --name azure-cli-iot-ext`.

## Create an IoT Edge device

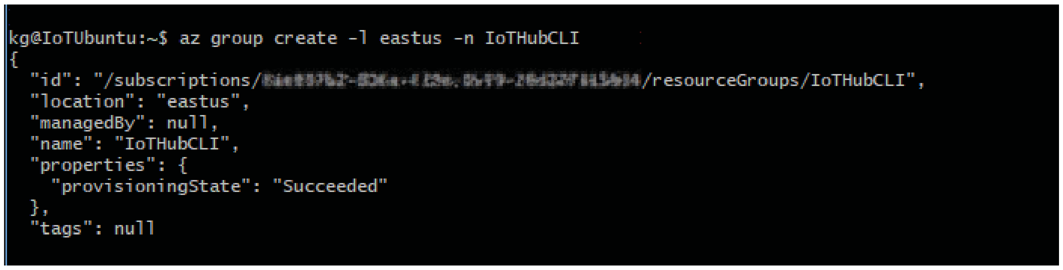
This article gives instructions to create an IoT Edge deployment. The example shows you how to sign in to your Azure account, create an Azure Resource Group (a container that holds related resources for an Azure solution), create an IoT Hub, create three IoT Edge devices identity, set tags and then create an IoT Edge deployment that targets those devices.

Log in to your Azure account. After you enter the following login command, you're prompted to use a web browser to sign in using a one-time code:

```
az login
```

Create a new resource group called **IoTHubCLI** in the East US region:

```
az group create -l eastus -n IoTHubCLI
```



```
kg@IoTUbuntu:~$ az group create -l eastus -n IoTHubCLI
{
  "id": "/subscriptions/[redacted]/resourceGroups/IoTHubCLI",
  "location": "eastus",
  "managedBy": null,
  "name": "IoTHubCLI",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
```

Create an IoT hub called **CLIDemoHub** in the newly created resource group:

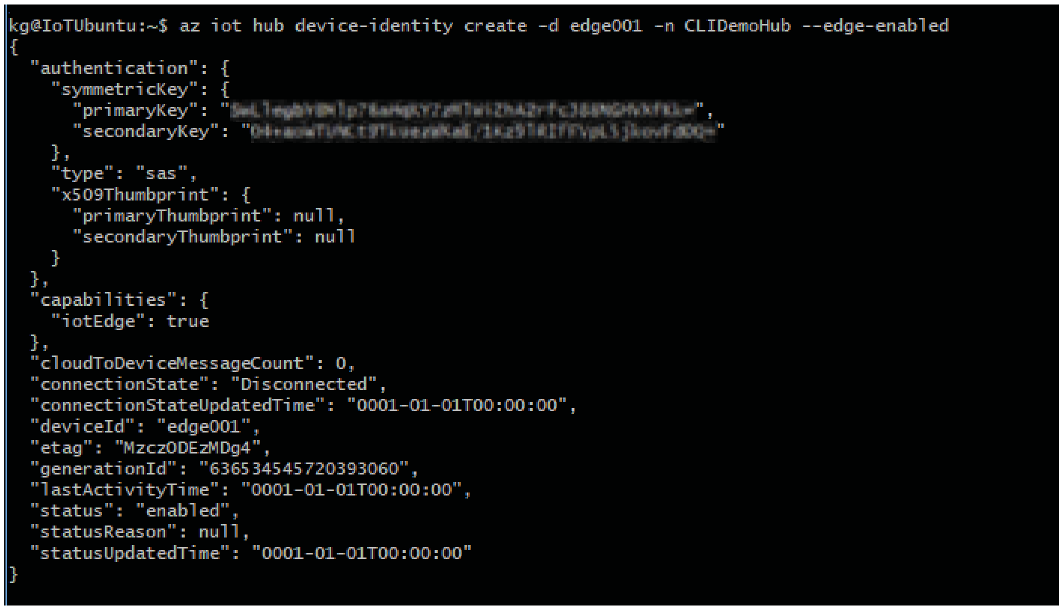
```
az iot hub create --name CLIDemoHub --resource-group IoTHubCLI --sku S1
```

**TIP**

Each subscription is allotted one free IoT hub. To create a free hub with the CLI command, replace the SKU value with `--sku F1`. If you already have a free hub in your subscription, you'll get an error message when you try to create a second one.

Create an IoT Edge device:

```
az iot hub device-identity create --device-id edge001 -hub-name CLIDemoHub --edge-enabled
```



```
kg@IoTUbuntu:~$ az iot hub device-identity create -d edge001 -n CLIDemoHub --edge-enabled
{
  "authentication": {
    "symmetricKey": {
      "primaryKey": "[redacted]",
      "secondaryKey": "[redacted]"
    },
    "type": "sas",
    "x509Thumbprint": {
      "primaryThumbprint": null,
      "secondaryThumbprint": null
    }
  },
  "capabilities": {
    "iotEdge": true
  },
  "cloudToDeviceMessageCount": 0,
  "connectionState": "Disconnected",
  "connectionStateUpdatedTime": "0001-01-01T00:00:00",
  "deviceId": "edge001",
  "etag": "MzczODEzMDg4",
  "generationId": "636534545720393060",
  "lastActivityTime": "0001-01-01T00:00:00",
  "status": "enabled",
  "statusReason": null,
  "statusUpdatedTime": "0001-01-01T00:00:00"
}
```

## Configure the IoT Edge device

Create a deployment JSON template, and save it locally as a txt file. You will need the path to the file when you run the apply-configuration command.

Deployment JSON templates should always include the two system modules, edgeAgent and edgeHub. In addition to those two, you can use this file to deploy additional modules to the IoT Edge device. Use the following sample to configure you IoT Edge device with one tempSensor module:

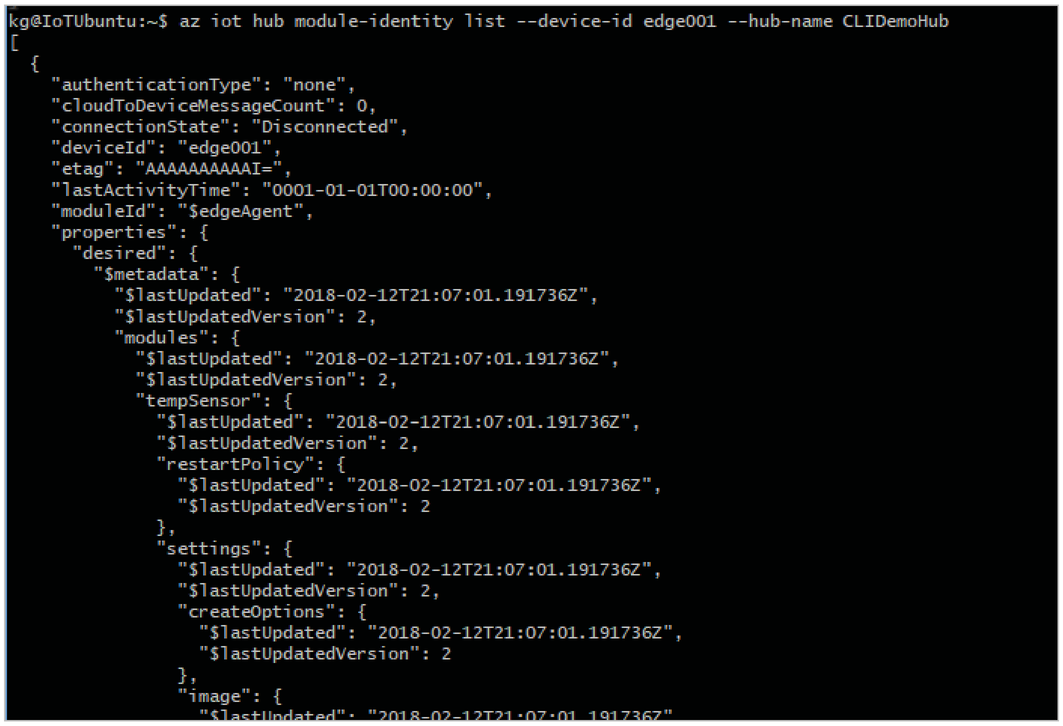
```
{
  "moduleContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": ""
          }
        },
        "systemModules": {
          "edgeAgent": {
            "type": "docker",
            "settings": {
              "image": "microsoft/azureiotedge-agent:1.0-preview",
              "createOptions": "{}"
            }
          },
          "edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "microsoft/azureiotedge-hub:1.0-preview",
              "createOptions": "{}"
            }
          }
        },
        "modules": {
          "tempSensor": {
            "version": "1.0",
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview",
              "createOptions": "{}"
            }
          }
        }
      }
    },
    "$edgeHub": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "routes": {},
        "storeAndForwardConfiguration": {
          "timeToLiveSecs": 7200
        }
      }
    },
    "tempSensor": {
      "properties.desired": {}
    }
  }
}
```

Apply the configuration to your IoT Edge device:

```
az iot hub apply-configuration --device-id edge001 --hub-name CLIDemoHub --content C:\<configuration.txt file
path>
```

View the modules on your IoT Edge device:

```
az iot hub module-identity list --device-id edge001 --hub-name CLIDemoHub
```



```
kg@IoTUbuntu:~$ az iot hub module-identity list --device-id edge001 --hub-name CLIDemoHub
[
  {
    "authenticationType": "none",
    "cloudToDeviceMessageCount": 0,
    "connectionState": "Disconnected",
    "deviceId": "edge001",
    "etag": "AAAAAAAAAAI=",
    "lastActivityTime": "0001-01-01T00:00:00",
    "moduleId": "$edgeAgent",
    "properties": {
      "desired": {
        "$metadata": {
          "$lastUpdated": "2018-02-12T21:07:01.191736Z",
          "$lastUpdatedVersion": 2,
          "modules": {
            "$lastUpdated": "2018-02-12T21:07:01.191736Z",
            "$lastUpdatedVersion": 2,
            "tempSensor": {
              "$lastUpdated": "2018-02-12T21:07:01.191736Z",
              "$lastUpdatedVersion": 2,
              "restartPolicy": {
                "$lastUpdated": "2018-02-12T21:07:01.191736Z",
                "$lastUpdatedVersion": 2
              },
              "settings": {
                "$lastUpdated": "2018-02-12T21:07:01.191736Z",
                "$lastUpdatedVersion": 2,
                "createOptions": {
                  "$lastUpdated": "2018-02-12T21:07:01.191736Z",
                  "$lastUpdatedVersion": 2
                },
                "image": {
                  "$lastUpdated": "2018-02-12T21:07:01.191736Z"
```

## Next steps

- Learn how to use an IoT Edge device as a gateway

# Deploy and monitor IoT Edge modules at scale - preview

5/7/2018 • 7 min to read • Edit Online

Azure IoT Edge enables you to move analytics to the edge and provides a cloud interface so that you can manage and monitor your IoT Edge devices without having to physically access each one. The capability to remotely manage devices is increasingly important as Internet of Things solutions are growing larger and more complex. Azure IoT Edge is designed to support your business goals, no matter how many devices you add.

You can manage individual devices and deploy modules to them one at a time. However, if you want to make changes to devices at a large scale, you can create an **IoT Edge automatic deployment**, which is part of Automatic Device Management in IoT Hub. Deployments are dynamic processes that enable you to deploy multiple modules to multiple devices at once, track the status and health of the modules, and make changes when necessary.

## Identify devices using tags

Before you can create a deployment, you have to be able to specify which devices you want to affect. Azure IoT Edge identifies devices using **tags** in the device twin. Each device can have multiple tags, and you can define them any way that makes sense for your solution. For example, if you manage a campus of smart buildings, you may add the following tags to a device:

```
"tags":{
    "location":{
        "building": "20",
        "floor": "2"
    },
    "roomtype": "conference",
    "environment": "prod"
}
```

For more information about device twins and tags, see Understand and use device twins in IoT Hub.

## Create a deployment

1. In the Azure portal, go to your IoT hub.
2. Select **IoT Edge (preview)**.
3. Select **Add IoT Edge Deployment**.

There are five steps to create a deployment. The following sections walk through each one.

### Step 1: Name and Label

1. Give your deployment a unique name. Avoid spaces and the following invalid characters: `& ^ [ ] { } \ | " < > /`.
2. Add labels to help track your deployments. Labels are **Name**, **Value** pairs that describe your deployment. For example, `HostPlatform, Linux` or `Version, 3.0.1`.
3. Select **Next** to move to step two.

### Step 2: Add Modules (optional)

There are two types of modules that you can add to a deployment. The first is a module based off of an Azure service, like Storage Account or Stream Analytics. The second is a module based off of your own code. You can add multiple modules of either type to a deployment.

If you create a deployment with no modules, it removes any existing modules from the devices.

**NOTE**

Azure Machine Learning and Azure Functions don't support the automated Azure service deployment yet. Use the custom module deployment to manually add those services to your deployment.

To add a module from Azure Stream Analytics, follow these steps:

1. Select **Import Azure Stream Analytics IoT Edge module**.
2. Use the drop-down menus to select the Azure service instances that you want to deploy.
3. Select **Save** to add your module to the deployment.

To add custom code as a module, or to manually add an Azure service module, follow these steps:

1. Select **Add IoT Edge module**.
2. Give your module a **Name**.
3. For the **Image URI** field, enter the Docker container image for your module.
4. Specify any **Container Create Options** that should be passed to the container. For more information, see docker create.
5. Use the drop-down menu to select a **Restart policy**. Choose from the following options:
   - **Always** - The module always restarts if it shuts down for any reason.
   - **Never** - The module never restarts if it shuts down for any reason.
   - **On-failed** - The module restarts if it crashes, but not if it shuts down cleanly.
   - **On-unhealthy** - The module restarts if it crashes or returns an unhealthy status. It's up to each module to implement the health status function.
6. Use the drop-down menu to select the **Desired Status** for the module. Choose from the following options:
   - **Running** - This is the default option. The module will start running immediately after being deployed.
   - **Stopped** - After being deployed, the module will remain idle until called upon to start by you or another module.
7. Select **Enable** if you want to add any tags or desired properties to the module twin.
8. Select **Save** to add your module to the deployment.

Once you have all the modules for a deployment configured, select **Next** to move to step three.

### Step 3: Specify Routes (optional)

Routes define how modules communicate with each other within a deployment. Specify any routes for your deployment, then select **Next** to move to step four.

### Step 4: Target Devices

Use the tags property from your devices to target the specific devices that should receive this deployment.

Since multiple deployments may target the same device, you should give each deployment a priority number. If there's ever a conflict, the deployment with the highest priority wins. If two deployments have the same priority number, the one that was created most recently wins.

1. Enter a positive integer for the deployment **Priority**.
2. Enter a **Target condition** to determine which devices will be targeted with this deployment. The condition is based on device twin tags and should match the expression format. For example, `tags.environment='test'`.
3. Select **Next** to move on to the final step.

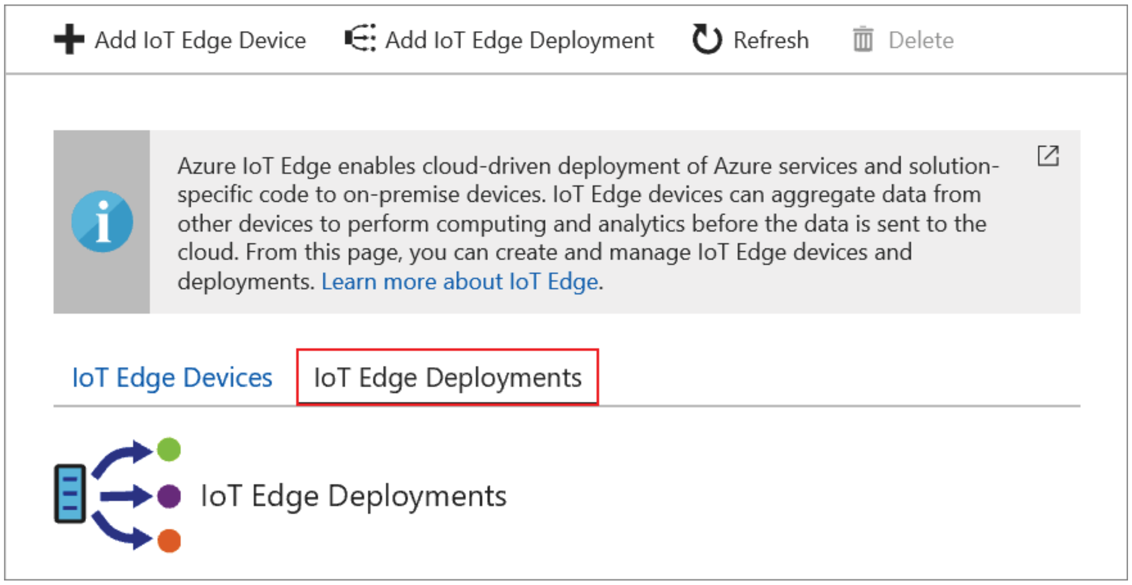
**Step 5: Review Template**

Review your deployment information, then select **Submit**.

## Monitor a deployment

To view the details of a deployment and monitor the devices running it, use the following steps:

1. Sign in to the Azure portal and navigate to your IoT hub.
2. Select **IoT Edge (preview)**.
3. Select **IoT Edge deployments**.



Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. Learn more about IoT Edge.

4. Inspect the deployment list. For each deployment, you can view the following details:

   - **ID** - the name of the deployment.
   - **Target condition** - the tag used to define targeted devices.
   - **Priority** - the priority number assigned to the deployment.
   - **IoT Edge agent status** - the number of devices that received the deployment, and their health statuses.
   - **Unhealthy modules** - the number of modules in the deployment reporting errors.
   - **Creation time** - the timestamp from when the deployment was created. This timestamp is used to break ties when two deployments have the same priority.

5. Select the deployment that you want to monitor.
6. Inspect the deployment details. You can use tabs to view specific details about the devices that received the deployment:
   - **Targeted** - the Edge devices that match the target condition.
   - **Applied** - the targeted Edge devices that are not targeted by another deployment of higher priority. These are the devices that actually receive the deployment.
   - **Reporting success** - the applied Edge devices that reported back to the service that the modules were deployed successfully.
   - **Reporting failure** - the applied Edge devices that reported back to the service that one or more modules were not deployed successfully. To further investigate the error, you will need to connect remotely to those devices and view the log files.
   - **Reporting unhealthy modules** - the applied Edge devices that reported back to the service that one or more modules were deployed successfully, but are now reporting errors.

## Modify a deployment

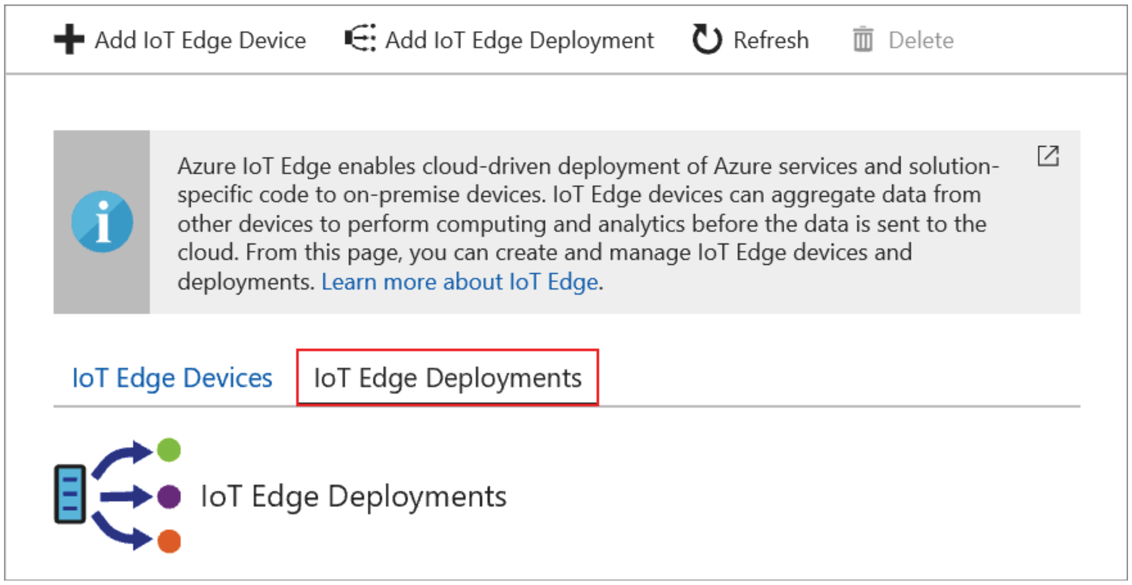
When you modify a deployment, the changes immediately replicate to all targeted devices.

If you update the target condition, the following updates occur:

- If a device didn't meet the old target condition, but meets the new target condition and this deployment is the highest priority for that device, then this deployment is applied to the device.
- If a device currently running this deployment no longer meets the target condition, it uninstalls this deployment and takes on the next highest priority deployment.
- If a device currently running this deployment no longer meets the target condition and doesn't meet the target condition of any other deployments, then no change occurs on the device. The device continues running its current modules in their current state, but is not managed as part of this deployment anymore. Once it meets the target condition of any other deployment, it uninstalls this deployment and takes on the new one.

To modify a deployment, use the following steps:

1. Sign in to the Azure portal and navigate to your IoT hub.
2. Select **IoT Edge (preview)**.
3. Select **IoT Edge deployments**.



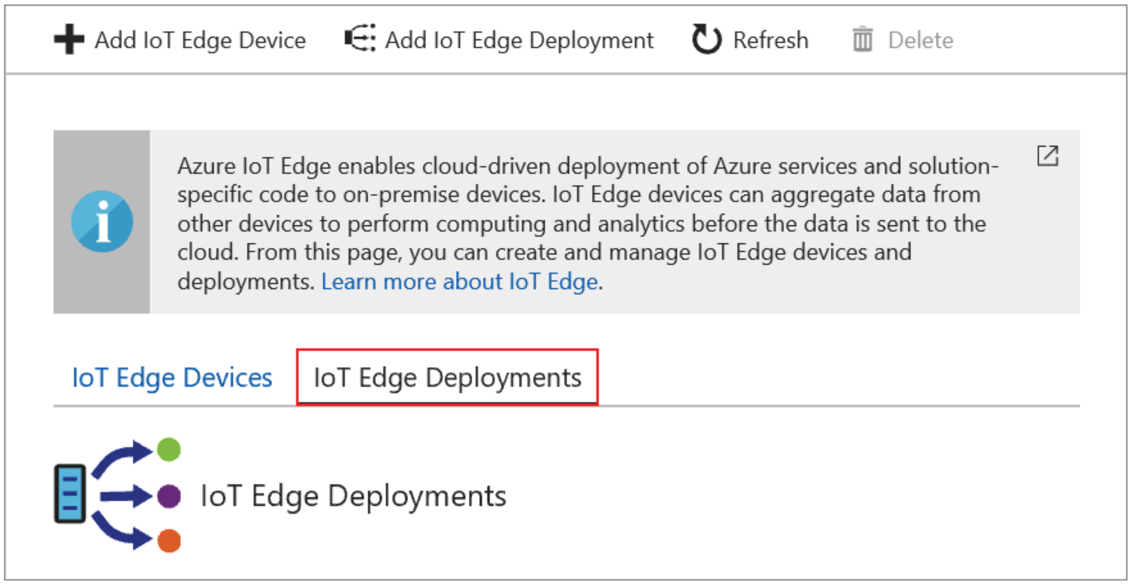
Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. Learn more about IoT Edge.

4. Select the deployment that you want to modify.
5. Make updates to the following fields:
   - Target condition
   - Labels
   - Priority
6. Select **Save**.
7. Follow the steps in Monitor a deployment to watch the changes roll out.

## Delete a deployment

When you delete a deployment, any devices take on their next highest priority deployment. If your devices don't meet the target condition of any other deployment, then the modules are not removed when the deployment is deleted.

1. Sign in to the Azure portal and navigate to your IoT hub.
2. Select **IoT Edge (preview)**.
3. Select **IoT Edge deployments**.


Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. Learn more about IoT Edge.

4. Use the checkbox to select the deployment that you want to delete.
5. Select **Delete**.
6. A prompt will inform you that this action will delete this deployment and revert to the previous state for all devices. This means that a deployment with a lower priority will apply. If no other deployment is targeted, no modules will be removed. If customers wish to do this, they need to create a deployment with zero modules and deploy it to the same devices. Select **Yes** if you wish to continue.

## Next steps

Learn more about Deploying modules to Edge devices.

# Install the IoT Edge runtime on Windows IoT Core - preview

3/6/2018 • 1 min to read • Edit Online

Azure IoT Edge and Windows IoT Core work together to enable edge computing on even small devices. The Azure IoT Edge Runtime can run even on tiny Single Board Computer (SBC) devices which are very prevalent in the IoT industry.

This article walks through provisioning the runtime on a MinnowBoard Turbot development board running Windows IoT Core. Windows IoT Core supports Azure IoT Edge only on Intel x64-based processors.

## Install the runtime

1. Install Windows 10 IoT Core Dashboard on a host system.
2. Follow the steps in Set up your device to configure your board with the MinnowBoard Turbot/MAX Build 16299 image.
3. Turn on the device, then login remotely with PowerShell.
4. In the PowerShell console, install the container runtime:

   ```
   Invoke-WebRequest https://master.dockerproject.org/windows/x86_64/docker-0.0.0-dev.zip -o temp.zip
   Expand-Archive .\temp.zip $env:ProgramFiles -f
   Remove-Item .\temp.zip
   $env:Path += ";$env:programfiles\docker"
   SETX /M PATH "$env:Path"
   dockerd --register-service
   start-service docker
   ```

   **NOTE**

   This container runtime is from the Moby project build server, and is intended for evaluation purposes only. It's not tested, endorsed, or supported by Docker.

5. Install the IoT Edge runtime and verify your configuration:

   ```
   Invoke-Expression (Invoke-WebRequest -useb https://aka.ms/iotedgewin)
   ```

   This script provides the following:

   - Python 3.6
   - The IoT Edge control script (iotedgectl.exe)

You may see informational output from the iotedgectl.exe tool in green in the remote PowerShell window. This doesn't necessarily indicate errors.

## Next steps

Now that you have a device running the IoT Edge runtime, learn how to Deploy and monitor IoT Edge modules at scale.

# Develop an IoT Edge solution with multiple modules in Visual Studio Code - preview

4/9/2018 • 3 min to read • Edit Online

You can use Visual Studio Code to develop your IoT Edge solution with multiple modules. This article walks through creating, updating, and deploying an IoT Edge solution that pipes sensor data on the simulated IoT Edge device in Visual Studio Code. In this article, you learn how to:

- Use Visual Studio Code to create an IoT Edge solution
- Use VS Code to add a new module to your working IoT Edge solution.
- Deploy the IoT Edge solution (multiple modules) to your IoT Edge device
- View generated data

## Prerequisites

- Complete below tutorials
  - Deploy C# module
  - Deploy C# Function
  - Deploy Python module
- Docker for VS Code with explorer integration for managing Images and Containers.

## Prepare your first IoT Edge solution

1. In VS Code command palette, type and run the command **Edge: New IoT Edge solution**. Then select your workspace folder, provide the solution name (The default name is **EdgeSolution**), and create a C# Module (**SampleModule**) as the first user module in this solution. You also need to specify the Docker image repository for your first module. The default image repository is based on a local Docker registry ( `localhost:5000/<first module name>` ). You can also change it to Azure container registry or Docker Hub.

> **NOTE**
>
> If you are using a local Docker registry, please make sure the registry is running by typing the command `docker run -d -p 5000:5000 --restart=always --name registry registry:2` in your console window.

1. The VS Code window will load your IoT Edge solution workspace. There is a `modules` folder, a `.vscode` folder and a deployment manifest template file in the root folder. You can see debug configurations in `.vscode` folder. All user module codes will be subfolders under the folder `modules`. The `deployment.template.json` is the deployment manifest template. Some of the parameters in this file will be parsed from the `module.json`, which exists in every module folder.

2. Add your second module into this solution project. This time type and run **Edge: Add IoT Edge module** and select the deployment template file to update. Then select an **Azure Function - C#** with name **SampleFunction** and its Docker image repository to add.

3. Now your first IoT Edge solution with two basic modules is ready. The default C# module acts as a pipe message module while the C# Funtion acts as a pipe message function. In the `deployment.template.json`, you will see this solution contains three modules. The message will be generated from the `tempSensor` module, and will be directly piped via `SampleModule` and `SampleFunction`, then sent to your IoT hub. Update

the routes for these modules with below content.

```
    "routes": {
      "SensorToPipeModule": "FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/SampleModule/inputs/input1\")",
      "PipeModuleToPipeFunction": "FROM /messages/modules/SampleModule/outputs/output1 INTO
BrokeredEndpoint(\"/modules/SampleFunction/inputs/input1\")",
      "PipeFunctionToIoTHub": "FROM /messages/modules/SampleFunction/outputs/output1 INTO $upstream"
    },
```

4. Save this file.

## Build and deploy your IoT Edge solution

1. In VS Code command palette, type and run the command **Edge: Build IoT Edge solution**. Based on the `module.json` file in each module folder, this command will check and start to build, containerize and push each module docker image. Then it will parse the required value to `deployment.template.json`, generate the `deployment.json` with actual value under `config` folder. You can see the build progress in VS Code integrated terminal.

2. In Azure IoT Hub Devices explorer, right-click an IoT Edge device ID, then select **Create deployment for Edge device**. Select the `deployment.json` under `config` folder. Then you can see the deployment is successfully created with a deployment ID in VS Code integrated terminal.

3. If you are simulating an IoT Edge device on your development machine. You will see that all the module image containers will be started in a few minutes.

## View generated data

1. To monitor data arriving at the IoT hub, select the **View** > **Command Palette...** and search for **IoT: Start monitoring D2C message**.
2. To stop monitoring data, use the **IoT: Stop monitoring D2C message** command in the Command Palette.

## Next steps

You can continue on to either of the following articles to learn about other scenarios when developing Azure IoT Edge in Visual Studio Code:

- Debug a C# module in VS Code
- Debug a C# Function in VS Code

# Use Visual Studio Code to debug a C# module with Azure IoT Edge

5/2/2018 • 2 min to read • Edit Online

This article provides detailed instructions for using Visual Studio Code as the main development tool to debug your Azure IoT Edge modules.

## Prerequisites

This article assumes that you are using a computer or virtual machine running Windows or Linux as your development machine. Your IoT Edge device can be another physical device, or you can simulate your IoT Edge device on your development machine.

**NOTE**

You can only debug C# module in linux-amd64 containers.

Before following the guidance in this article, complete the steps in Develop an IoT Edge solution with multiple modules in Visual Studio Code. After that, you should have the following items ready:

- A local Docker registry running on your development machine. It is suggested to use a local Docker registry for prototype and testing purpose. You can update the container registry in the `module.json` file in each module folder.
- An IoT Edge solution project workspace with a C# module subfolder in it.
- The `Program.cs` file, with the latest module code.
- An Edge runtime running on your development machine.

## Build your IoT Edge C# module for debugging

1. To start debugging, you need to use the **Dockerfile.amd64.debug** to rebuild your docker image and deploy your Edge solution again. In VS Code explorer, navigate to `deployment.template.json` file. Update your function image URL by adding a `.debug` in the end.
2. Rebuild your solution. In VS Code command palette, type and run the command **Edge: Build IoT Edge solution**.
3. In Azure IoT Hub Devices explorer, right-click an IoT Edge device ID, then select **Create deployment for Edge device**. Select the `deployment.json` under `config` folder. Then you can see the deployment is successfully created with a deployment ID in VS Code integrated terminal.

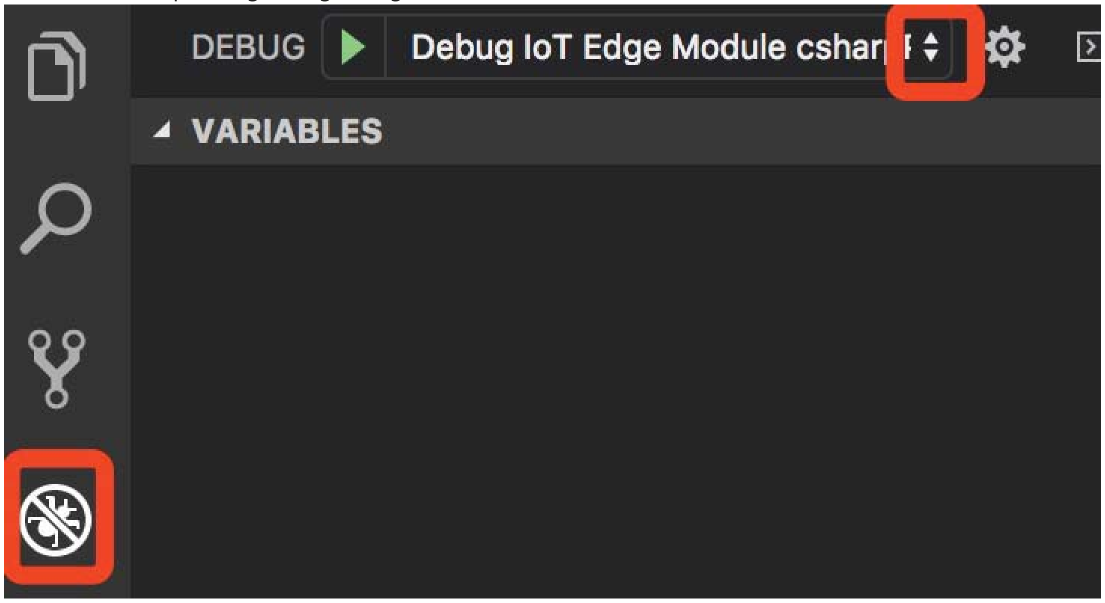
**NOTE**

You can check your container status in the VS Code Docker explorer or by run the `docker images` command in the terminal.

## Start debugging C# module in VS Code

1. VS Code keeps debugging configuration information in a `launch.json` file located in a `.vscode` folder in

your workspace. This `launch.json` file has been generated when creating a new IoT Edge solution. And it will be updated each time you add a new module that support debugging. Navigate to the debug view and select the corresponding debug configuration file.



2. Navigate to `program.cs`. Add a breakpoint in this file.

3. Click Start Debugging button or press **F5**, and select the process to attach to.

4. In VS Code Debug view, you can see the variables in left panel.

> **NOTE**
>
> The preceding example shows how to debug .NET Core IoT Edge modules on containers. It's based on the debug version of the `Dockerfile.debug`, which includes VSDBG (the .NET Core command-line debugger) in your container image while building it. After you finish debugging your C# modules, we recommend you directly use or customize `Dockerfile` without VSDBG for production-ready IoT Edge modules.

## Next steps

Use Visual Studio Code to debug Azure Functions with Azure IoT Edge

# Use Visual Studio Code to debug Azure Functions with Azure IoT Edge

5/2/2018 • 2 min to read • Edit Online

This article provides detailed instructions for using Visual Studio Code as the main development tool to debug your Azure Functions on IoT Edge.

## Prerequisites

This article assumes that you are using a computer or virtual machine running Windows or Linux as your development machine. Your IoT Edge device could be another physical device or you can simulate your IoT Edge device on your development machine.

**NOTE**

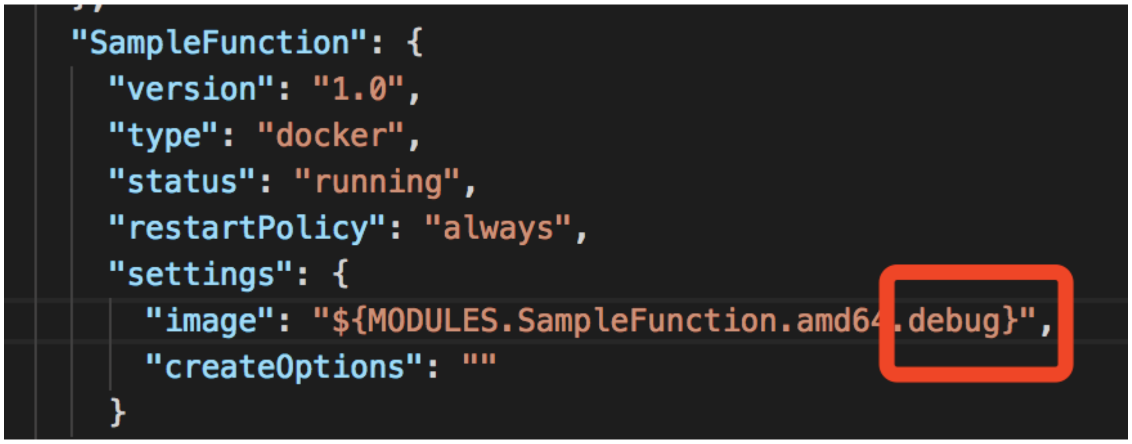
You can only debug C# Functions in linux-amd64 containers.

Before following the guidance in this article, complete the steps in Develop an IoT Edge solution with multiple modules in Visual Studio Code. After that, you should have the following items ready:

- A local Docker registry running on your development machine. It is suggested to use a local Docker registry for prototype and testing purpose. You can update the container registry in the `module.json` file in each module folder.
- An IoT Edge solution project workspace with an Azure Function module subfolder in it.
- The `run.csx` file with your function code.
- An Edge runtime running on your development machine.

## Build your IoT Edge Function module for debugging purpose

1. To start debugging, you need to use the **Dockerfile.amd64.debug** to rebuild your docker image and deploy your Edge solution again. In VS Code explorer, navigate to `deployment.template.json` file. Update your function image URL by adding a `.debug` in the end.



```
"SampleFunction": {
  "version": "1.0",
  "type": "docker",
  "status": "running",
  "restartPolicy": "always",
  "settings": {
    "image": "${MODULES.SampleFunction.amd64.debug}",
    "createOptions": ""
  }
```

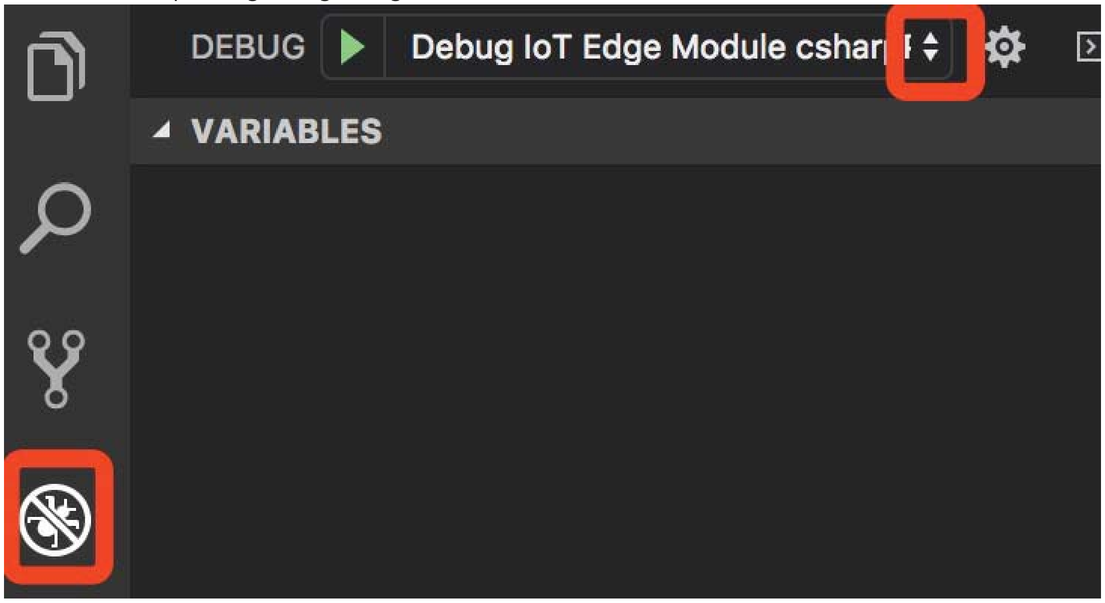
2. Rebuild your solution. In VS Code command palette, type and run the command **Edge: Build IoT Edge solution**.

3. In Azure IoT Hub Devices explorer, right-click an IoT Edge device ID, then select **Create deployment for Edge device**. Select the `deployment.json` under `config` folder. Then you can see the deployment is successfully created with a deployment ID in VS Code integrated terminal.

> **NOTE**
> You can check your container status in the VS Code Docker explorer or by run the `docker images` command in the terminal.

## Start debugging C# Function in VS Code

1. VS Code keeps debugging configuration information in a `launch.json` file located in a `.vscode` folder in your workspace. This `launch.json` file has been generated when creating a new IoT Edge solution. And it will be updated each time you add a new module that support debugging. Navigate to the debug view and select the corresponding debug configuration file.



2. Navigate to `run.csx`. Add a breakpoint in the function.

3. Click Start Debugging button or press **F5**, and select the process to attach to.

4. In VS Code Debug view, you can see the variables in left panel.

> **NOTE**
> Above example shows how to debugging .Net Core IoT Edge Function on containers. It's based on the debug version of the `Dockerfile.amd64.debug`, which includes VSDBG(the .NET Core command-line debugger) in your container image while building it. We recommend you directly use or customize the `Dockerfile` without VSDBG for production-ready IoT Edge function after you finish debugging your C# function.

## Next steps

Use Visual Studio Code to debug a C# module with Azure IoT Edge

# Continuous integration and continuous deployment to Azure IoT Edge - preview

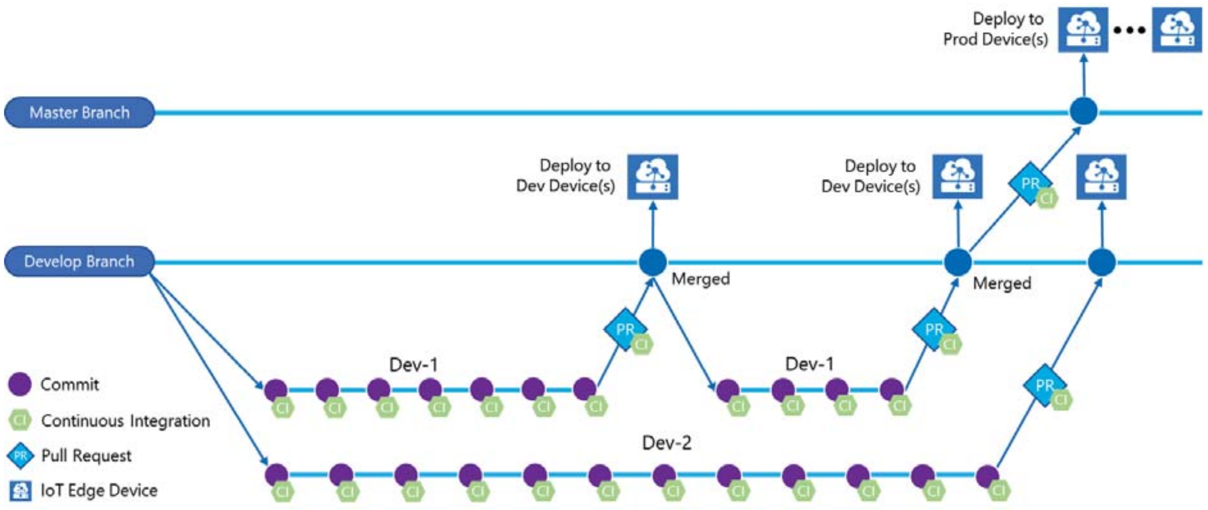
4/30/2018 • 11 min to read • Edit Online

This tutorial demonstrates how you can use the continuous integration and continuous deployment features of Visual Studio Team Services (VSTS) and Microsoft Team Foundation Server (TFS) to build, test, and deploy applications quickly and efficiently to your Azure IoT Edge.

In this tutorial, you will learn how to:

- Create and check in a sample IoT Edge solution containing unit tests.
- Install Azure IoT Edge extension for your VSTS.
- Configure continuous integration (CI) to build the solution and run the unit tests.
- Configure continuous deployment (CD) to deploy the solution and view responses.

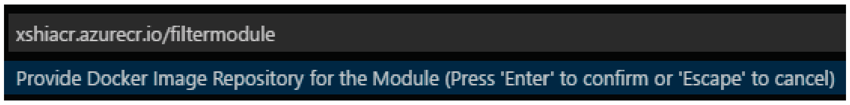
It will take 30 minutes to complete this tutorial.



## Create a sample Azure IoT Edge solution using Visual Studio Code

In this section, you will create a sample IoT Edge solution containing unit tests that you can execute as part of the build process. Before following the guidance in this section, complete the steps in Develop an IoT Edge solution with multiple modules in Visual Studio Code.

1. In VS Code command palette, type and run the command **Edge: New IoT Edge solution**. Then select your workspace folder, provide the solution name (The default name is **EdgeSolution**), and create a C# Module (**FilterModule**) as the first user module in this solution. You also need to specify the Docker image repository for your first module. The default image repository is based on a local Docker registry ( `localhost:5000/filtermodule` ). You need to change it to Azure Container Registry( `<your container registry address>/filtermodule` ) or Docker Hub for further continuous integration.

   xshiacr.azurecr.io/filtermodule

   Provide Docker Image Repository for the Module (Press 'Enter' to confirm or 'Escape' to cancel)

2. The VS Code window will load your IoT Edge solution workspace. You can optionally type and run **Edge: Add IoT Edge module** to add more modules. There is a `modules` folder, a `.vscode` folder, and a

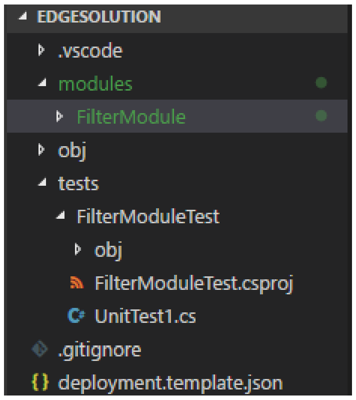
deployment manifest template file in the root folder. All user module codes will be subfolders under the folder `modules`. The `deployment.template.json` is the deployment manifest template. Some of the parameters in this file will be parsed from the `module.json`, which exists in every module folder.

3. Now your sample IoT Edge solution is ready. The default C# module acts as a pipe message module. In the `deployment.template.json`, you will see this solution contains two modules. The message will be generated from the `tempSensor` module, and will be directly piped via `FilterModule`, then sent to your IoT hub. Replace the entire **Program.cs** file with below content. For more information about this code snippet, you can refer to Create an IoT Edge C# module project.

```
namespace FilterModule
{
    using System;
    using System.IO;
    using System.Runtime.InteropServices;
    using System.Runtime.Loader;
    using System.Security.Cryptography.X509Certificates;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.Devices.Client;
    using Microsoft.Azure.Devices.Client.Transport.Mqtt;
    using System.Collections.Generic;     // for KeyValuePair<>
    using Microsoft.Azure.Devices.Shared; // for TwinCollection
    using Newtonsoft.Json;                // for JsonConvert

    public class MessageBody
    {
        public Machine machine { get; set; }
        public Ambient ambient { get; set; }
        public string timeCreated { get; set; }
    }
    public class Machine
    {
        public double temperature { get; set; }
        public double pressure { get; set; }
    }
    public class Ambient
    {
        public double temperature { get; set; }
        public int humidity { get; set; }
    }

    public class Program
    {
        static int counter;
        static int temperatureThreshold { get; set; } = 25;

        static void Main(string[] args)
        {
            // The Edge runtime gives us the connection string we need -- it is injected as an
environment variable
            string connectionString = Environment.GetEnvironmentVariable("EdgeHubConnectionString");

            // Cert verification is not yet fully functional when using Windows OS for the container
            bool bypassCertVerification = RuntimeInformation.IsOSPlatform(OSPlatform.Windows);
            if (!bypassCertVerification) InstallCert();
            Init(connectionString, bypassCertVerification).Wait();

            // Wait until the app unloads or is cancelled
            var cts = new CancellationTokenSource();
            AssemblyLoadContext.Default.Unloading += (ctx) => cts.Cancel();
            Console.CancelKeyPress += (sender, cpe) => cts.Cancel();
            WhenCancelled(cts.Token).Wait();
        }
        /// <summary>
        /// Handles cleanup operations when app is cancelled or unloads
        /// </summary>
        public static Task WhenCancelled(CancellationToken cancellationToken)
        {
            var tcs = new TaskCompletionSource<bool>();
            cancellationToken.Register(s => ((TaskCompletionSource<bool>)s).SetResult(true), tcs);
            return tcs.Task;
        }

        /// <summary>
        /// Add certificate in local cert store for use by client for secure connection to IoT Edge
runtime
        /// </summary>
        static void InstallCert()
        {
            string certPath = Environment.GetEnvironmentVariable("EdgeModuleCACertificateFile");
            if (string.IsNullOrWhiteSpace(certPath))
            {
                // We cannot proceed further without a proper cert file
                Console.WriteLine($"Missing path to certificate collection file: {certPath}");
                throw new InvalidOperationException("Missing path to certificate file.");
            }
            else if (!File.Exists(certPath))
            {
                // We cannot proceed further without a proper cert file
                Console.WriteLine($"Missing path to certificate collection file: {certPath}");
                throw new InvalidOperationException("Missing certificate file.");
            }
            X509Store store = new X509Store(StoreName.Root, StoreLocation.CurrentUser);
            store.Open(OpenFlags.ReadWrite);
            store.Add(new X509Certificate2(X509Certificate2.CreateFromCertFile(certPath)));
            Console.WriteLine("Added Cert: " + certPath);
            store.Close();
        }
        /// <summary>
        /// Initializes the DeviceClient and sets up the callback to receive
        /// messages containing temperature information
        /// </summary>
        static async Task Init(string connectionString, bool bypassCertVerification = false)
        {
            Console.WriteLine("Connection String {0}", connectionString);

            MqttTransportSettings mqttSetting = new MqttTransportSettings(TransportType.Mqtt_Tcp_Only);
            // During dev you might want to bypass the cert verification. It is highly recommended to
verify certs systematically in production
            if (bypassCertVerification)
            {
                mqttSetting.RemoteCertificateValidationCallback = (sender, certificate, chain,
sslPolicyErrors) => true;
            }
            ITransportSettings[] settings = { mqttSetting };

            // Open a connection to the Edge runtime
            DeviceClient ioTHubModuleClient = DeviceClient.CreateFromConnectionString(connectionString,
settings);
            await ioTHubModuleClient.OpenAsync();
            Console.WriteLine("IoT Hub module client initialized.");

            // Register callback to be called when a message is received by the module
            // await ioTHubModuleClient.SetImputMessageHandlerAsync("input1", PipeMessage,
iotHubModuleClient);

            // Read TemperatureThreshold from Module Twin Desired Properties
            var moduleTwin = await ioTHubModuleClient.GetTwinAsync();
            var moduleTwinCollection = moduleTwin.Properties.Desired;
            try {
                temperatureThreshold = moduleTwinCollection["TemperatureThreshold"];
            } catch(ArgumentOutOfRangeException) {
            } catch(ArgumentOutOfRangeException) {
                Console.WriteLine("Proerty TemperatureThreshold not exist");
            }

            // Attach callback for Twin desired properties updates
            await ioTHubModuleClient.SetDesiredPropertyUpdateCallbackAsync(onDesiredPropertiesUpdate,
null);

            // Register callback to be called when a message is received by the module
            await ioTHubModuleClient.SetInputMessageHandlerAsync("input1", FilterMessages,
ioTHubModuleClient);
        }

        static Task onDesiredPropertiesUpdate(TwinCollection desiredProperties, object userContext)
        {
            try
            {
                Console.WriteLine("Desired property change:");
                Console.WriteLine(JsonConvert.SerializeObject(desiredProperties));

                if (desiredProperties["TemperatureThreshold"] != null)
                    temperatureThreshold = desiredProperties["TemperatureThreshold"];

            }
            catch (AggregateException ex)
            {
                foreach (Exception exception in ex.InnerExceptions)
                {
                    Console.WriteLine();
                    Console.WriteLine("Error when receiving desired property: {0}", exception);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine();
                Console.WriteLine("Error when receiving desired property: {0}", ex.Message);
            }
            return Task.CompletedTask;
        }

        public static Message filter(Message message)
        {
            var counterValue = Interlocked.Increment(ref counter);

            var messageBytes = message.GetBytes();
            var messageString = Encoding.UTF8.GetString(messageBytes);
            Console.WriteLine($"Received message {counterValue}: [{messageString}]");

            // Get message body
            var messageBody = JsonConvert.DeserializeObject<MessageBody>(messageString);

            if (messageBody != null && messageBody.machine.temperature > temperatureThreshold)
            {
                Console.WriteLine($"Machine temperature {messageBody.machine.temperature} " +
                    $"exceeds threshold {temperatureThreshold}");
                var filteredMessage = new Message(messageBytes);
                foreach (KeyValuePair<string, string> prop in message.Properties)
                {
                    filteredMessage.Properties.Add(prop.Key, prop.Value);
                }

                filteredMessage.Properties.Add("MessageType", "Alert");
                return filteredMessage;
            }
            return null;
        }

        static async Task<MessageResponse> FilterMessages(Message message, object userContext)
        {
            try
            {
                DeviceClient deviceClient = (DeviceClient)userContext;

                var filteredMessage = filter(message);
                if (filteredMessage != null)
                {
                    await deviceClient.SendEventAsync("output1", filteredMessage);
                }

                // Indicate that the message treatment is completed
                return MessageResponse.Completed;
            }
            catch (AggregateException ex)
            {
                foreach (Exception exception in ex.InnerExceptions)
                {
                    Console.WriteLine();
                    Console.WriteLine("Error in sample: {0}", exception);
                }
                // Indicate that the message treatment is not completed
                var deviceClient = (DeviceClient)userContext;
                return MessageResponse.Abandoned;
            }
            catch (Exception ex)
            {
                Console.WriteLine();
                Console.WriteLine("Error in sample: {0}", ex.Message);
                // Indicate that the message treatment is not completed
                DeviceClient deviceClient = (DeviceClient)userContext;
                return MessageResponse.Abandoned;
            }
        }
    }
}
```

4. Create a .Net Core unit test project. In VS Code file explorer, create a new folder **tests\FilterModuleTest** in your workspace. Then in VS Code integrated terminal (**Ctrl + `**), run following commands to create a xunit test project and add reference to the **FilterModule** project.

```
cd tests\FilterModuleTest
dotnet new xunit
dotnet add reference ../../modules/FilterModule/FilterModule.csproj
```

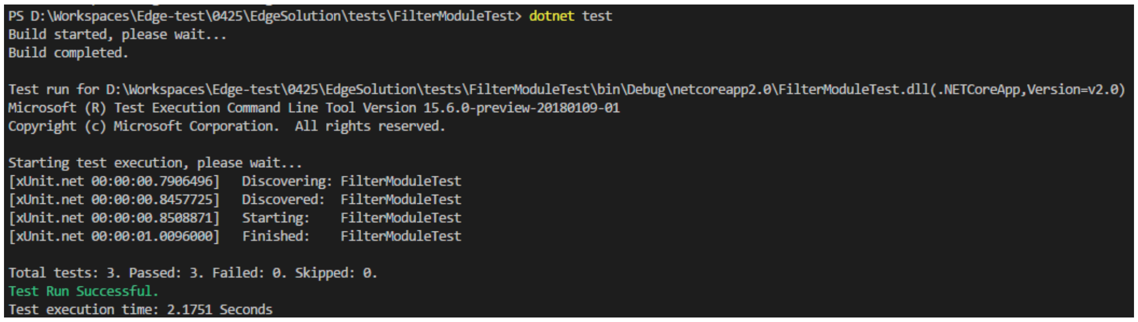


5. In the **FilterModuleTest** folder, update the file name of **UnitTest1.cs** to **FilterModuleTest.cs**. Select and open **FilterModuleTest.cs**, replace the entire code with below code snippet, which contains the unit tests against the FilterModule project.

```
using Xunit;
using FilterModule;
using Newtonsoft.Json;
using System;
using System.IO;
using System.Runtime.InteropServices;
using System.Runtime.Loader;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.Devices.Client.Transport.Mqtt;

namespace FilterModuleTest
{
    public class FilterModuleTest
    {
        [Fact]
        public void filterLessThanThresholdTest()
        {
            var source = createMessage(25 - 1);
            var result = Program.filter(source);
            Assert.True(result == null);
        }

        [Fact]
        public void filterMoreThanThresholdAlertPropertyTest()
        {
            var source = createMessage(25 + 1);
            var result = Program.filter(source);
            Assert.True(result.Properties["MessageType"] == "Alert");
        }

        [Fact]
        public void filterMoreThanThresholdCopyPropertyTest()
        {
            var source = createMessage(25 + 1);
            source.Properties.Add("customTestKey", "customTestValue");
            var result = Program.filter(source);
            Assert.True(result.Properties["customTestKey"] == "customTestValue");
        }

        private Message createMessage(int temperature)
        {
            var messageBody = createMessageBody(temperature);
            var messageString = JsonConvert.SerializeObject(messageBody);
            var messageBytes = Encoding.UTF8.GetBytes(messageString);
            return new Message(messageBytes);
        }

        private MessageBody createMessageBody(int temperature)
        {
            var messageBody = new MessageBody
            {
                machine = new Machine
                {
                    temperature = temperature,
                    pressure = 0
                },
                ambient = new Ambient
                {
                    temperature = 0,
                    humidity = 0
                },
                timeCreated = string.Format("{0:O}", DateTime.Now)
            };

            return messageBody;
        }
    }
    }
}
```

6. In integrated terminal, you can enter following commands to run unit tests locally.

```
dotnet test
```



```
PS D:\Workspaces\Edge-test\0425\EdgeSolution\tests\FilterModuleTest> dotnet test
Build started, please wait...
Build completed.

Test run for D:\Workspaces\Edge-test\0425\EdgeSolution\tests\FilterModuleTest\bin\Debug\netcoreapp2.0\FilterModuleTest.dll(.NETCoreApp,Version=v2.0)
Microsoft (R) Test Execution Command Line Tool Version 15.6.0-preview-20180109-01
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
[xUnit.net 00:00:00.7906496]   Discovering: FilterModuleTest
[xUnit.net 00:00:00.8457725]   Discovered:  FilterModuleTest
[xUnit.net 00:00:00.8508871]   Starting:    FilterModuleTest
[xUnit.net 00:00:01.0096000]   Finished:    FilterModuleTest

Total tests: 3. Passed: 3. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 2.1751 Seconds
```

7. Save these projects, then check it into your VSTS or TFS repository.

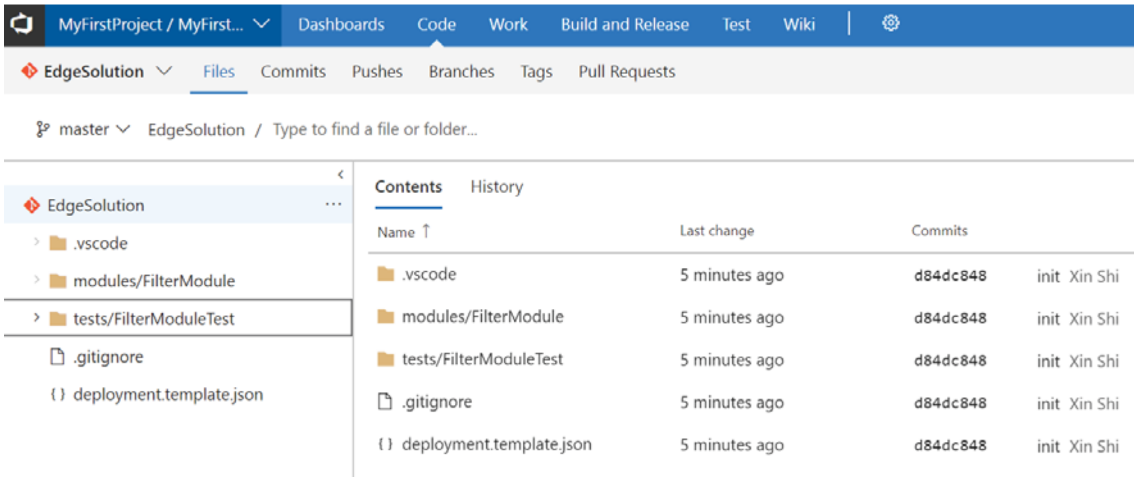
**NOTE**

For more information about using VSTS code repositories, see Share your code with Visual Studio and VSTS Git.

## Configure continuous integration

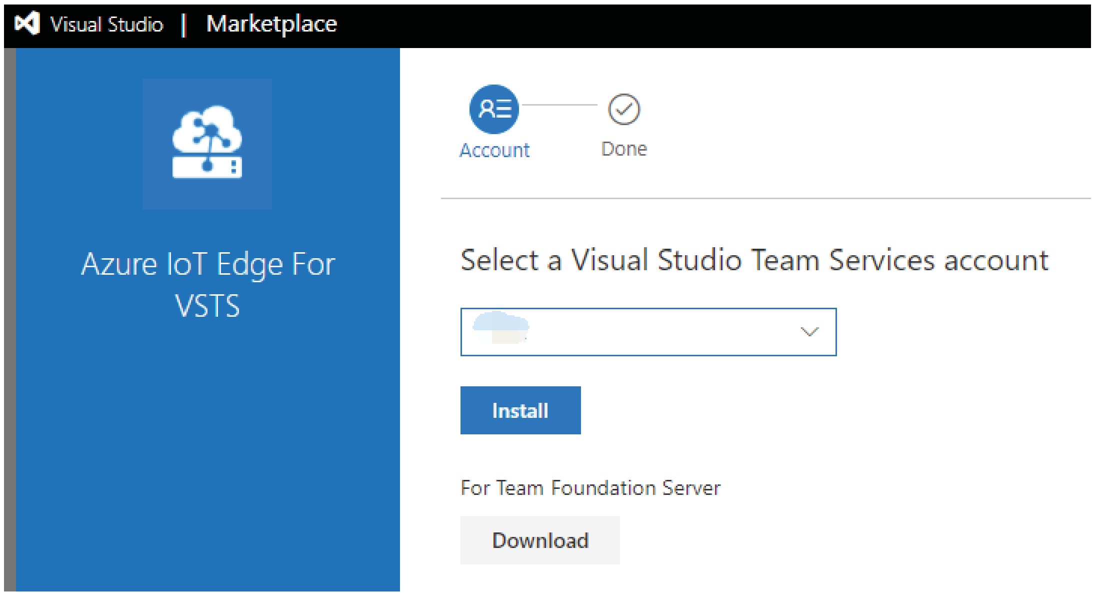
In this section, you will create a build definition that is configured to run automatically when you check in any changes to the sample IoT Edge solution, and it will automatically execute the unit tests it contains.

1. Sign into your VSTS account (**https://***your-account***.visualstudio.com**) and open the project where you checked in the sample app.

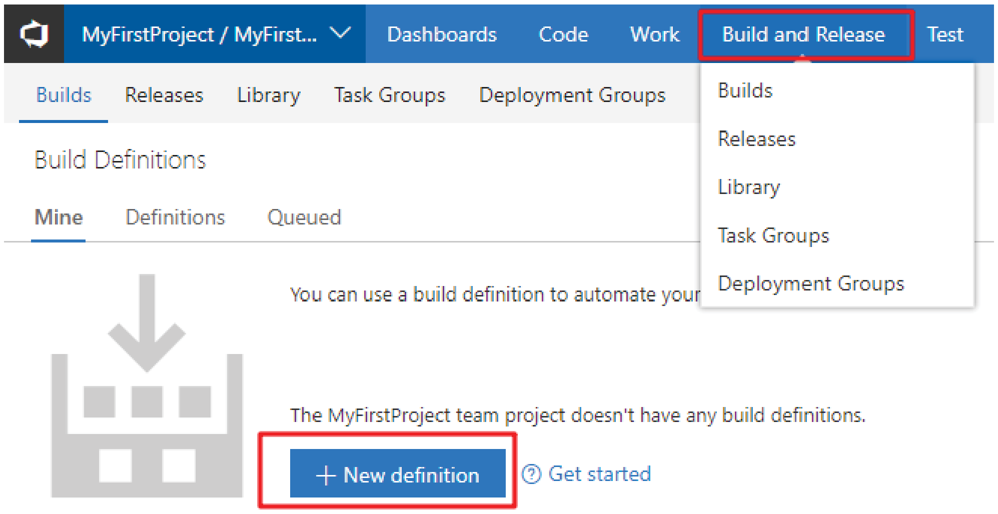


| Name | Last change | Commits | |
|---|---|---|---|
| .vscode | 5 minutes ago | d84dc848 | init Xin Shi |
| modules/FilterModule | 5 minutes ago | d84dc848 | init Xin Shi |
| tests/FilterModuleTest | 5 minutes ago | d84dc848 | init Xin Shi |
| .gitignore | 5 minutes ago | d84dc848 | init Xin Shi |
| deployment.template.json | 5 minutes ago | d84dc848 | init Xin Shi |

2. Visit Azure IoT Edge For VSTS on VSTS Marketplace. Click **Get it free** and follow the wizard to install this extension to your VSTS account or download to your TFS.


Visual Studio | Marketplace
Azure IoT Edge For VSTS
Account
Done
Select a Visual Studio Team Services account
Install
For Team Foundation Server
Download

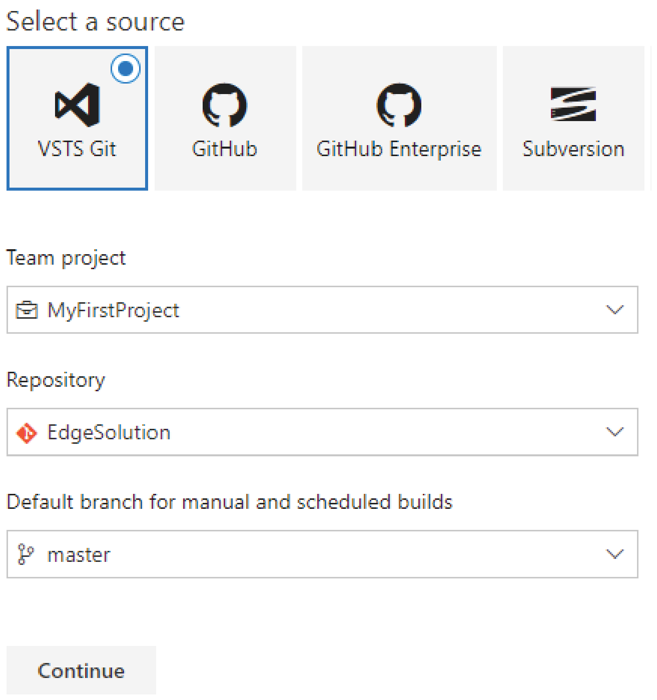
3. In your VSTS, open the **Build & Release** hub and, in the **Builds** tab, choose **+ New definition**. Or, if you already have build definitions, choose the **+ New** button.



You can use a build definition to automate your

The MyFirstProject team project doesn't have any build definitions.

4. If prompted, select the **VSTS Git** source type; then select the project, repository, and branch where your code is located. Choose **Continue**.


Select a source

VSTS Git | GitHub | GitHub Enterprise | Subversion

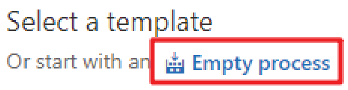
Team project: MyFirstProject

Repository: EdgeSolution

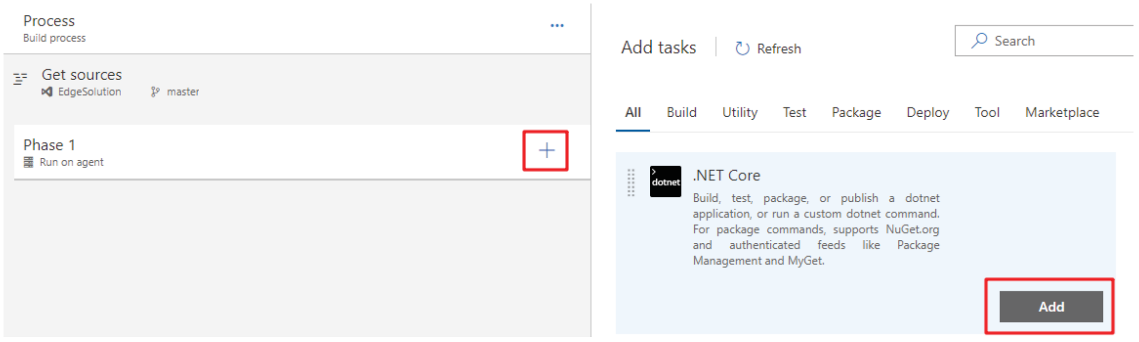
Default branch for manual and scheduled builds: master

Continue

5. In **Select a template** window, choose **start with an Empty process**.



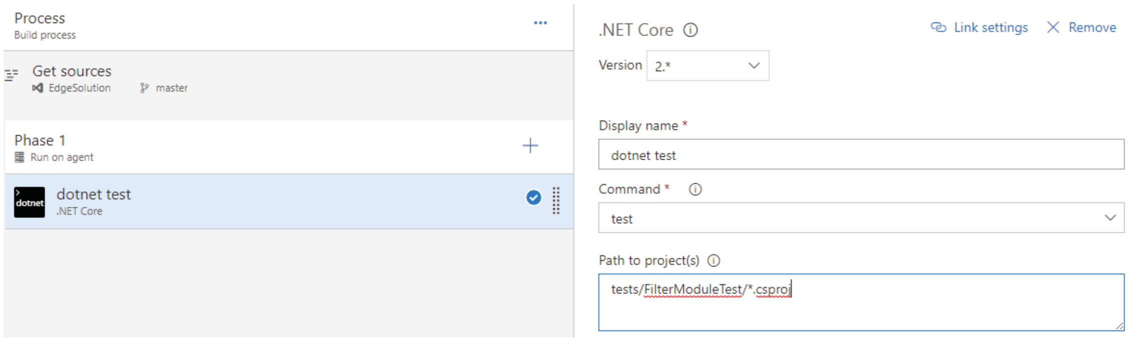
6. Click **+** on the right side of **Phase 1** to add a task to the phase. Then search and select **.Net Core**, and click **Add** to add this task to the phase.



Build, test, package, or publish a dotnet application, or run a custom dotnet command. For package commands, supports NuGet.org and authenticated feeds like Package Management and MyGet.

7. Update the **Display name** to **dotnet test**, and in the **Command** dropdown list, select **test**. Add below path to the **Path to project(s)**.

```
tests/FilterModuleTest/*.csproj
```



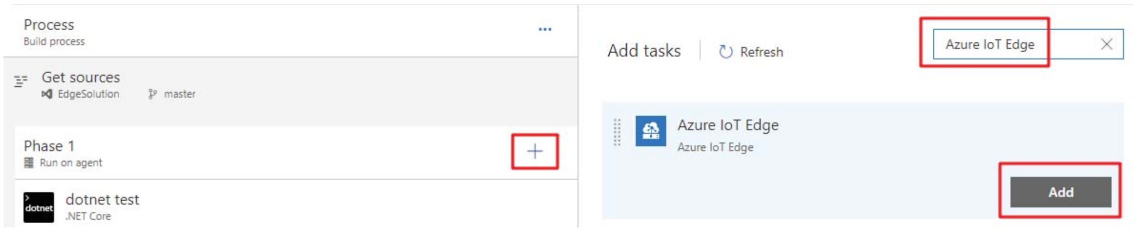
Version: 2.*

Display name *: dotnet test

Command *: test

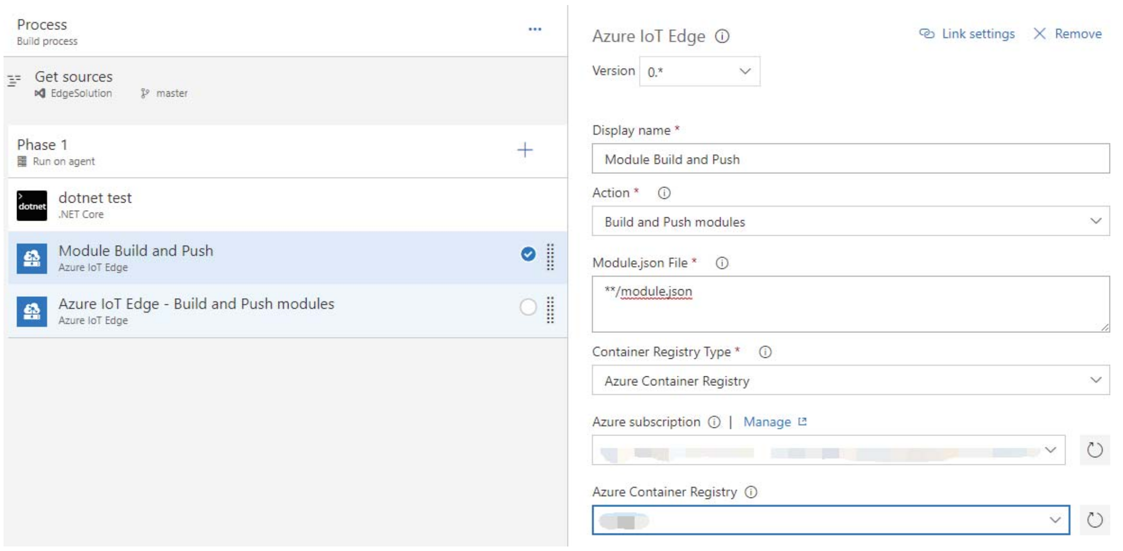
Path to project(s): tests/FilterModuleTest/*.csproj

8. Click **+** on the right side of **Phase 1** to add a task to the phase. Then search and select **Azure IoT Edge**, and click **Add** button **twice** to add these tasks to the phase.



9. In the first Azure IoT Edge task, update the **Display name** to **Module Build and Push**, and in the **Action** dropdown list, select **Build and Push**. In the **Module.json File** textbox, add below path to it. Then choose **Container Registry Type**, make sure you configure and select the same registry in your code. This task will build and push all your modules in the solution and publish to the container registry you specified.

```
**/module.json
```



Version: 0.*

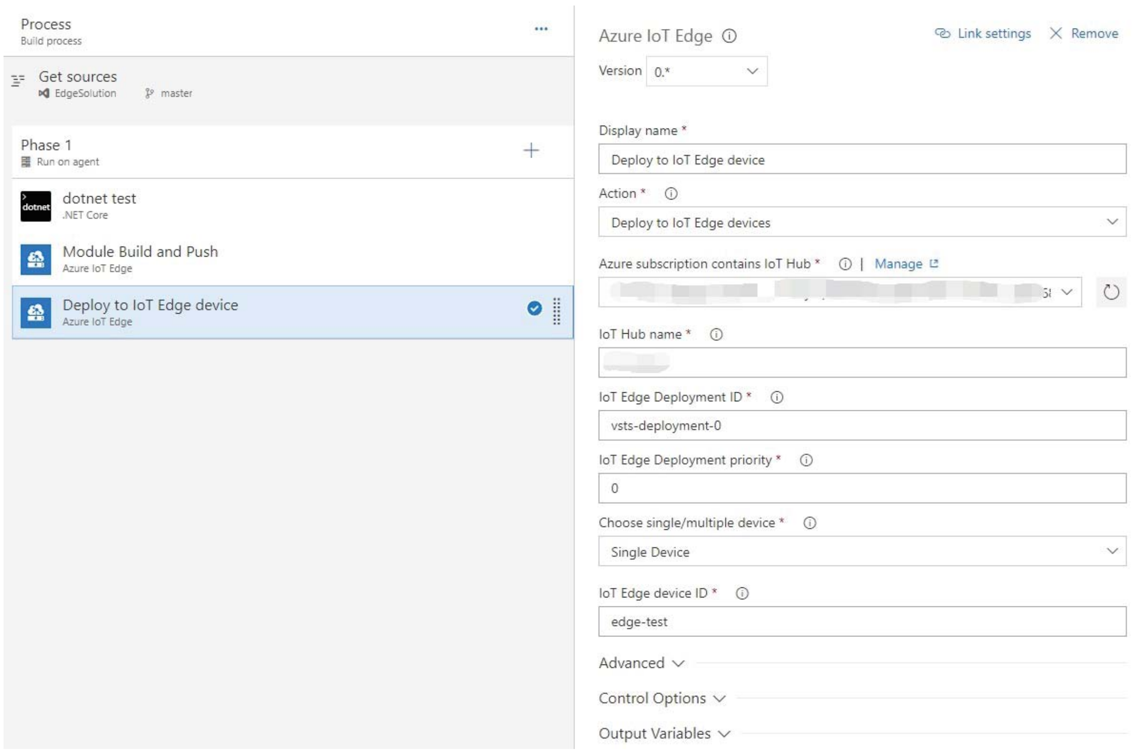
Display name *: Module Build and Push

Action *: Build and Push modules

Module.json File *: **/module.json

Container Registry Type *: Azure Container Registry

10. In the second Azure IoT Edge task, update the **Display name** to **Deploy to IoT Edge device**, and in the **Action** dropdown list, select **Deploy to IoT Edge device**. Select your Azure subscription and input your IoT Hub name. You can specify an IoT Edge deployment ID and the deployment priority. You can also choose to deploy to single or multiple devices. If you are deploying to multiple devices, you need to specify the device target condition. For example, if you want to use device Tags as the condition, you need to update your corresponding devices Tags before the deployment.



Process
Build process

Get sources
EdgeSolution master

Phase 1
Run on agent

dotnet test
.NET Core

Module Build and Push
Azure IoT Edge

Deploy to IoT Edge device
Azure IoT Edge

Azure IoT Edge

Link settings  Remove

Version: 0.*

Display name *: Deploy to IoT Edge device

Action *: Deploy to IoT Edge devices

Azure subscription contains IoT Hub * | Manage: [redacted]

IoT Hub name *: [redacted]

IoT Edge Deployment ID *: vsts-deployment-0

IoT Edge Deployment priority *: 0

Choose single/multiple device *: Single Device

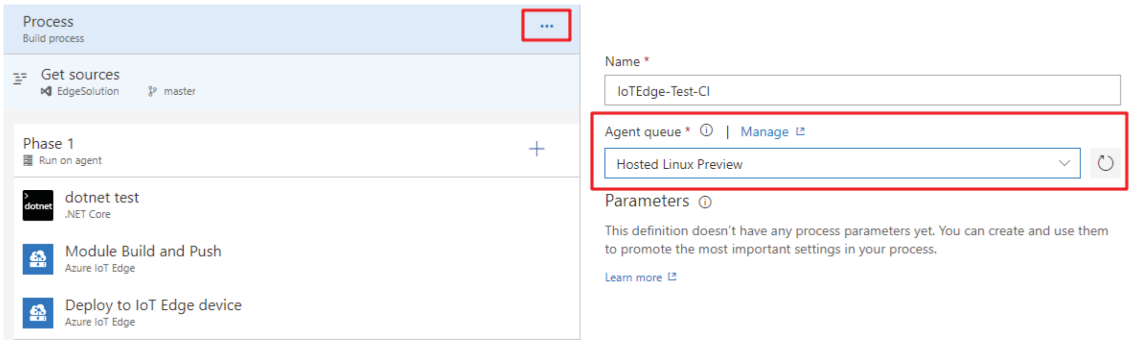
IoT Edge device ID *: edge-test

Advanced

Control Options

Output Variables

11. Click the **Process** and make sure your **Agent queue** is **Hosted Linux Preview**.



Process
Build process

Get sources
EdgeSolution master

Phase 1
Run on agent

dotnet test
.NET Core

Module Build and Push
Azure IoT Edge

Deploy to IoT Edge device
Azure IoT Edge

Name *: IoTEdge-Test-CI

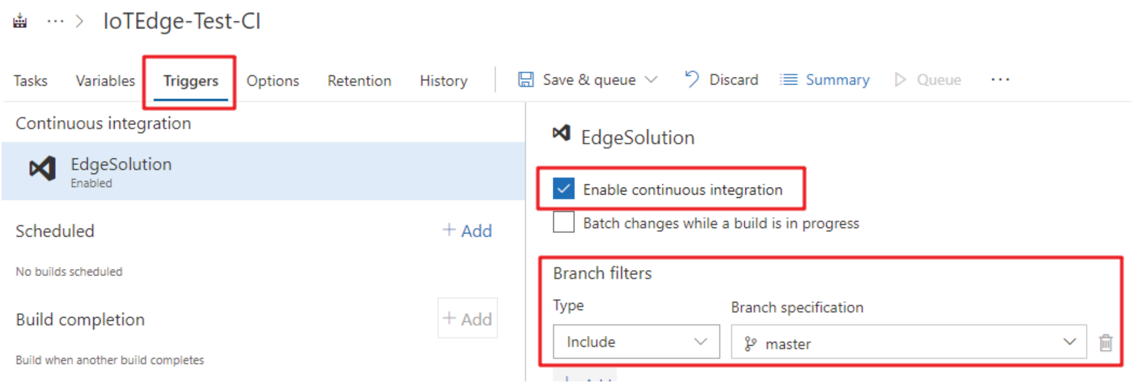
Agent queue * | Manage: Hosted Linux Preview

Parameters

This definition doesn't have any process parameters yet. You can create and use them to promote the most important settings in your process.

Learn more

12. Open the **Triggers** tab and turn on the **Continuous integration** trigger. Make sure the branch containing your code is included.



IoTEdge-Test-CI

Tasks  Variables  Triggers  Options  Retention  History  Save & queue  Discard  Summary  Queue

Continuous integration

EdgeSolution
Enabled

Scheduled  + Add

No builds scheduled

Build completion  + Add

Build when another build completes

EdgeSolution

[✓] Enable continuous integration

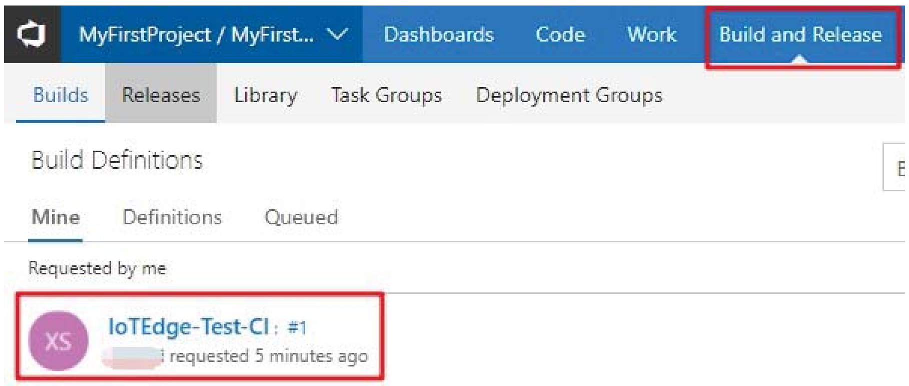
[ ] Batch changes while a build is in progress

Branch filters

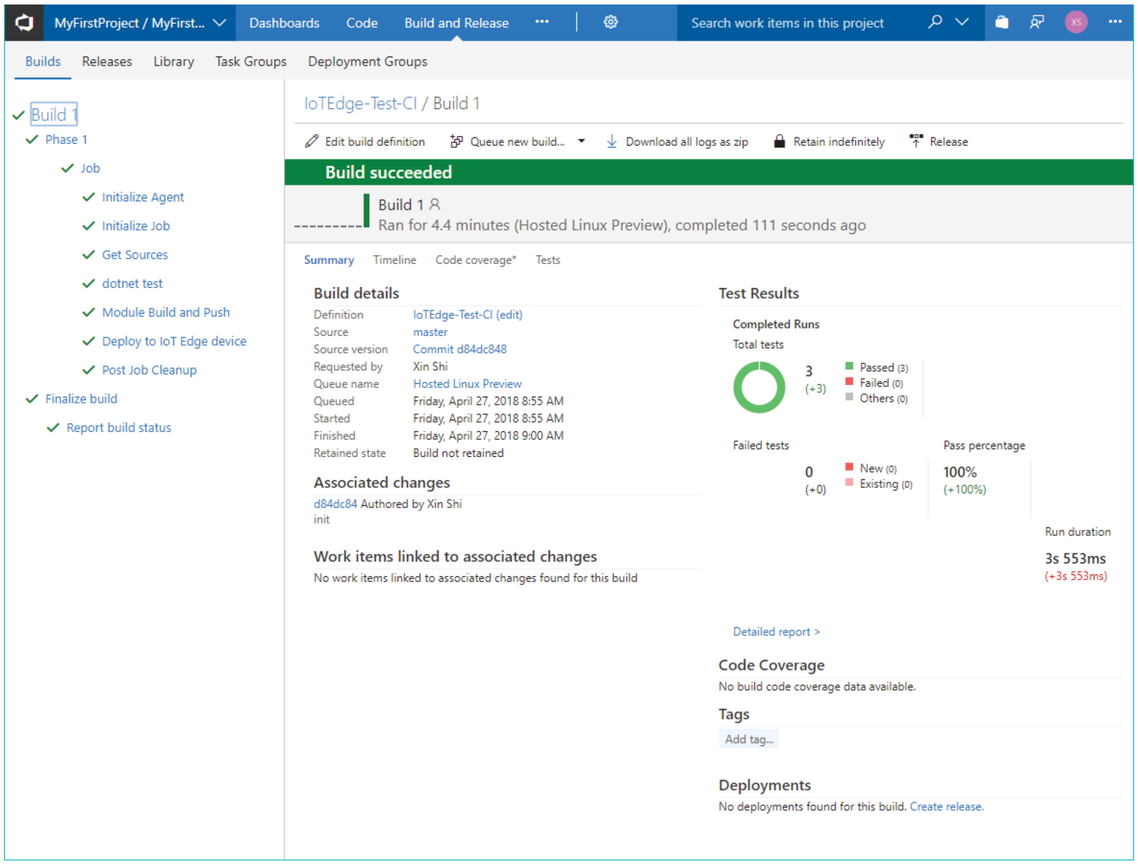
| Type | Branch specification |
|---|---|
| Include | master |

13. Save the new build definition and queue a new build. Click the **Save & queue** button.

14. Choose the link to the build in the message bar that appears. Or go to build definition to see the latest queued build job.



15. After the build has finished, you see the summary for each task and the results in the live log file.



Build succeeded

Build 1

Ran for 4.4 minutes (Hosted Linux Preview), completed 111 seconds ago

Build details

| | |
|---|---|
| Definition | IoTEdge-Test-CI (edit) |
| Source | master |
| Source version | Commit d84dc848 |
| Requested by | Xin Shi |
| Queue name | Hosted Linux Preview |
| Queued | Friday, April 27, 2018 8:55 AM |
| Started | Friday, April 27, 2018 8:55 AM |
| Finished | Friday, April 27, 2018 9:00 AM |
| Retained state | Build not retained |

Associated changes

d84dc84 Authored by Xin Shi
init

Work items linked to associated changes

No work items linked to associated changes found for this build

Test Results

Completed Runs

Total tests 3 (+3) 🟩 Passed (3) 🟥 Failed (0) Others (0)

Failed tests 0 (+0) 🟥 New (0) Existing (0)

Pass percentage 100% (+100%)

Run duration 3s 553ms (+3s 553ms)

Detailed report >

Code Coverage

No build code coverage data available.

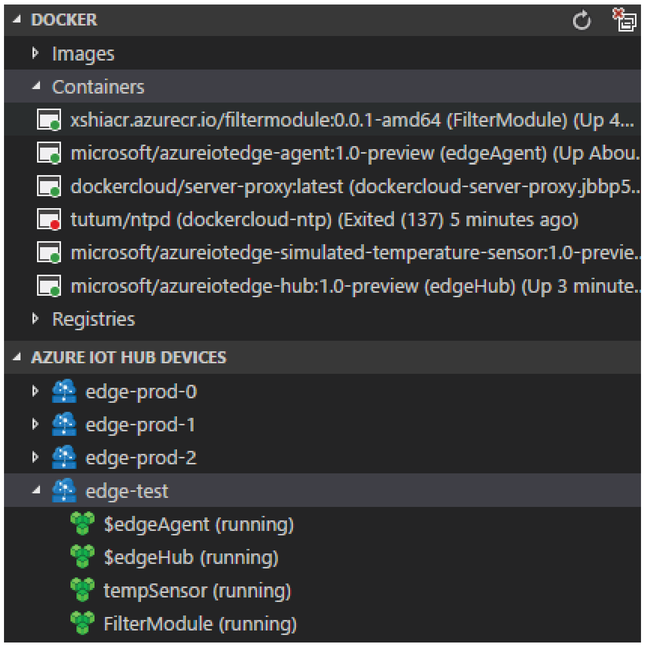
Tags

Add tag...

Deployments

No deployments found for this build. Create release.

16. You can go back to VS Code and check the IoT Hub device explorer. The Edge device with the module should start running (Make sure you've added registry credentials to Edge runtime).



```
xshiacr.azurecr.io/filtermodule:0.0.1-amd64 (FilterModule) (Up 4...
microsoft/azureiotedge-agent:1.0-preview (edgeAgent) (Up Abou...
dockercloud/server-proxy:latest (dockercloud-server-proxy.jbbp5...
tutum/ntpd (dockercloud-ntp) (Exited (137) 5 minutes ago)
microsoft/azureiotedge-simulated-temperature-sensor:1.0-previe...
microsoft/azureiotedge-hub:1.0-preview (edgeHub) (Up 3 minute...
```

```
edge-prod-0
edge-prod-1
edge-prod-2
edge-test
  $edgeAgent (running)
  $edgeHub (running)
  tempSensor (running)
  FilterModule (running)
```

## Continuous deployment to IoT Edge devices

To enable continuous deployment, basically you need to set up CI jobs with proper IoT Edge devices, enabling the **Triggers** for your branches in your project. In a classic DevOps practice, a project contains two main branches. The master branch should be the stable version of the code, and the develop branch contains the latest code changes. Every developer in the team should fork develop branch to his or her own feature branch when starting updating the code, which means all commits happens on feature branches off the develop branch. And every pushed commit should be tested via the CI system. After fully tested the code locally, the feature branch should be merged to the develop branch via a pull request. When the code on developer branch is tested via CI system, it can be merged to master branch via a pull request.

So, when deploying to IoT Edge devices, there are three main environments.

- On feature branch, you can use simulated IoT Edge device on your development machine or deploy to a physical IoT Edge device.
- On develop branch, you should deploy to a physical IoT Edge device.
- On master branch, the target IoT Edge devices should be the production devices.

## Next steps

This tutorial demonstrates how you can use the continuous integration and continuous deployment features of VSTS or TFS.

- Understand the IoT Edge deployment in Understand IoT Edge deployments for single devices or at scale
- Walk through the steps to create, update, or delete a deployment in [Deploy and monitor IoT Edge modules at scale][how-to-deploy-monitor.md].

# Store data at the edge with SQL Server databases

4/25/2018 • 8 min to read • Edit Online

Use Azure IoT Edge devices to store the data that is generated at the edge. Devices with intermittent internet connections can maintain their own databases and report changes back to the cloud only when connected. Devices that have been programmed to send only critical data to the cloud can save the rest of the data for regular bulk uploads. Once in the cloud, the structured data can be shared with other Azure services, for instance to build a machine learning model.

This article provides instructions for deploying a SQL Server database to an IoT Edge device. Azure Functions, running on the IoT Edge device, structures the incoming data then sends it to the database. The steps in this article can also be applied to other databases that work in containers, like MySQL or PostgreSQL.

## Prerequisites

Before you start the instructions in this article, you should complete the following tutorials:

- Deploy Azure IoT Edge on a simulated device in Windows or Linux
- Deploy Azure Function as an IoT Edge module

The following articles aren't required to successfully complete this tutorial, but may provide helpful context:

- Run the SQL Server 2017 container image with Docker
- Use Visual Studio Code to develop and deploy Azure Functions to Azure IoT Edge

After you complete the required tutorials, you should have all the required prerequisites ready on your machine:

- An active Azure IoT hub.
- An IoT Edge device with at least 2-GB RAM and a 2-GB disk drive.
- Visual Studio Code.
- Azure IoT Edge extension for Visual Studio Code.
- C# for Visual Studio Code (powered by OmniSharp) extension.
- Docker
- .NET Core 2.0 SDK.
- Python 2.7
- IoT Edge control script
- AzureIoTEdgeFunction template (`dotnet new -i Microsoft.Azure.IoT.Edge.Function`)
- An active IoT hub with at least an IoT Edge device.

Both Windows and Linux containers on x64 processor architectures work for this tutorial. SQL Server does not support ARM processors.

## Deploy a SQL Server container

In this section, you add an MS-SQL database to your simulated IoT Edge device. Use the SQL Server 2017 docker container image, available as a Windows container and as a Linux container.

### Deploy SQL Server 2017

By default, the code in this section creates a container with the free Developer edition of SQL Server 2017. If you want to run production editions instead, see Run production container images for detailed information.

In step 3, you add create options to the SQL Server container, which are important for establishing environment variables and persistant storage. The configured environment variables accept the End-User License Agreement, and define a password. The persistant storage is configured using mounts. Mounts create the SQL Server 2017 container with a *sqlvolume* volume container attached so that your data persists even if the container is deleted.

1. Open the `deployment.json` file in Visual Studio Code.
2. Replace the **modules** section of the file with the following code:

```
"modules": {
       "filterFunction": {
         "version": "1.0",
         "type": "docker",
         "status": "running",
         "restartPolicy": "always",
         "settings": {
           "image": "<docker registry address>/filterfunction:latest",
           "createOptions": "{}"
         }
       },
       "tempSensor": {
         "version": "1.0",
         "type": "docker",
         "status": "running",
         "restartPolicy": "always",
         "settings": {
           "image": "microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview",
           "createOptions": "{}"
         }
       },
       "sql": {
         "version": "1.0",
         "type": "docker",
         "status": "running",
         "restartPolicy": "always",
         "settings": {
           "image": "",
           "createOptions": ""
          }
       }
     }
```

3. Replace the `<docker registry address>` with the address filled in at the completed tutorial Deploy Azure Function as an IoT Edge module - preview

**NOTE**

The container registry address is the same as the login server that you copied from your registry. It should be in the form of `<your container registry name>.azurecr.io`

4. Depending on the operating system that you're running, update the settings for the SQL module with the following code:

- Windows:

```
"image": "microsoft/mssql-server-windows-developer",
"createOptions": "{\"Env\":
[\"ACCEPT_EULA=Y\",\"MSSQL_SA_PASSWORD=Strong!Passw0rd\"],\"HostConfig\": {\"Mounts\":
[{\"Target\": \"C:\\\\mssql\",\"Source\": \"sqlVolume\",\"Type\": \"volume\"}],\"PortBindings\":
{\"1433/tcp\": [{\"HostPort\": \"1401\"}]}}}"
```

- Linux:

  ```
  "image": "microsoft/mssql-server-linux:2017-latest",
  "createOptions": "{\"Env\":
  [\"ACCEPT_EULA=Y\",\"MSSQL_SA_PASSWORD=Strong!Passw0rd\"],\"HostConfig\": {\"Mounts\":
  [{\"Target\": \"/var/opt/mssql\",\"Source\": \"sqlVolume\",\"Type\":
  \"volume\"}],\"PortBindings\": {\"1433/tcp\": [{\"HostPort\": \"1401\"}]}}}"
  ```

5. Save the file.
6. In the VS Code Command Palette, select **Edge: Create deployment for Edge device**.
7. Select your IoT Edge device ID.
8. Select the `deployment.json` file that you updated. In the output window, you can see corresponding outputs for your deployment.
9. To start your Edge runtime, select **Edge: Start Edge** in the Command Palette.

**TIP**

Any time that you create a SQL Server container in a production environment, you should change the default system administrator password.

## Create the SQL database

This section guides you through setting up the SQL database to store the temperature data received from the sensors connected to the IoT Edge device. If you're using a simulated device, this data comes from the *tempSensor* container.

In a command-line tool, connect to your database:

- Windows container

  ```
  docker exec -it sql cmd
  ```

- Linux container

  ```
  docker exec -it sql bash
  ```

Open the SQL command tool:

- Windows container

  ```
  sqlcmd -S localhost -U SA -P 'Strong!Passw0rd'
  ```

- Linux container

  ```
  /opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P 'Strong!Passw0rd'
  ```

Create your database:

- Windows container

```
CREATE DATABASE MeasurementsDB
ON
(NAME = MeasurementsDB, FILENAME = 'C:\mssql\measurementsdb.mdf')
GO
```

- Linux container

```
CREATE DATABASE MeasurementsDB
ON
(NAME = MeasurementsDB, FILENAME = '/var/opt/mssql/measurementsdb.mdf')
GO
```

Define your table:

```
CREATE TABLE MeasurementsDB.dbo.TemperatureMeasurements (measurementTime DATETIME2, location NVARCHAR(50),
temperature FLOAT)
GO
```

You can customize your SQL Server docker file to automatically set up your SQL Server to be deployed on multiple IoT Edge devices. For more information, see the Microsoft SQL Server container demo project.

## Understand the SQL connection

In other tutorials, we use routes to allow containers to communicate while remaining isolated from each other. When you work with a SQL Server database, though, a closer relationship is necessary.

IoT Edge automatically builds a bridge (Linux) or NAT (Windows) network when it starts. The network is called **azure-iot-edge**. If you ever need to debug this connection, you can look up its properties in the command line:

- Windows

```
docker network inspect azure-iot-edge
```

- Linux

```
sudo docker network inspect azure-iot-edge
```

IoT Edge can also resolve the DNS of a container name through docker, so you don't need to refer to your SQL Server database by its IP address.

As an example, here is the connection string that we use in the next section:

```
Data Source=tcp:sql,1433;Initial Catalog=MeasurementsDB;User
Id=SA;Password=Strong!Passw0rd;TrustServerCertificate=False;Connection Timeout=30;
```

You can see that the connection string references the container by its name, **sql**. If you changed the module name to be something else, update this connection string as well. Otherwise, continue on to the next section.

## Update your Azure Function

To send the data to your database, update the FilterFunction Azure Function that you made in the previous tutorial. Change this file so that it structures the data received by your sensors then stores it in a SQL table.

1. In Visual Studio Code, open your FilterFunction folder.
2. Replace the run.csx file with the following code that includes the SQL connection string from the previous section:

```
#r "Microsoft.Azure.Devices.Client"
#r "Newtonsoft.Json"
#r "System.Data.SqlClient"

using System.IO;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
using Sql = System.Data.SqlClient;
using System.Threading.Tasks;

// Filter messages based on the temperature value in the body of the message and the temperature
threshold value.
public static async Task Run(Message messageReceived, IAsyncCollector<Message> output, TraceWriter log)
{
    const int temperatureThreshold = 25;
    byte[] messageBytes = messageReceived.GetBytes();
    var messageString = System.Text.Encoding.UTF8.GetString(messageBytes);

    if (!string.IsNullOrEmpty(messageString))
    {
        // Get the body of the message and deserialize it
        var messageBody = JsonConvert.DeserializeObject<MessageBody>(messageString);

        //Store the data in SQL db
        const string str = "Data Source=tcp:sql,1433;Initial Catalog=MeasurementsDB;User
Id=SA;Password=Strong!Passw0rd;TrustServerCertificate=False;Connection Timeout=30;";
        using (Sql.SqlConnection conn = new Sql.SqlConnection(str))
        {
        conn.Open();
        var insertMachineTemperature = "INSERT INTO MeasurementsDB.dbo.TemperatureMeasurements VALUES
(CONVERT(DATETIME2,'" + messageBody.timeCreated + "', 127), 'machine', " +
messageBody.machine.temperature + ");";
        var insertAmbientTemperature = "INSERT INTO MeasurementsDB.dbo.TemperatureMeasurements VALUES
(CONVERT(DATETIME2,'" + messageBody.timeCreated + "', 127), 'ambient', " +
messageBody.ambient.temperature + ");";
            using (Sql.SqlCommand cmd = new Sql.SqlCommand(insertMachineTemperature + "\n" +
insertAmbientTemperature, conn))
            {
            //Execute the command and log the # rows affected.
            var rows = await cmd.ExecuteNonQueryAsync();
            log.Info($"{rows} rows were updated");
            }
        }

        if (messageBody != null && messageBody.machine.temperature > temperatureThreshold)
        {
            // Send the message to the output as the temperature value is greater than the threshold
            var filteredMessage = new Message(messageBytes);
            // Copy the properties of the original message into the new Message object
            foreach (KeyValuePair<string, string> prop in messageReceived.Properties)
            {
                filteredMessage.Properties.Add(prop.Key, prop.Value);
            }
            // Add a new property to the message to indicate it is an alert
            filteredMessage.Properties.Add("MessageType", "Alert");
            // Send the message
            await output.AddAsync(filteredMessage);
            log.Info("Received and transferred a message with temperature above the threshold");
        }
    }
}

//Define the expected schema for the body of incoming messages
```

```
class MessageBody
{
    public Machine machine {get;set;}
    public Ambient ambient {get; set;}
    public string timeCreated {get; set;}
}
class Machine
{
   public double temperature {get; set;}
   public double pressure {get; set;}
}
class Ambient
{
   public double temperature {get; set;}
   public int humidity {get; set;}
}
```

## Update your container image

To apply the changes that you've made, update your container image, publish it, and restart IoT Edge.

1. In Visual Studio Code, expand the **Docker** folder.
2. Based on the platform you're using, expand either the **windows-nano** or **linux-x64** folder.
3. Right-click the **Dockerfile** file and select **Build IoT Edge module Docker image**.
4. Navigate to the **FilterFunction** project folder and click **Select folder as EXE_DIR**.
5. In the pop-up text box at the top of the VS Code window, enter the image name. For example, `<your container registry address>/filterfunction:latest`. If you are deploying to a local registry, the name should be `<localhost:5000/filterfunction:latest>`.
6. In the VS Code command palette, select **Edge: Push IoT Edge module Docker image**.
7. In the pop-up text box, enter the same image name.
8. In the VS Code command palette, select **Edge: Restart Edge**.

## View the local data

Once your containers restart, the data received from the temperature sensors is stored in a local SQL Server 2017 database on your IoT Edge device.

In a command-line tool, connect to your database:

- Windows container

  ```
  docker exec -it sql cmd
  ```

- Linux container

  ```
  docker exec -it sql bash
  ```

Open the SQL command tool:

- Windows container

  ```
  sqlcmd -S localhost -U SA -P 'Strong!Passw0rd'
  ```

- Linux container

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P 'Strong!Passw0rd'
```

View your data:

```
SELECT * FROM MeasurementsDB.dbo.TemperatureMeasurements
GO
```

## Next steps

- Learn how to configure SQL Server 2017 container images on Docker.
- Visit the mssql-docker GitHub repository for resources, feedback, and known issues.

# Common issues and resolutions for Azure IoT Edge

4/9/2018 • 4 min to read • Edit Online

If you experience issues running Azure IoT Edge in your environment, use this article as a guide for troubleshooting and resolution.

## Standard diagnostic steps

When you encounter an issue, learn more about the state of your IoT Edge device by reviewing the container logs and messages that pass to and from the device. Use the commands and tools in this section to gather information.

- Look at the logs of the docker containers to detect issues. Start with your deployed containers, then look at the containers that make up the IoT Edge runtime: Edge Agent and Edge Hub. The Edge Agent logs typically provide info on the lifecycle of each container. The Edge Hub logs provide info on messaging and routing.

  ```
  docker logs <container name>
  ```

- View the messages going through the Edge Hub, and gather insights on device properties updates with verbose logs from the runtime containers.

  ```
  iotedgectl setup --connection-string "{device connection string}" --runtime-log-level debug
  ```

- View verbose logs from iotedgectl commands:

  ```
  iotedgectl --verbose DEBUG <command>
  ```

- If you experience connectivity issues, inspect your edge device environment variables like your device connection string:

  ```
  docker exec edgeAgent printenv
  ```

You can also check the messages being sent between IoT Hub and the IoT Edge devices. View these messages by using the Azure IoT Toolkit extension for Visual Studio Code. For more guidance, see Handy tool when you develop with Azure IoT.

After investigating the logs and messages for information, you can also try restarting the Azure IoT Edge runtime:

```
iotedgectl restart
```

## Edge Agent stops after about a minute

The Edge Agent starts and runs successfully for about a minute, then stops. The logs indicate that the Edge Agent is attempting to connect to IoT Hub over AMQP, and then approximately 30 seconds later attempt to connect using AMQP over websocket. When that fails, the Edge Agent exits.

Example Edge Agent logs:

```
2017-11-28 18:46:19 [INF] - Starting module management agent.
2017-11-28 18:46:19 [INF] - Version - 1.0.7516610 (03c94f85d0833a861a43c669842f0817924911d5)
2017-11-28 18:46:19 [INF] - Edge agent attempting to connect to IoT Hub via AMQP...
2017-11-28 18:46:49 [INF] - Edge agent attempting to connect to IoT Hub via AMQP over WebSocket...
```

**Root cause**

A networking configuration on the host network is preventing the Edge Agent from reaching the network. The agent attempts to connect over AMQP (port 5671) first. If this fails, it tries websockets (port 443).

The IoT Edge runtime sets up a network for each of the modules to communicate on. On Linux, this network is a bridge network. On Windows, it uses NAT. This issue is more common on Windows devices using Windows containers that use the NAT network.

**Resolution**

Ensure that there is a route to the internet for the IP addresses assigned to this bridge/NAT network. Sometimes a VPN configuration on the host overrides the IoT Edge network.

## Edge Hub fails to start

The Edge Hub fails to start, and prints the following message to the logs:

```
One or more errors occurred.
(Docker API responded with status code=InternalServerError, response=
{\"message\":\"driver failed programming external connectivity on endpoint edgeHub
(6a82e5e994bab5187939049684fb64efe07606d2bb8a4cc5655b2a9bad5f8c80):
Error starting userland proxy: Bind for 0.0.0.0:443 failed: port is already allocated\"}\n)
```

**Root cause**

Some other process on the host machine has bound port 443. The Edge Hub maps ports 5671 and 443 for use in gateway scenarios. This port mapping fails if another process has already bound this port.

**Resolution**

Find and stop the process that is using port 443. This process is usually a web server.

## Edge Agent can't access a module's image (403)

A container fails to run, and the Edge Agent logs show a 403 error.

**Root cause**

The Edge Agent doesn't have permissions to access a module's image.

**Resolution**

Try running the `iotedgectl login` command again.

## iotedgectl can't find Docker

The commands `iotedgectl setup` or `iotedgectl start` fail and print the following message to the logs:

```
File "/usr/local/lib/python2.7/dist-packages/edgectl/host/dockerclient.py", line 98, in get_os_type
  info = self._client.info()
File "/usr/local/lib/python2.7/dist-packages/docker/client.py", line 174, in info
  return self.api.info(*args, **kwargs)
File "/usr/local/lib/python2.7/dist-packages/docker/api/daemon.py", line 88, in info
  return self._result(self._get(self._url("/info")), True)
```

### Root cause

iotedgectl can't find Docker, which is a pre-requisite.

### Resolution

Install Docker, make sure that it is running and retry.

## iotedgectl setup fails with an invalid hostname

The command `iotedgectl setup` fails and prints the following message:

```
Error parsing user input data: invalid hostname. Hostname cannot be empty or greater than 64 characters
```

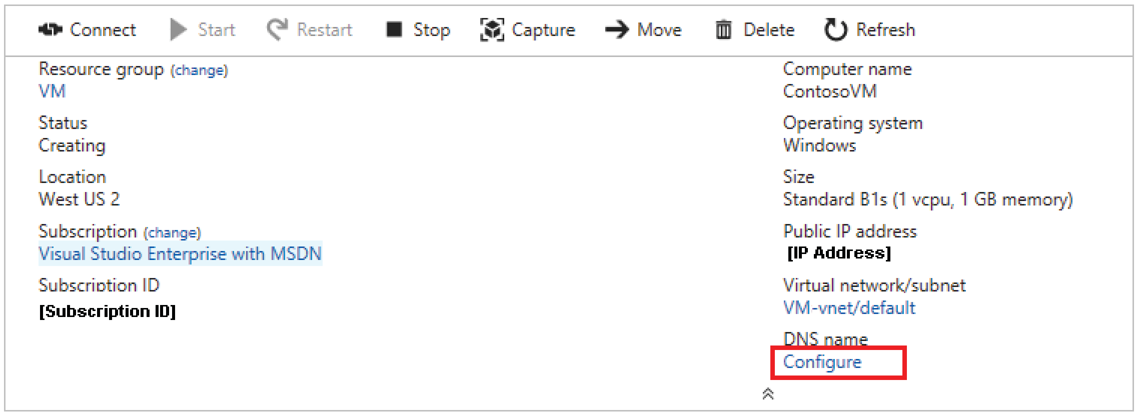
### Root cause

The IoT Edge runtime can only support hostnames that are shorter than 64 characters. This usually isn't an issue for physical machines, but can occur when you set up the runtime on a virtual machine. The automatically generated hostnames for Windows virtual machines hosted in Azure, in particular, tend to be long.

### Resolution

When you see this error, you can resolve it by configuring the DNS name of your virtual machine, and then setting the DNS name as the hostname in the setup command.

1. In the Azure portal, navigate to the overview page of your virtual machine.
2. Select **configure** under DNS name. If your virtual machine already has a DNS name configured, you don't need to configure a new one.



Resource group (change)
VM

Status
Creating

Location
West US 2

Subscription (change)
Visual Studio Enterprise with MSDN

Subscription ID
**[Subscription ID]**

Computer name
ContosoVM

Operating system
Windows

Size
Standard B1s (1 vcpu, 1 GB memory)

Public IP address
**[IP Address]**

Virtual network/subnet
VM-vnet/default

DNS name
Configure

3. Provide a value for **DNS name label** and select **Save**.
4. Copy the new DNS name, which should be in the format **<DNSnamelabel>.<vmlocation>.cloudapp.azure.com**.
5. Inside the virtual machine, use the following command to set up the IoT Edge runtime with your DNS name:

```
iotedgectl setup --connection-string "<connection string>" --nopass --edge-hostname "<DNS name>"
```

## Next steps

Do you think that you found a bug in the IoT Edge platform? Please, submit an issue so that we can continue to improve.

# Understand Azure IoT Edge modules - preview

2/15/2018 • 3 min to read • Edit Online

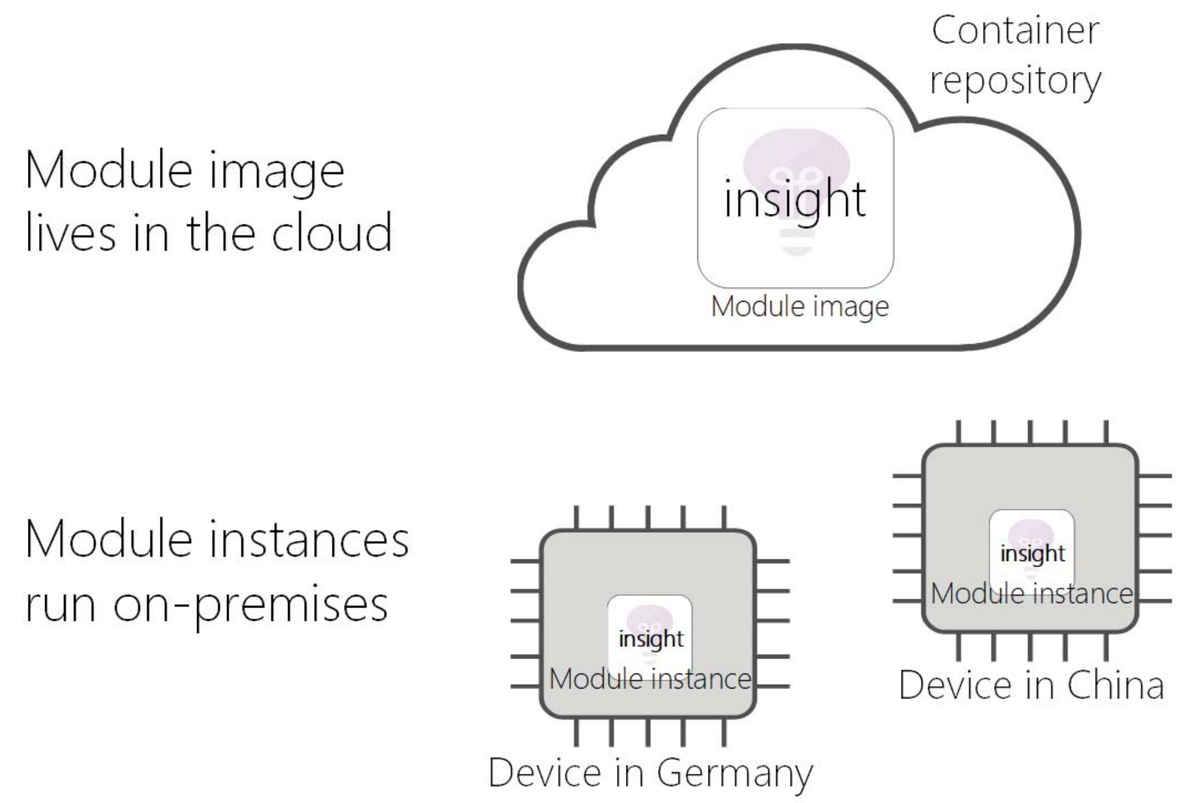
Azure IoT Edge lets you deploy and manage business logic on the edge in the form of *modules*. Azure IoT Edge modules are the smallest unit of computation managed by IoT Edge, and can contain Azure services (such as Azure Stream Analytics) or your own solution-specific code. To understand how modules are developed, deployed, and maintained, it helps to think of four conceptual pieces that make up a module:

- A **module image** is a package containing the software that defines a module.
- A **module instance** is the specific unit of computation running the module image on an IoT Edge device. The module instance is started by the IoT Edge runtime.
- A **module identity** is a piece of information (including security credentials) stored in IoT Hub, that is associated to each module instance.
- A **module twin** is a JSON document stored in IoT Hub, that contains state information for a module instance, including metadata, configurations, and conditions.

## Module images and instances

IoT Edge module images contain applications that take advantage of the management, security, and communication features of the IoT Edge runtime. You can develop your own module images, or export one from a supported Azure service, such as Azure Stream Analytics. The images exist in the cloud and they can be updated, changed, and deployed in different solutions. For instance, a module that uses machine learning to predict production line output exists as a separate image than a module that uses computer vision to control a drone.

Each time a module image is deployed to a device and started by the IoT Edge runtime, a new instance of that module is created. Two devices in different parts of the world could use the same module image; however each would have their own module instance when the module is started on the device.

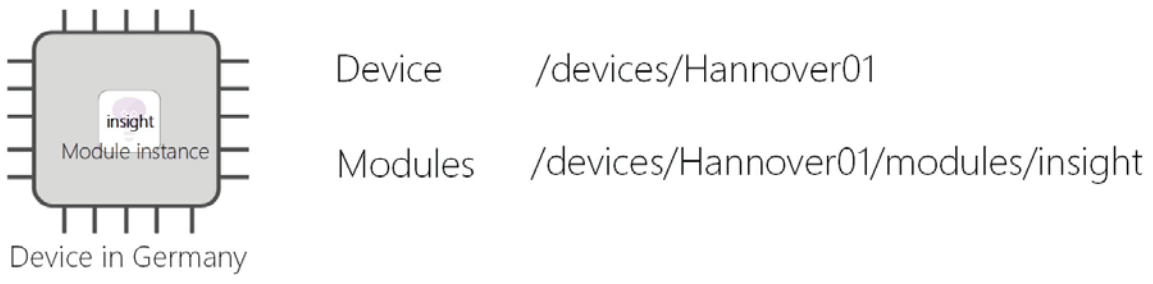


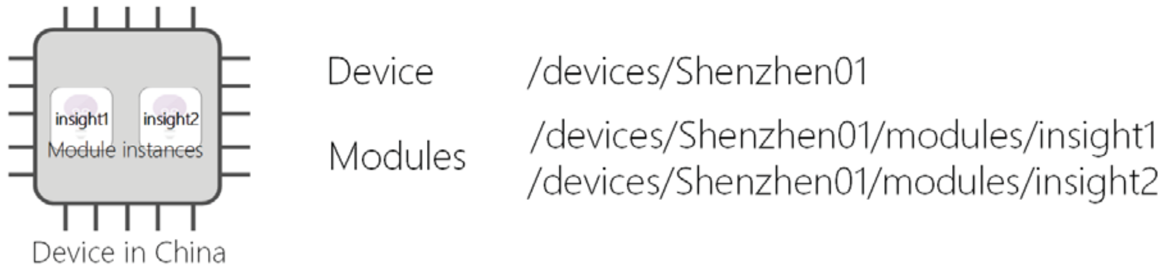
In implementation, modules images exist as container images in a repository, and module instances are containers on devices. As use cases for Azure IoT Edge grow, new types of module images and instances will be created. For example, resource constrained devices cannot run containers so may require module images that exist as dynamic link libraries and instances that are executables.

## Module identities

When a new module instance is created by the IoT Edge runtime, the instance is associated with a corresponding module identity. The module identity is stored in IoT Hub, and is employed as the addressing and security scope for all local and cloud communications for that specific module instance. The identity associated with a module instance depends on the identity of the device on which the instance is running and the name you provide to that module in your solution. For instance, if you call `insight` a module that uses an Azure Stream Analytics, and you deploy it on a device called `Hannover01`, the IoT Edge runtime creates a corresponding module identity called `/devices/Hannover01/modules/insight`.

Clearly, in scenarios when you need to deploy one module image multiple times on the same device, you can deploy the same image multiple times with different names.





## Module twins

Each module instance also has a corresponding module twin that you can use to configure the module instance. The instance and the twin are associated with each other through the module identity.

A module twin is a JSON document that stores module information and configuration properties. This concept parallels the device twin concept from IoT Hub. The structure of a module twin is exactly the same as a device twin. The APIs used to interact with both types of twins are also the same. The only difference between the two is the identity used to instantiate the client SDK.

```
// Create a DeviceClient object. This DeviceClient will act on behalf of a
// module since it is created with a module’s connection string instead
// of a device connection string.
DeviceClient client = new DeviceClient.CreateFromConnectionString(moduleConnectionString, settings);
await client.OpenAsync();

// Get the model twin
Twin twin = await client.GetTwinAsync();
```

## Next steps

- Understand the Azure IoT Edge runtime and its architecture

# Understand the requirements and tools for developing IoT Edge modules - preview

11/15/2017 • 3 min to read • Edit Online

This article explains what functionalities are available when writing applications that run as IoT Edge module, and how to take advantage of them.

## IoT Edge runtime environment

The IoT Edge runtime provides the infrastructure to integrate the functionality of multiple IoT Edge modules and to deploy them onto IoT Edge devices. At a high level, any program can be packaged as an IoT Edge module. However, to take full advantage of IoT Edge communication and management functionalities, a program running in a module can connect to the local IoT Edge hub, integrated in the IoT Edge runtime.

## Using the IoT Edge hub

The IoT Edge hub provides two main functionalities: proxy to IoT Hub, and local communications.

### IoT Hub primitives

IoT Hub sees a module instance analogously to a device, in the sense that it:

- it has a module twin, that is distinct and isolated from the device twin and the other module twins of that device;
- it can send device-to-cloud messages;
- it can receive direct methods targeted specifically at its identity.

Currently, a module cannot receive cloud-to-device messages nor use the file upload feature.

When writing a module, you can simply use the Azure IoT Device SDK to connect to the IoT Edge hub and use the above functionality as you would when using IoT Hub with a device application, the only difference being that, from your application back-end, you have to refer to the module identity instead of the device identity.

See Develop and deploy an IoT Edge module to a simulated device for an example of a module application that sends device-to-cloud messages, and uses the module twin.

### Device-to-cloud messages

In order to enable complex processing of device-to-cloud messages, IoT Edge hub provides declarative routing of messages between modules, and between modules and IoT Hub. This allows modules to intercept and process messages sent by other modules and propagate them into complex pipelines. The article Module composition explains how to compose modules into complex pipelines using routes.

An IoT Edge module, differently than a normal IoT Hub device application, can receive device-to-cloud messages that are being proxied by its local IoT Edge hub, in order to process them.

IoT Edge hub propagates the messages to your module based on declarative routes described in the Module composition article. When developing an IoT Edge module, you can receive these messages by setting message handlers, as shown in the tutorial Develop and deploy an IoT Edge module to a simulated device.

In order to simplify the creation of routes, IoT Edge adds the concept of module *input* and *output* endpoints. A module can receive all device-to-cloud messages routed to it without specifying any input, and can send device-to-cloud messages without specifying any output. Using explicit inputs and outputs, though, makes routing rules simpler to understand. See Module composition for more information on routing rules and input and output endpoints for modules.

Finally, device-to-cloud messages handled by the Edge hub are stamped with the following system properties:

| PROPERTY | DESCRIPTION |
|---|---|
| $connectionDeviceId | The device ID of the client that sent the message |
| $connectionModuleId | The module ID of the module that sent the message |
| $inputName | The input that received this message. Can be empty. |
| $outputName | The output used to send the message. Can be empty. |

### Connecting to IoT Edge hub from a module

Connecting to the local IoT Edge hub from a module involves two steps: use the connection string provided by the IoT Edge runtime when your module starts, and make sure your application accepts the certificate presented by the IoT Edge hub on that device.

The connecting string to use is injected by the IoT Edge runtime in the environment variable `EdgeHubConnectionString`. This makes it available to any program that wants to use it.

Analogously, the certificate to use to validate the IoT Edge hub connection is injected by the IoT Edge runtime in a file whose path is available in the environment variable `EdgeModuleCACertificateFile`.

The tutorial Develop and deploy an IoT Edge module to a simulated device shows how to make sure that the certificate is in the machine store in your module application. Clearly, any other method to trust connections using that certificate work.

## Packaging as an image

IoT Edge modules are packaged as Docker images. You can use Docker toolchain directly, or Visual Studio Code as shown in the tutorial Develop and deploy an IoT Edge module to a simulated device.

## Next steps

After you develop a module, learn how to Deploy and monitor IoT Edge modules at scale.

# Understand how IoT Edge modules can be used, configured, and reused - preview

4/25/2018 • 6 min to read • Edit Online

Each IoT Edge device runs at least two modules: $edgeAgent and $edgeHub, which make up the IoT Edge runtime. In addition to those standard two, any IoT Edge device can run multiple modules to perform any number of processes. When you deploy all these modules to a device at once, you need a way to declare which modules are included how they interact with each other.

The *deployment manifest* is a JSON document that describes:

- Which IoT Edge modules have to be deployed, along with their creation and management options.
- The configuration of the Edge hub, which includes how messages flow between modules and eventually to IoT Hub.
- Optionally, the values to set in the desired properties of the module twins, to configure the individual module applications.

All IoT Edge devices need to be configured with a deployment manifest. A newly installed IoT Edge runtime reports an error code until configured with a valid manifest.

In the Azure IoT Edge tutorials, you build a deployment manifest by going through a wizard in the Azure IoT Edge portal. You can also apply a deployment manifest programmatically using REST or the IoT Hub Service SDK. Refer to Deploy and monitor for more information on IoT Edge deployments.

## Create a deployment manifest

At a high level, the deployment manifest configures a module twin's desired properties for IoT Edge modules deployed on an IoT Edge device. Two of these modules are always present: the Edge agent, and the Edge hub.

A deployment manifest that contains only the IoT Edge runtime (agent and hub) is valid.

The manifest follows this structure:

```
{
    "moduleContent": {
        "$edgeAgent": {
            "properties.desired": {
                // desired properties of the Edge agent
                // includes the image URIs of all modules
            }
        },
        "$edgeHub": {
            "properties.desired": {
                // desired properties of the Edge hub
                // includes the routing information between modules, and to IoT Hub
            }
        },
        "{module1}": {  // optional
            "properties.desired": {
                // desired properties of module with id {module1}
            }
        },
        "{module2}": {  // optional
            ...
        },
        ...
    }
}
```

## Configure modules

In addition to establishing the desired properties of any modules that you want to deploy, you need to tell the IoT Edge runtime how to install them. The configuration and management information for all modules goes inside the **$edgeAgent** desired properties. This information includes the configuration parameters for the Edge agent itself.

For a complete list of properties that can or must be included, see Properties of the Edge agent and Edge hub.

The $edgeAgent properties follow this structure:

```
"$edgeAgent": {
    "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
        },
        "systemModules": {
            "edgeAgent": {
                // configuration and management details
            },
            "edgeHub": {
                // configuration and management details
            }
        },
        "modules": {
            "{module1}": { //optional
                // configuration and management details
            },
            "{module2}": { // optional
                // configuration and management details
            }
        }
    }
},
```

## Declare routes

Edge hub provides a way to declaratively route messages between modules, and between modules and IoT Hub. The Edge hub manages all communication, so the route information goes inside the **$edgeHub** desired properties. You can have multiple routes within the same deployment.

Routes are declared in the **$edgeHub** desired properties with the following syntax:

```
"$edgeHub": {
    "properties.desired": {
        "routes": {
            "{route1}": "FROM <source> WHERE <condition> INTO <sink>",
            "{route2}": "FROM <source> WHERE <condition> INTO <sink>"
        },
    }
}
```

Every route needs a source and a sink, but the condition is an optional piece that you can use to filter messages.

### Source

The source specifies where the messages come from. It can be any of the following values:

| SOURCE | DESCRIPTION |
|---|---|
| `/*` | All device-to-cloud messages from any device or module |
| `/messages/*` | Any device-to-cloud message sent by a device or a module through some or no output |
| `/messages/modules/*` | Any device-to-cloud message sent by a module through some or no output |
| `/messages/modules/{moduleId}/*` | Any device-to-cloud message sent by {moduleId} with no output |
| `/messages/modules/{moduleId}/outputs/*` | Any device-to-cloud message sent by {moduleId} with some output |
| `/messages/modules/{moduleId}/outputs/{output}` | Any device-to-cloud message sent by {moduleId} using {output} |

### Condition

The condition is optional in a route declaration. If you want to pass all messages from the sink to the source, just leave out the **WHERE** clause entirely. Or you can use the IoT Hub query language to filter for certain messages or message types that satisfy the condition.

The messages that pass between modules in IoT Edge are formatted the same as the messages that pass between your devices and Azure IoT Hub. All messages are formatted as JSON and have **systemProperties**, **appProperties**, and **body** parameters.

You can build queries around all three parameters with the following syntax:

- System properties: `$<propertyName>` or `{$<propertyName>}`
- Application properties: `<propertyName>`
- Body properties: `$body.<propertyName>`

For examples about how to create queries for message properties, see Device-to-cloud message routes query expressions.

An example that is specific to IoT Edge is when you want to filter for messages that arrived at a gateway device from a leaf device. Messages that come from modules contain a system property called **connectionModuleId**. So if you want to route messages from leaf devices directly to IoT Hub, use the following route to exclude module messages:

```
FROM /messages/* WHERE NOT IS_DEFINED($connectionModuleId) INTO $upstream
```

**Sink**

The sink defines where the messages are sent. It can be any of the following values:

| SINK | DESCRIPTION |
|---|---|
| `$upstream` | Send the message to IoT Hub |
| `BrokeredEndpoint("/modules/{moduleId}/inputs/{input}")` | Send the message to input `{input}` of module `{moduleId}` |

It is important to note that Edge hub provides at-least-once guarantees, which means that messages are stored locally in case a route cannot deliver the message to its sink, for example, the Edge hub cannot connect to IoT Hub, or the target module is not connected.

Edge hub stores the messages up to the time specified in the `storeAndForwardConfiguration.timeToLiveSecs` property of the Edge hub desired properties.

## Define or update desired properties

The deployment manifest can specify desired properties for the module twin of each module deployed to the IoT Edge device. When the desired properties are specified in the deployment manifest, they overwrite any desired properties currently in the module twin.

If you do not specify a module twin's desired properties in the deployment manifest, IoT Hub will not modify the module twin in any way, and you will be able to set the desired properties programmatically.

The same mechanisms that allow you to modify device twins are used to modify module twins. Refer to the device twin developer guide for further information.

## Deployment manifest example

This an example of a deployment manifest JSON document.

```
{
"moduleContent": {
    "$edgeAgent": {
        "properties.desired": {
            "schemaVersion": "1.0",
            "runtime": {
                "type": "docker",
                "settings": {
                    "minDockerVersion": "v1.25",
                    "loggingOptions": ""
                }
            },
            "systemModules": {
                "edgeAgent": {
                    "type": "docker",
                    "settings": {
                    "image": "microsoft/azureiotedge-agent:1.0-preview",
                    "createOptions": ""
                    }
                },
                "edgeHub": {
                    "type": "docker",
                    "status": "running",
                    "restartPolicy": "always",
                    "settings": {
                    "image": "microsoft/azureiotedge-hub:1.0-preview",
                    "createOptions": ""
                    }
                }
            },
            "modules": {
                "tempSensor": {
                    "version": "1.0",
                    "type": "docker",
                    "status": "running",
                    "restartPolicy": "always",
                    "settings": {
                    "image": "microsoft/azureiotedge-simulated-temperature-sensor:1.0-preview",
                    "createOptions": "{}"
                    }
                },
                "filtermodule": {
                    "version": "1.0",
                    "type": "docker",
                    "status": "running",
                    "restartPolicy": "always",
                    "settings": {
                    "image": "myacr.azurecr.io/filtermodule:latest",
                    "createOptions": "{}"
                    }
                }
            }
        }
    },
    "$edgeHub": {
        "properties.desired": {
            "schemaVersion": "1.0",
            "routes": {
                "sensorToFilter": "FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/filtermodule/inputs/input1\")",
                "filterToIoTHub": "FROM /messages/modules/filtermodule/outputs/output1 INTO $upstream"
            },
            "storeAndForwardConfiguration": {
                "timeToLiveSecs": 10
            }
        }
    }
}
}
```

## Next steps

- For a complete list of properties that can or must be included in $edgeAgent and $edgeHub, see Properties of the Edge agent and Edge hub.
- Now that you know how IoT Edge modules are used, Understand the requirements and tools for developing IoT Edge modules.

# Properties of the Edge agent and Edge hub module twins

4/9/2018 • 5 min to read • Edit Online

The Edge agent and Edge hub are two modules that make up the IoT Edge runtime. For more information about what duties each module performs, see Understand the Azure IoT Edge runtime and its architecture.

This article provides the desired properties and reported properties of the runtime module twins. See Deployment and monitoring for more information on how to deploy modules on IoT Edge devices.

## EdgeAgent desired properties

The module twin for the Edge agent is called `$edgeAgent` and coordinates the communications between the Edge agent running on a device and IoT Hub. The desired properties are set when applying a deployment manifest on a specific device as part of a single-device or at-scale deployment.

| PROPERTY | DESCRIPTION | REQUIRED |
|---|---|---|
| schemaVersion | Has to be "1.0" | Yes |
| runtime.type | Has to be "docker" | Yes |
| runtime.settings.minDockerVersion | Set to the minimum Docker version required by this deployment manifest | Yes |
| runtime.settings.loggingOptions | A stringified JSON containing the logging options for the Edge agent container. Docker logging options | No |
| systemModules.edgeAgent.type | Has to be "docker" | Yes |
| systemModules.edgeAgent.settings.image | The URI of the image of the Edge agent. Currently, the Edge agent is not able to update itself. | Yes |
| systemModules.edgeAgent.settings.createOptions | A stringified JSON containing the options for the creation of the Edge agent container. Docker create options | No |
| systemModules.edgeAgent.configuration.id | The ID of the deployment that deployed this module. | This is set by IoT Hub when this manifest is applied using a deployment. Not part of a deployment manifest. |
| systemModules.edgeHub.type | Has to be "docker" | Yes |
| systemModules.edgeHub.status | Has to be "running" | Yes |
| systemModules.edgeHub.restartPolicy | Has to be "always" | Yes |

| PROPERTY | DESCRIPTION | REQUIRED |
|---|---|---|
| systemModules.edgeHub.settings.image | The URI of the image of the Edge hub. | Yes |
| systemModules.edgeHub.settings.createOptions | A stringified JSON containing the options for the creation of the Edge hub container. Docker create options | No |
| systemModules.edgeHub.configuration.id | The ID of the deployment that deployed this module. | This is set by IoT Hub when this manifest is applied using a deployment. Not part of a deployment manifest. |
| modules.{moduleId}.version | A user-defined string representing the version of this module. | Yes |
| modules.{moduleId}.type | Has to be "docker" | Yes |
| modules.{moduleId}.restartPolicy | {"never" \| "on-failed" \| "on-unhealthy" \| "always"} | Yes |
| modules.{moduleId}.settings.image | The URI to the module image. | Yes |
| modules.{moduleId}.settings.createOptions | A stringified JSON containing the options for the creation of the module container. Docker create options | No |
| modules.{moduleId}.configuration.id | The ID of the deployment that deployed this module. | This is set by IoT Hub when this manifest is applied using a deployment. Not part of a deployment manifest. |

## EdgeAgent reported properties

The Edge agent reported properties include three main pieces of information:

1. The status of the application of the last-seen desired properties;
2. The status of the modules currently running on the device, as reported by the Edge agent; and
3. A copy of the desired properties currently running on the device.

This last piece of information is useful in case the latest desired properties are not applied successfully by the runtime, and the device is still running a previous deployment manifest.

**NOTE**

The reported properties of the Edge agent are useful as they can be queried with the IoT Hub query language to investigate the status of deployments at scale. Refer to Deployments for more information on how to use this feature.

The following table does not include the information that is copied from the desired properties.

| PROPERTY | DESCRIPTION |
|---|---|
| lastDesiredVersion | This integer refers to the last version of the desired properties processed by the Edge agent. |

| PROPERTY | DESCRIPTION |
|---|---|
| lastDesiredStatus.code | This is the status code referring to last desired properties seen by the Edge agent. Allowed values: `200` Success, `400` Invalid configuration, `412` Invalid schema version, `417` the desired properties are empty, `500` Failed |
| lastDesiredStatus.description | Text description of the status |
| deviceHealth | `healthy` if the runtime status of all modules is either `running` or `stopped`, `unhealthy` otherwise |
| configurationHealth.{deploymentId}.health | `healthy` if the runtime status of all modules set by the deployment {deploymentId} is either `running` or `stopped`, `unhealthy` otherwise |
| runtime.platform.OS | Reporting the OS running on the device |
| runtime.platform.architecture | Reporting the architecture of the CPU on the device |
| systemModules.edgeAgent.runtimeStatus | The reported status of Edge agent: {"running" \| "unhealthy"} |
| systemModules.edgeAgent.statusDescription | Text description of the reported status of the Edge agent. |
| systemModules.edgeHub.runtimeStatus | Current status of Edge hub: { "running" \| "stopped" \| "failed" \| "backoff" \| "unhealthy" } |
| systemModules.edgeHub.statusDescription | Text description of the current status of Edge hub if unhealthy. |
| systemModules.edgeHub.exitCode | If exited, the exit code reported by the Edge hub container |
| systemModules.edgeHub.startTimeUtc | Time when Edge hub was last started |
| systemModules.edgeHub.lastExitTimeUtc | Time when Edge hub last exited |
| systemModules.edgeHub.lastRestartTimeUtc | Time when Edge hub was last restarted |
| systemModules.edgeHub.restartCount | Number of times this module was restarted as part of the restart policy. |
| modules.{moduleId}.runtimeStatus | Current status of the module: { "running" \| "stopped" \| "failed" \| "backoff" \| "unhealthy" } |
| modules.{moduleId}.statusDescription | Text description of the current status of the module if unhealthy. |
| modules.{moduleId}.exitCode | If exited, the exit code reported by the module container |
| modules.{moduleId}.startTimeUtc | Time when the module was last started |
| modules.{moduleId}.lastExitTimeUtc | Time when the module last exited |
| modules.{moduleId}.lastRestartTimeUtc | Time when the module was last restarted |

| PROPERTY | DESCRIPTION |
|---|---|
| modules.{moduleId}.restartCount | Number of times this module was restarted as part of the restart policy. |

## EdgeHub desired properties

The module twin for the Edge hub is called `$edgeHub` and coordinates the communications between the Edge hub running on a device and IoT Hub. The desired properties are set when applying a deployment manifest on a specific device as part of a single-device or at-scale deployment.

| PROPERTY | DESCRIPTION | REQUIRED IN THE DEPLOYMENT MANIFEST |
|---|---|---|
| schemaVersion | Has to be "1.0" | Yes |
| routes.{routeName} | A string representing an Edge hub route. | The `routes` element can be present but empty. |
| storeAndForwardConfiguration.timeToLiveSecs | The time in seconds that Edge hub keeps messages in case of disconnected routing endpoints, for example, disconnected from IoT Hub, or local module | Yes |

## EdgeHub reported properties

| PROPERTY | DESCRIPTION |
|---|---|
| lastDesiredVersion | This integer refers to the last version of the desired properties processed by the Edge hub. |
| lastDesiredStatus.code | This is the status code referring to last desired properties seen by the Edge hub. Allowed values: `200` Success, `400` Invalid configuration, `500` Failed |
| lastDesiredStatus.description | Text description of the status |
| clients.{device or module identity}.status | The connectivity status of this device or module. Possible values {"connected" \| "disconnected"}. Only module identities can be in disconnected state. Downstream devices connecting to Edge hub appear only when connected. |
| clients.{device or module identity}.lastConnectTime | Last time the device or module connected |
| clients.{device or module identity}.lastDisconnectTime | Last time the device or module disconnected |

## Next steps

To learn how to use these properties to build out deployment manifests, see Understand how IoT Edge modules can be used, configured, and reused.

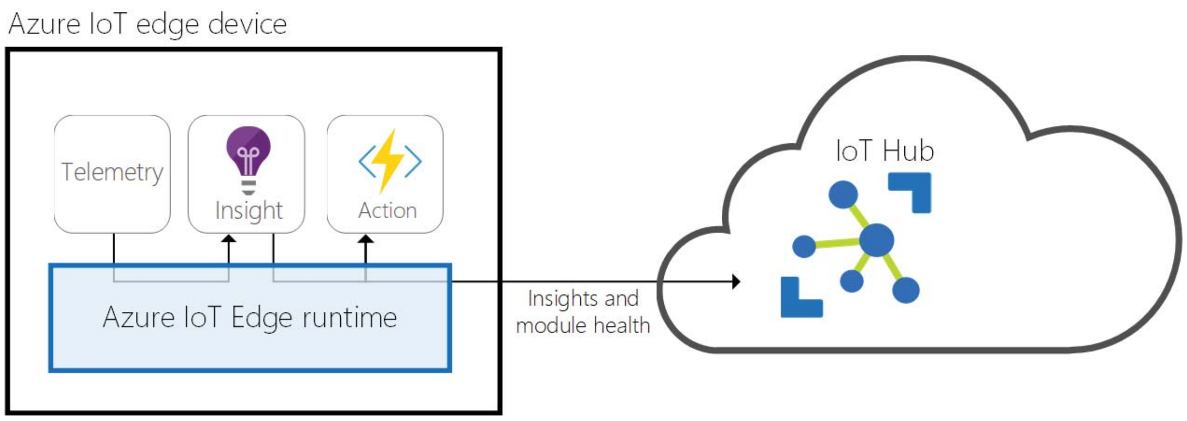
# Understand the Azure IoT Edge runtime and its architecture - preview

2/22/2018 • 6 min to read • Edit Online

The IoT Edge runtime is a collection of programs that need to be installed on a device for it to be considered an IoT Edge device. Collectively, the components of the IoT Edge runtime enable IoT Edge devices to receive code to run at the edge, and communicate the results.

The IoT Edge runtime performs the following functions on IoT Edge devices:

- Installs and updates workloads on the device.
- Maintains Azure IoT Edge security standards on the device.
- Ensures that IoT Edge modules are always running.
- Reports module health to the cloud for remote monitoring.
- Facilitates communication between downstream leaf devices and the IoT Edge device.
- Facilitates communication between modules on the IoT Edge device.
- Facilitates communication between the IoT Edge device and the cloud.



The responsibilities of the IoT Edge runtime fall into two categories: module management and communication. These two roles are performed by two components that make up the IoT Edge runtime. The IoT Edge hub is responsible for communication, while the IoT Edge agent manages deploying and monitoring the modules.

Both the Edge agent and the Edge hub are modules, just like any other module running on an IoT Edge device. For more information about how modules work, see lnk-modules.

## IoT Edge hub

The Edge hub is one of two modules that make up the Azure IoT Edge runtime. It acts as a local proxy for IoT Hub by exposing the same protocol endpoints as IoT Hub. This consistency means that clients (whether devices or modules) can connect to the IoT Edge runtime just as they would to IoT Hub.

**NOTE**

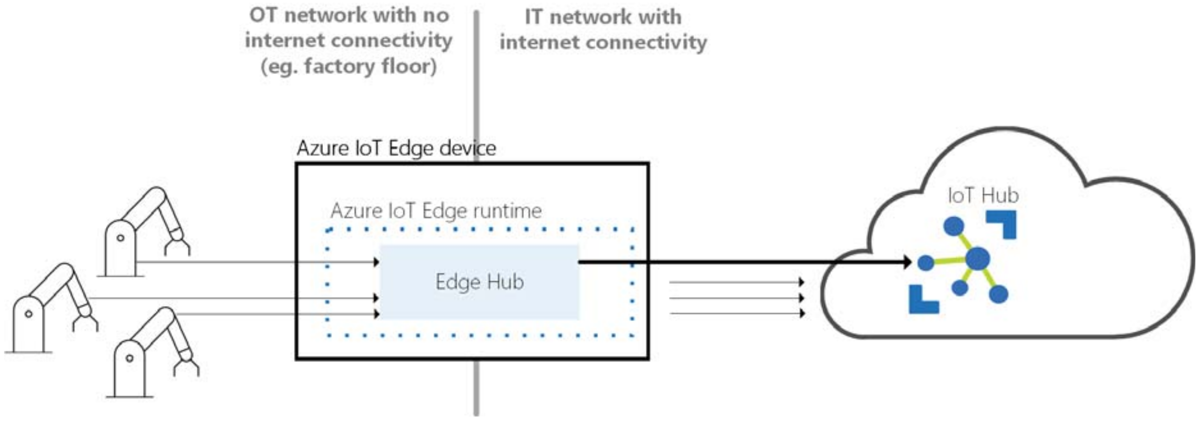
During public preview Edge Hub only supports clients that connect using MQTT.

The Edge hub is not a full version of IoT Hub running locally. There are some things that the Edge hub silently delegates to IoT Hub. For example, Edge hub forwards authentication requests to IoT Hub when a device first tries to connect. After the first connection is established, security information is cached locally by Edge hub. Subsequent connections from that device are allowed without having to authenticate to the cloud.

**NOTE**

During public preview the runtime must be connected every time it tries to authenticate a device.

To reduce the bandwidth your IoT Edge solution uses, the Edge hub optimizes how many actual connections are made to the cloud. Edge hub takes logical connections from clients like modules or leaf devices and combines them for a single physical connection to the cloud. The details of this process are transparent to the rest of the solution. Clients think they have their own connection to the cloud even though they are all being sent over the same connection.


OT network with no internet connectivity (eg. factory floor)
IT network with internet connectivity
Azure IoT Edge device
Azure IoT Edge runtime
Edge Hub
IoT Hub

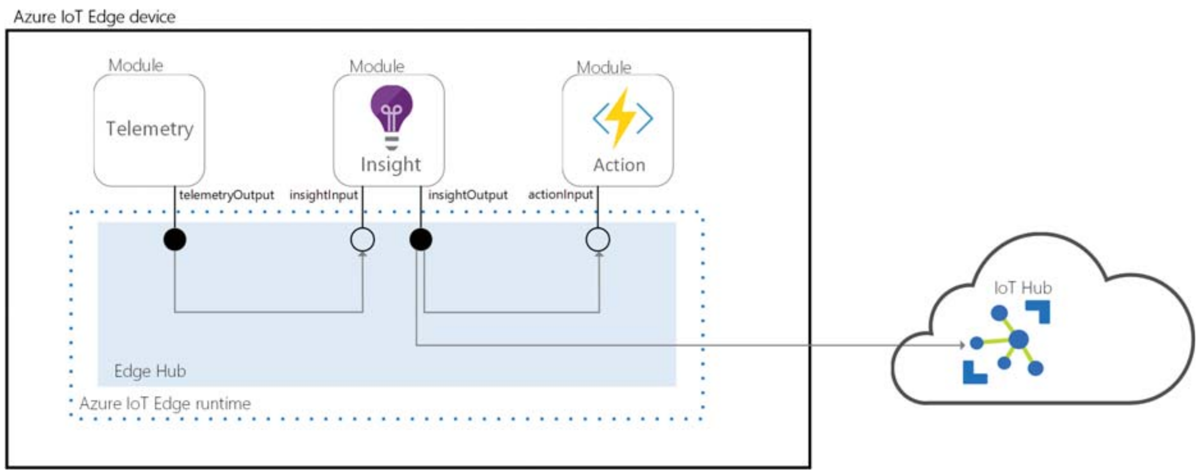
Edge hub can determine whether it's connected to IoT Hub. If the connection is lost, Edge hub saves messages or twin updates locally. Once a connection is reestablished, it syncs all the data. The location used for this temporary cache is determined by a property of the Edge hub's module twin. The size of the cache is not capped and will grow as long as the device has storage capacity.

**NOTE**

Adding control over additional caching parameters will be added to the product before it enters general availability.

### Module communication

Edge Hub facilitates module to module communication. Using Edge Hub as a message broker keeps modules independent from each other. Modules only need to specify the inputs on which they accept messages and the outputs to which they write messages. A solution developer then stitches these inputs and outputs together so that the modules process data in the order specific to that solution.



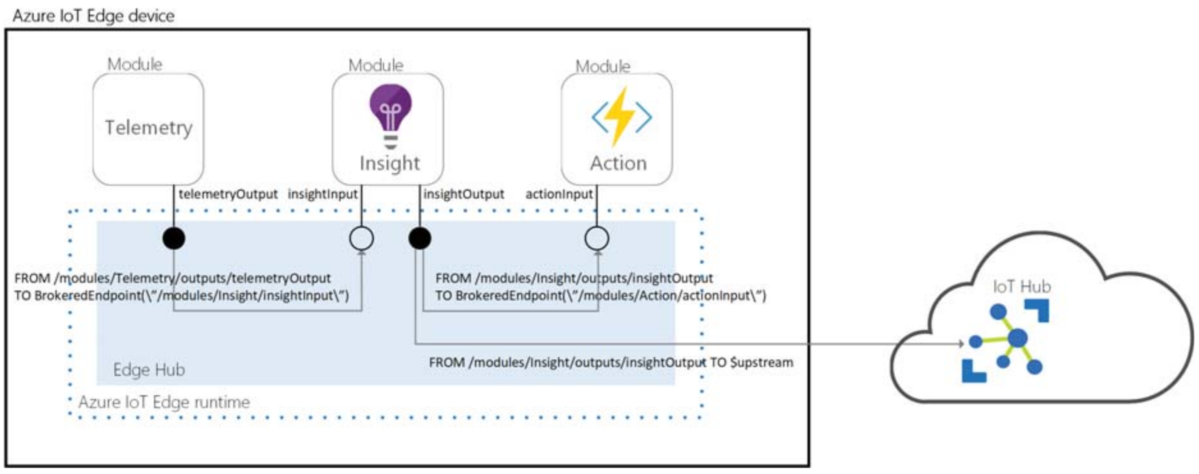
To send data to the Edge hub, a module calls the SendEventAsync method. The first argument specifies on which output to send the message. The following pseudocode sends a message on output1:

```
DeviceClient client = new DeviceClient.CreateFromConnectionString(moduleConnectionString, settings);
await client.OpenAsync();
await client.SendEventAsync(“output1”, message);
```

To receive a message, register a callback that processes messages coming in on a specific input. The following pseudocode registers the function messageProcessor to be used for processing all messages received on input1:

```
await client.SetEventHandlerAsync(“input1”, messageProcessor, userContext);
```

The solution developer is responsible for specifying the rules that determine how Edge hub passes messages between modules. Routing rules are defined in the cloud and pushed down to Edge hub in its device twin. The same syntax for IoT Hub routes is used to define routes between modules in Azure IoT Edge.



Azure IoT Edge device
Module
Telemetry
Module
Insight
Module
Action
telemetryOutput
insightInput
insightOutput
actionInput
FROM /modules/Telemetry/outputs/telemetryOutput
TO BrokeredEndpoint(\”/modules/Insight/insightInput\”)
FROM /modules/Insight/outputs/insightOutput
TO BrokeredEndpoint(\”/modules/Action/actionInput\”)
FROM /modules/Insight/outputs/insightOutput TO $upstream
Edge Hub
Azure IoT Edge runtime
IoT Hub

## IoT Edge agent

The IoT Edge agent is the other module that makes up the Azure IoT Edge runtime. It is responsible for instantiating modules, ensuring that they continue to run, and reporting the status of the modules back to IoT Hub. Just like any other module, the Edge agent uses its module twin to store this configuration data.

To begin execution of the Edge agent, run the azure-iot-edge-runtime-ctl.py start command. The agent retrieves its module twin from IoT Hub and inspects the modules dictionary. The modules dictionary is the collection of modules that need to be started.

Each item in the modules dictionary contains specific information about a module and is used by the Edge agent for controlling the module’s lifecycle. Some of the more interesting properties are:

- **settings.image** – The container image that the Edge agent uses to start the module. The Edge agent must be configured with credentials for the container registry if the image is protected by a password. To configure the Edge agent, use the following command: `azure-iot-edge-runtime-ctl.py –configure`
- **settings.createOptions** – A string that is passed directly to the Docker daemon when starting a module’s container. Adding Docker options in this property allows for advanced options like port forwarding or mounting volumes into a module’s container.
- **status** – The state in which the Edge agent places the module. This value is usually set to *running* as most people want the Edge agent to immediately start all modules on the device. However, you could specify the initial state of a module to be stopped and wait for a future time to tell the Edge agent to start a module. The Edge agent reports the status of each module back to the cloud in the reported properties. A difference between the desired property and the reported property is an indicator or a misbehaving device. The

supported statuses are:
  - Downloading
  - Running
  - Unhealthy
  - Failed
  - Stopped
- **restartPolicy** – How the Edge agent restarts a module. Possible values include:
  - Never – The Edge agent never restarts the module.
  - onFailure - If the module crashes, the Edge agent restarts it. If the module shuts down cleanly, the Edge agent does not restart it.
  - Unhealthy - If the module crashes or is deemed unhealthy, the Edge agent restarts it.
  - Always - If the module crashes, is deemed unhealthy, or shuts down in any way, the Edge agent restarts it.

IoT Edge agent sends runtime response to IoT Hub. Here is a list of possible responses:

- 200 - OK
- 400 - The deployment configuration is malformed or invalid.
- 417 - The device does not have a deployment configuration set.
- 412 - The schema version in the deployment configuration is invalid.
- 406 - The edge device is offline or not sending status reports.
- 500 - An error occurred in the edge runtime.

**Security**

The IoT Edge agent plays a critical role in the security of an IoT Edge device. For example, it performs actions like verifying a module's image before starting it. These features will be added at general availability of V2 features.

## Next steps

- Understand Azure IoT Edge modules

# Understand IoT Edge deployments for single devices or at scale - preview

1/12/2018 • 7 min to read • Edit Online

Azure IoT Edge devices follow a device lifecycle that is similar to other types of IoT devices:

1. IoT Edge devices are provisioned, which involves imaging a device with an OS and installing the IoT Edge runtime.
2. The devices are configured to run IoT Edge modules, and then monitored for health.
3. Finally, devices may be retired when they are replaced or become obsolete.

Azure IoT Edge provides two ways to configure the modules to run on IoT Edge devices: one for development and fast iterations on a single device (that you used in the Azure IoT Edge tutorials), and one for managing large fleets of IoT Edge devices. Both of these approaches are available in the Azure Portal and programmatically.

This article focuses on the configuration and monitoring stages for fleets of devices, collectively referred to as IoT Edge deployments. The overall deployment steps are as follows:

1. An operator defines a deployment that describes a set of modules as well as the target devices. Each deployment has a deployment manifest that reflects this information.
2. The IoT Hub service communicates with all targeted devices to configure them with the desired modules.
3. The IoT Hub service retrieves status from the IoT Edge devices and surfaces those for the operator to monitor. For example, an operator can see when an Edge device is not configured successfully or if a module fails during runtime.
4. At any time, new IoT Edge devices that meet the targeting conditions are configured for the deployment. For example, a deployment that targets all IoT Edge devices in Washington State automatically configures a new IoT Edge device once it is provisioned and added to the Washington State device group.

This article walks through each component involved in configuring and monitoring a deployment. For a walkthrough of creating and updating a deployment, see Deploy and monitor IoT Edge modules at scale.

## Deployment

A deployment assigns IoT Edge module images to run as instances on a targeted set of IoT Edge devices. It works by configuring an IoT Edge deployment manifest to include a list of modules with the corresponding initialization parameters. A deployment can be assigned to a single device (usually based on Device Id) or to a group of devices (based on tags). Once an IoT Edge device receives a deployment manifest, it downloads and installs the module container images from the respective container repositories, and configures them accordingly. Once a deployment is created, an operator can monitor the deployment status to see whether targeted devices are correctly configured.

Devices need to be provisioned as IoT Edge devices to be configured with a deployment. The following are prerequisites, and are not included in the deployment:

- The base operating system
- Docker
- Provisioning of the IoT Edge runtime

**Deployment manifest**

A deployment manifest is a JSON document that describes the modules to be configured on the targeted IoT

Edge devices. It contains the configuration metadata for all the modules, including the required system modules (specifically the IoT Edge agent and IoT Edge hub).

The configuration metadata for each module includes:

- Version
- Type
- Status (e.g. Running or Stopped)
- Re-start policy
- Image and container repository
- Routes for data input and output

#### Target condition

The target condition is continuously evaluated to include any new devices that meet the requirements or remove devices that no longer do through the life time of the deployment. The deployment will be reactivated if the service detects any target condition change. For instance, you have a deployment A which has a target condition tags.environment = 'prod'. When you kick off the deployment, there are 10 prod devices. The modules are successfully installed in these 10 devices. The IoT Edge Agent Status is shown as 10 total devices, 10 successfuly responses, 0 failure responses, and 0 pending responses. Now you add 5 more devices with tags.environment = 'prod'. The service detects the change and the IoT Edge Agent Status becomes 15 total devices, 10 successfuly responses, 0 failure responses, and 5 pending responses when it tries to deploy to the five new devices.

Use any Boolean condition on device twins tags or deviceId to select the target devices. If you want to use condition with tags, you need to add "tags":{} section in the device twin under the same level as properties. Learn more about tags in device twin

Target condition examples:

- deviceId ='linuxprod1'
- tags.environment ='prod'
- tags.environment = 'prod' AND tags.location = 'westus'
- tags.environment = 'prod' OR tags.location = 'westus'
- tags.operator = 'John' AND tags.environment = 'prod' NOT deviceId = 'linuxprod1'

Here are some constrains when you construct a target condition:

- In device twin, you can only build a target condition using tags or deviceId.
- Double quotes aren't allowed in any portion of the target condition. Please use single quotes.
- Single quotes represent the values of the target condition. Therefore, you must escape the single quote with another single quote if it's part of the device name. For example, the target condition for: operator'sDevice would need to be written as deviceId='operator''sDevice'.
- Numbers, letters and the following characters are allowed in target condition values:-:.+%_#*?!(),=@;$

#### Priority

A priority defines whether a deployment should be applied to a targeted device relative to other deployments. A deployment priority is a positive integer, with larger numbers denoting higher priority. If an IoT Edge device is targeted by more than one deployment, the deployment with the highest priority applies. Deployments with lower priorities are not applied, nor are they merged. If a device is targeted with two or more deployments with equal priority, the most recently created deployment (determined by the creation timestamp) applies.

#### Labels

Labels are string key/value pairs that you can use to filter and group of deployments. A deployment may have multiple labels. Labels are optional and do no impact the actual configuration of IoT Edge devices.

#### Deployment status

A deployment can be monitored to determine whether it applied successfully for any targeted IoT Edge device. A targeted Edge device will appear in one or more of the following status categories:

- **Target** shows the IoT Edge devices that match the Deployment targeting condition.
- **Actual** shows the targeted IoT Edge devices that are not targeted by another deployment of higher priority.
- **Healthy** shows the IoT Edge devices that have reported back to the service that the modules have been deployed successfully.
- **Unhealthy** shows the IoT Edge devices have reported back to the service that one or modules have not been deployed successfully. To further investigate the error, connect remotely to that devices and view the log files.
- **Unknown** shows the IoT Edge devices that did not report any status pertaining this deployment. To further investigate, view service info and log files.

## Phased rollout

A phased rollout is an overall process whereby an operator deploys changes to a broadening set of IoT Edge devices. The goal is to make changes gradually to reduce the risk of making wide scale breaking changes.

A phased rollout is executed in the following phases and steps:

1. Establish a test environment of IoT Edge devices by provisioning them and setting a device twin tag like `tag.environment='test'`. The test environment should mirror the production environment that the deployment will eventually target.
2. Create a deployment including the desired modules and configurations. The targeting condition should target the test IoT Edge device environment.
3. Validate the new module configuration in the test environment.
4. Update the deployment to include a subset of production IoT Edge devices by adding a new tag to the targeting condition. Also, ensure that the priority for the deployment is higher than other deployments currently targeted to those devices
5. Verify that the deployment succeeded on the targeted IoT Devices by viewing the deployment status.
6. Update the deployment to target all remaining production IoT Edge devices.

## Rollback

Deployments can be rolled back in case of errors or misconfigurations. Because a deployment defines the absolute module configuration for an IoT Edge device, an additional deployment must also be targeted to the same device at a lower priority even if the goal is to remove all modules.

Perform rollbacks in the following sequence:

1. Confirm that a second deployment is also targeted at the same device set. If the goal of the rollback is to remove all modules, the second deployment should not include any modules.
2. Modify or remove the target condition expression of the deployment you wish to roll back so that the devices no longer meet the targeting condition.
3. Verify that the rollback succeeded by viewing the deployment status.
   - The rolled-back deployment should no longer show status for the devices that were rolled back.
   - The second deployment should now include deployment status for the devices that were rolled back.

## Next steps

- Walk through the steps to create, update, or delete a deployment in Deploy and monitor IoT Edge modules at scale.
- Learn more about other IoT Edge concepts like the IoT Edge runtime and IoT Edge modules.

# How an IoT Edge device can be used as a gateway - preview

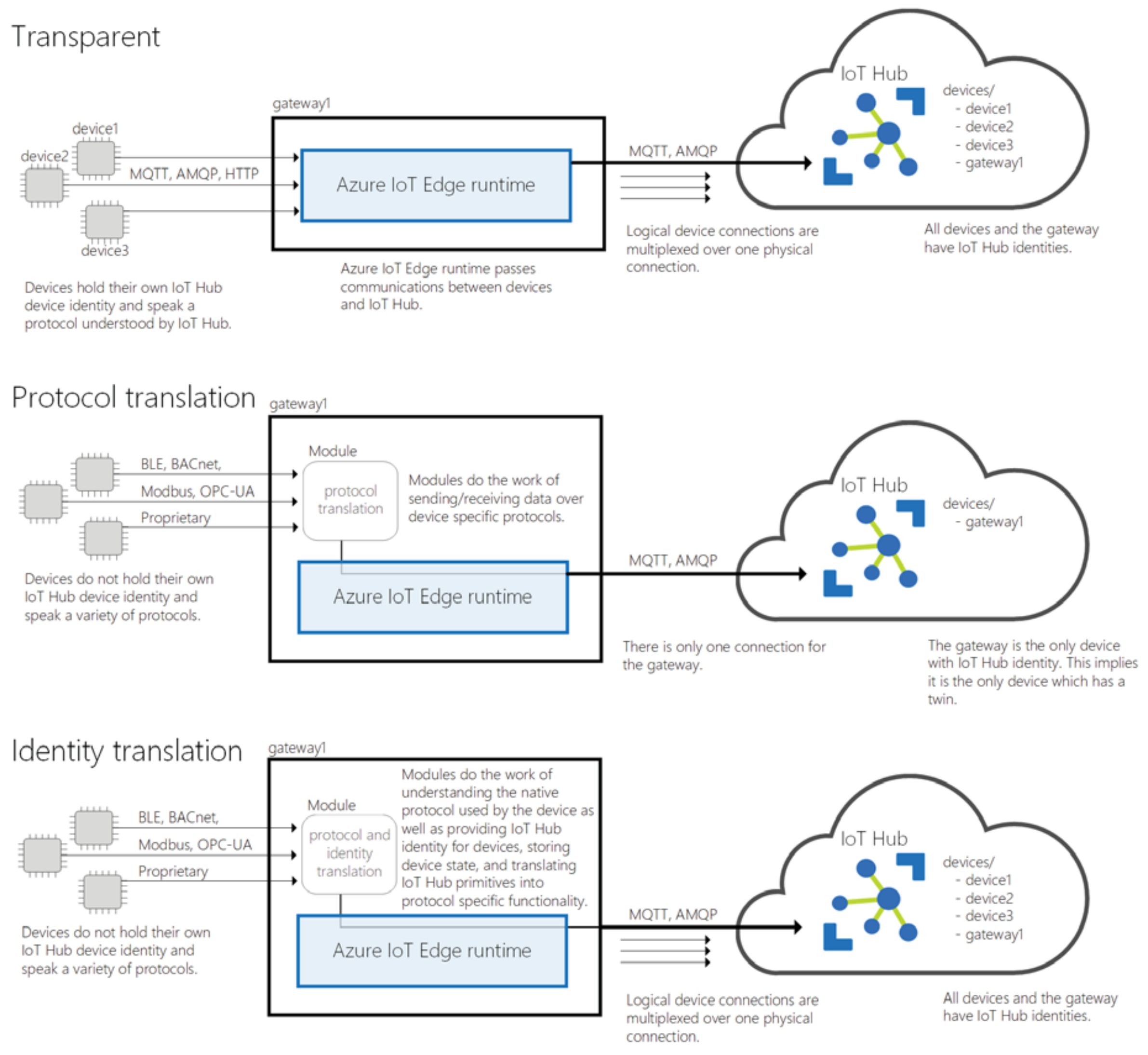
2/15/2018 • 4 min to read • Edit Online

The purpose of gateways in IoT solutions is specific to the solution and combine device connectivity with edge analytics. Azure IoT Edge can be used to satisfy all needs for an IoT gateway regardless of whether they are related to connectivity, identity, or edge analytics. Gateway patterns in this article only refer to characteristics of downstream device connectivity and device identity, not how device data is processed on the gateway.

## Patterns

There are three patterns for using an IoT Edge device as a gateway: transparent, protocol translation, and identity translation:

- **Transparent** – Devices that theoretically could connect to IoT Hub can connect to a gateway device instead. This implies that the downstream devices have their own IoT Hub identities and are using any of the MQTT, AMQP, or HTTP protocols. The gateway simply passes communications between the devices and IoT Hub. The devices are unaware that they are communicating with the cloud via a gateway and a user interacting with the devices in IoT Hub is unaware of the intermediate gateway device. Thus, the gateway is transparent. Refer to the Create a transparent gateway how-to for specifics on using an IoT Edge device as a transparent gateway.
- **Protocol translation** – Devices that do not support MQTT, AMQP, or HTTP use a gateway device to send data to IoT Hub. The gateway is smart enough to understand that protocol used by the downstream devices; however it is the only device that has identity in IoT Hub. All information looks like it is coming from one device, the gateway. This implies that downstream devices must embed additional identifying information in their messages if cloud applications want to reason about the data on a per device basis. Additionally, IoT Hub primitives like twin and methods are only available for the gateway device, not downstream devices.
- **Identity translation** - Devices that cannot connect to IoT Hub connect to a gateway device which provides IoT Hub identity and protocol translation on behalf of the downstream devices. The gateway is smart enough to understand the protocol used by the downstream devices, provide them identity, and translate IoT Hub primitives. Downstream devices appear in IoT Hub as first-class devices with twins and methods. A user can interact with the devices in IoT Hub and is unaware of the intermediate gateway device.


Transparent
gateway1
device1
device2
device3
MQTT, AMQP, HTTP
Azure IoT Edge runtime
MQTT, AMQP
IoT Hub
devices/
- device1
- device2
- device3
- gateway1
Devices hold their own IoT Hub device identity and speak a protocol understood by IoT Hub.
Azure IoT Edge runtime passes communications between devices and IoT Hub.
Logical device connections are multiplexed over one physical connection.
All devices and the gateway have IoT Hub identities.
Protocol translation
gateway1
BLE, BACnet,
Modbus, OPC-UA
Proprietary
Module
protocol translation
Modules do the work of sending/receiving data over device specific protocols.
Azure IoT Edge runtime
MQTT, AMQP
IoT Hub
devices/
- gateway1
Devices do not hold their own IoT Hub device identity and speak a variety of protocols.
There is only one connection for the gateway.
The gateway is the only device with IoT Hub identity. This implies it is the only device which has a twin.
Identity translation
gateway1
BLE, BACnet,
Modbus, OPC-UA
Proprietary
Module
protocol and identity translation
Modules do the work of understanding the native protocol used by the device as well as providing IoT Hub identity for devices, storing device state, and translating IoT Hub primitives into protocol specific functionality.
Azure IoT Edge runtime
MQTT, AMQP
IoT Hub
devices/
- device1
- device2
- device3
- gateway1
Devices do not hold their own IoT Hub device identity and speak a variety of protocols.
Logical device connections are multiplexed over one physical connection.
All devices and the gateway have IoT Hub identities.

## Use cases

All gateway patterns provide the following benefits:

- **Edge analytics** – Use AI services locally to process data coming from downstream devices without sending full fidelity telemetry to the cloud. Find and react to insights locally and only send a subset of data to IoT Hub.
- **Downstream device isolation** – The gateway device can shield all downstream devices from exposure to the internet. It can sit in between an OT network which does not have connectivity and an IT network which provides access to the web.
- **Connection multiplexing** - All devices connecting to IoT Hub through an IoT Edge device will use the same underlying connection.
- **Traffic smoothing** - The IoT Edge device will automatically implement exponential backoff in case of IoT Hub throttling, while persisting the messages locally. This will make your solution resilient to spikes in traffic.
- **Limited offline support** - The gateway device will store locally messages and twin updates that cannot be delivered to IoT Hub.

A gateway does protocol translation can also perform edge analytics, device isolation, traffic smoothing, and offline support to existing devices and new devices that are resource constrained. Many existing devices are producing data that can power business insights; however they were not designed with cloud connectivity in mind. Opaque gateways allow this data to be unlocked and utilized in an end to end IoT solution.

A gateway that does identity translation provide the benefits of protocol translation and additionally allow for full manageability of downstream devices from the cloud. All devices in your IoT solution show up in IoT Hub regardless of the protocol with they speak.

## Cheat sheet

Here is a quick cheat sheet that compares IoT Hub primitives when using transparent, opaque, and proxy gateways.

| | TRANSPARENT GATEWAY | PROTOCOL TRANSLATION | IDENTITY TRANSLATION |
|---|---|---|---|
| Identities stored in the IoT Hub identity registry | Identities of all connected devices | Only the identity of the gateway device | Identities of all connected devices |
| Device twin | Each connected device has its own device twin | Only the gateway has a device and module twins | Each connected device has its own device twin |
| Direct methods and cloud-to-device messages | The cloud can address each connected device individually | The cloud can only address the gateway device | The cloud can address each connected device individually |
| IoT Hub throttles and quotas | Apply to each device | Apply to the gateway device | Apply to each device |

When using an opaque gateway (protocol translation) pattern, all devices connecting through that gateway share the same cloud-to-device queue, which can contain at most 50 messages. It follows that the opaque gateway pattern should be used only when very few devices are connecting through each field gateway, and their cloud-to-device traffic is low.

## Next steps

Use an IoT Edge device as a transparent gateway

# Securing Azure IoT Edge - preview

11/15/2017 • 6 min to read • Edit Online

Securing the intelligent edge is necessary to confer confidence in the operation of an end to end IoT solution. Azure IoT Edge is designed for security that is extensible to different risk profiles, deployment scenarios, and offers the same protection that you expect from all Azure services.

Azure IoT Edge runs on different hardware, supports both Linux and Windows, and is applicable to diverse deployment scenarios. Assessed risk depends on many considerations including solution ownership, deployment geography, data sensitivity, privacy, application vertical, and regulatory requirements. Rather than offering concrete solutions to specific scenarios, it makes sense to design an extensible security framework based on well-grounded principles designed for scale.

This article provides an overview of the security framework. For more information, see Securing the intelligent edge.

**NOTE**

The security framework described below is being add to the product now and will be available at the general availability release of Azure IoT Edge. The product is currently in public preview, a release intended to allow development and prototyping of edge solutions, not full production deployments that need the full security framework.

## Standards

Standards promote ease of scrutiny and ease of implementation, which are the hallmark of security. A well architected security solution should lend itself to scrutiny under evaluation to build trust and should not be a hurdle to deployment. The design of the framework to secure Azure IoT Edge emanates from time-tested and industry proven security protocols to leverage familiarity and reuse.

## Authentication

Knowing without a doubt what actors, devices, and components are participating in the delivery of value through an end to end IoT solution is paramount in building trust. Such knowledge offers secure accountability of participants to enabling basis for admission. Azure IoT Edge attains this knowledge through authentication. The primary mechanism for authentication for the Azure IoT Edge platform is certificate-based authentication. This mechanism derives from a set of standards governing Public Key Infrastructure (PKiX) by the Internet Engineering Task Force (IETF).

The Azure IoT Edge security framework calls for unique certificate identities for all devices, modules (containers that encapsulate logic within the device), and actors interacting with the Azure IoT Edge device be it physically or through a network connection. Not every scenario or component may lend itself to certificate-based authentication for which the extensibility of the security framework offers secure avenues for accommodation.

## Authorization

The ability to delegate authority and control access is crucial towards achieving a fundamental security principle – the principle of least privilege. Devices, modules, and actors may gain access only to resources and data within their permission scope and only when it is architecturally allowable. This means some permissions are configurable with sufficient privileges and others architecturally enforced. For example, while a module may be authorized through privileged configuration to initiate a connection to Azure IoT Hub, there is no reason why a module in one Azure

IoT Edge device should access the twin of a module in another Azure IoT Edge device. For this reason, the latter would be architecturally precluded.

Other authorization schemes include certificate signing rights, and role-based access control (RBAC). The extensibility of the security framework permits adoption of other mature authorization schemes.

## Attestation

Attestation ensures the integrity of software bits. It is important for the detection and prevention of malware. The Azure IoT Edge security framework classifies attestation under three main categories:

- Static attestation
- Runtime attestation
- Software attestation

**Static attestation**

Static attestation is the verification of the integrity of all software bits including the operating systems, all runtimes, and configuration information at device power-up. It is often referred to as secure boot. The security framework for Azure IoT Edge devices extends to silicon vendors and incorporates secure hardware ingrained capabilities to assure static attestation processes. These processes include secure boot and secure firmware upgrade processes. Working in close collaboration with silicon vendors eliminates superfluous firmware layers thereby minimizing the threat surface.

**Runtime attestation**

Once a system has completed a validated boot process and is up and running, well designed secure systems would detect attempts to inject malware through the systems ports and interfaces and take proper countermeasures. For intelligent edge devices in physical custody of malicious actors, it is possible to inject malcontent through means other than device interfaces like tampering and side-channel attacks. Such malcontent, which can be in the form of malware or unauthorized configuration changes, would normally not be detected by the secure boot process because they happen after the boot process. Countermeasures offered or enforced by the device’s hardware greatly contributes towards warding off such threats. The security framework for Azure IoT Edge explicitly calls out for extensions for combatting runtime threats.

**Software attestation**

All healthy systems including intelligent edge systems must be amenable to patches and upgrades. Security is important for the update processes otherwise they can be potential threat vectors. The security framework for Azure IoT Edge calls for updates through measured and signed packages to assure the integrity and authenticate the source of the packages. This is applicable to all operating systems and application software bits.

## Hardware root of trust

For many deployments of intelligent edge devices, especially those deployed in places where potential malicious actors have physical access to the device, security offered by the device hardware is the last defense for protection. For this reason, anchoring trust in tamper resistant hardware is most crucial for such deployments. The security framework for Azure IoT Edge entails collaboration of secure silicon hardware vendors to offer different flavors of hardware root of trust to accommodate various risk profiles and deployment scenarios. These include use of secure silicon adhering to common security protocol standards like Trusted Platform Module (ISO/IEC 11889) and Trusted Computing Group’s Device Identifier Composition Engine (DICE). These also include secure enclave technologies like TrustZones and Software Guard Extensions (SGX).

## Certification

To help customers make informed decisions when procuring Azure IoT Edge devices for their deployment, the Azure IoT Edge framework includes certification requirements. Foundational to these requirements are certifications pertaining to security claims and certifications pertaining to validation of the security implementation. For example, a security claim certification would inform that the IoT Edge device uses secure hardware known to resist boot attacks. A validation certification would inform that the secure hardware was properly implemented to offer this value in the device. In keeping with the principle of simplicity, the framework offers the vision of keeping the burden of certification minimal.

## Extensibility

Extensibility is a first-class citizen in the Azure IoT Edge security framework. With IoT technology driving different types of business transformations, it stands to reason that security should seamlessly evolve in lockstep to address emerging scenarios. The Azure IoT Edge security framework starts with a solid foundation on which it builds in extensibility into different dimensions to include:

- First party security services like the Device Provisioning Service for Azure IoT Hub.
- Third-party services like managed security services for different application verticals (like industrial or healthcare) or technology focus (like security monitoring in mesh networks or silicon hardware attestation services) through a rich network of partners.
- Legacy systems to include retrofitting with alternate security strategies, like using secure technology other than certificates for authentication and identity management.
- Secure hardware for seamless adoption of emerging secure hardware technologies and silicon partner contributions.

These are just a few examples of dimensions for extensibility and Azure IoT Edge security framework is designed to be secure from the ground up to support this extensibility.

In the end, the highest success in securing the intelligent edge results from collaborative contributions from an open community driven by the common interest in securing IoT. These contributions might be in the form of secure technologies or services. The Azure IoT Edge security framework offers a solid foundation for security that is extensible for the maximum coverage to offer the same level of trust and integrity in the intelligent edge as with Azure cloud.

## Next steps

Read more about how Azure IoT Edge is Securing the intelligent edge.

# Glossary of IoT Edge terms

2/15/2018 • 3 min to read • Edit Online

This article lists some of the common terms used in the IoT Edge articles.

## Automatic Device Management

Automatic Device Management in Azure IoT Hub automates many of the repetitive and complex tasks of managing large device fleets over the entirety of their lifecycles. With Automatic Device Management, you can target a set of devices based on their properties, define a desired configuration, and let IoT Hub update devices whenever they come into scope. Consists of automatic device configurations and IoT Edge automatic deployments.

## IoT Edge

Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premises devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. For more information please see Azure IoT Edge.

## IoT Edge agent

The part of the IoT Edge runtime responsible for deploying and monitoring modules.

## IoT Edge device

IoT Edge devices have the IoT Edge runtime installed and are flagged as "IoT Edge device" in the device details. Learn how to deploy Azure IoT Edge on a simulated device in Linux - preview.

## IoT Edge automatic deployment

An IoT Edge automatic deployment configures a target set of IoT Edge devices to run a set of IoT Edge modules. Each deployment continuously ensures that all devices that match its target condition are running the specified set of modules, even when new devices are created or are modified to match the target condition. Each IoT Edge device only receives the highest priority deployment whose target condition it meets. Learn more about IoT Edge automatic deployment.

## IoT Edge deployment manifest

A Json document containing the information to be copied in one or more IoT Edge devices' module twin(s) to deploy a set of modules, routes and associated module desired properties.

## IoT Edge gateway device

An IoT Edge device with downstream device. The downstream device can be either IoT Edge or not IoT Edge device.

## IoT Edge hub

The part of the IoT Edge runtime responsible for module to module communications, upstream (toward IoT Hub) and downstream (away from IoT Hub) communications.

## IoT Edge leaf device

An IoT Edge device with no downstream device.

## IoT Edge module

An IoT Edge module is a Docker container that you can deploy to IoT Edge devices. It performs a specific task, such as ingesting a message from a device, transforming a message, or sending a message to an IoT hub. It communicates with other modules and sends data to the IoT Edge runtime. Understand the requirements and tools for developing IoT Edge modules.

## IoT Edge module identity

A record in the IoT Hub module identity registry detailing the existence and security credentials to be used by a module to authenticate with an edge hub or IoT Hub.

## IoT Edge module image

The docker image that is used by the IoT Edge runtime to instantiate module instances.

## IoT Edge module twin

A Json document persisted in the IoT Hub that stores the state information for a module instance.

## IoT Edge priority

When two IoT Edge deployments target the same device, the deployment with higher priority gets applied. If two deployments have the same priority, the deployment with the later creation date gets applied. Learn more about priority.

## IoT Edge runtime

IoT Edge runtime includes everything that Microsoft distributes to be installed on an IoT Edge device. It included Edge agent, Edge hub and Edge CTL tool.

## IoT Edge set modules to a single device

An operation that copies the content of an IoT Edge manifest on one device' module twin. The underlying API is a generic 'apply configuration', which simply takes an IoT Edge manifest as an input.

## IoT Edge target condition

In an IoT Edge deployment, Target condition is any Boolean condition on device twins’ tags to select the target devices of the deployment, e.g. "tag.environment = prod". The target condition is continuously evaluated to include any new devices that meet the requirements or remove devices that no longer do. Learn more about target condition

## Next steps

- See IoT Hub glossary
- Learn IoT Edge module development
- Learn IoT Edge deployment