# Table of Contents

# Azure and Internet of Things

8/24/2017 • 5 min to read • Edit Online

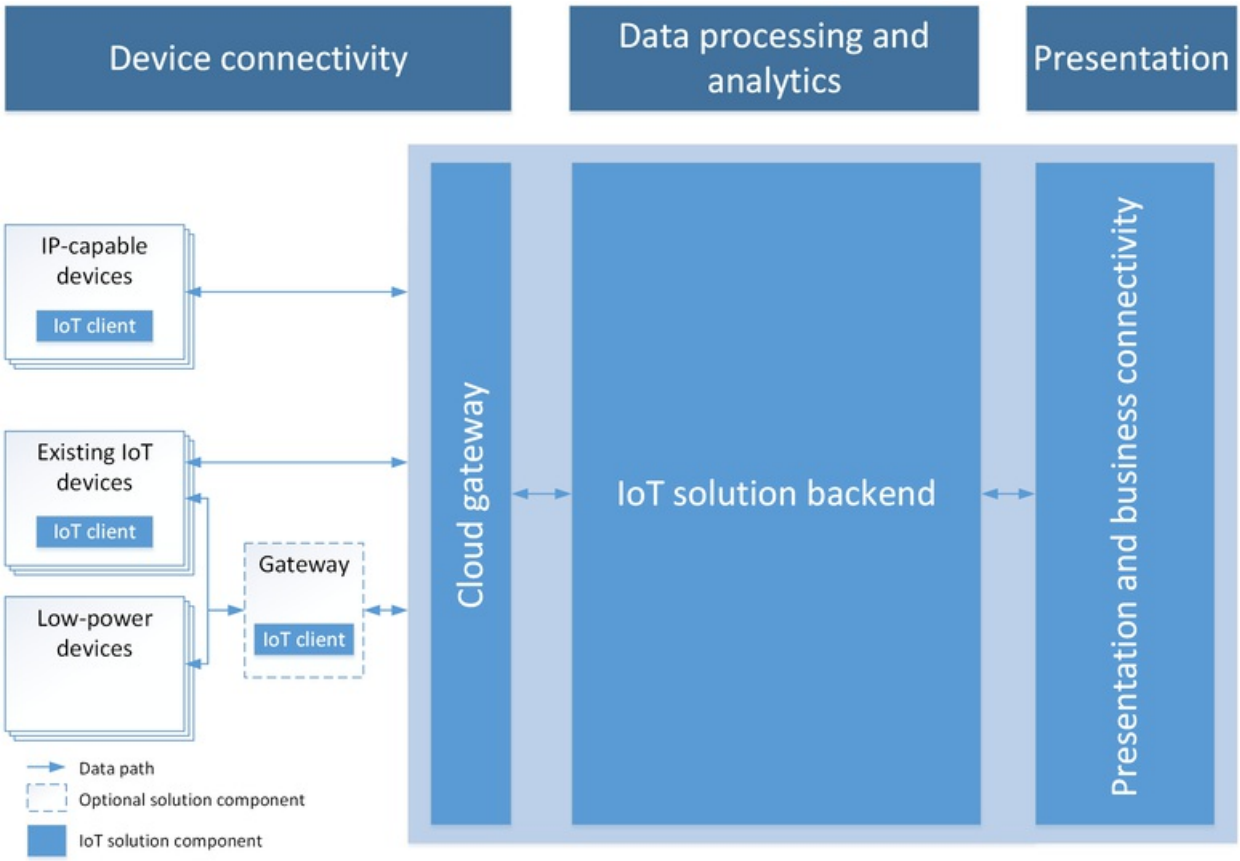
Welcome to Microsoft Azure and the Internet of Things (IoT). This article introduces an IoT solution architecture that describes the common characteristics of an IoT solution you might deploy using Azure services. IoT solutions require secure, bidirectional communication between devices, possibly numbering in the millions, and a solution back end. For example, a solution back end might use automated, predictive analytics to uncover insights from your device-to-cloud event stream.

Azure IoT Hub is a key building block when you implement this IoT solution architecture using Azure services. IoT Suite provides complete, end-to-end, implementations of this architecture for specific IoT scenarios. For example:

- The *remote monitoring* solution enables you to monitor the status of devices such as vending machines.
- The *predictive maintenance* solution helps you to anticipate maintenance needs of devices such as pumps in remote pumping stations and to avoid unscheduled downtime.
- The *connected factory* solution helps you to connect and monitor your industrial devices.

## IoT solution architecture

The following diagram shows a typical IoT solution architecture. The diagram does not include the names of any specific Azure services, but describes the key elements in a generic IoT solution architecture. In this architecture, IoT devices collect data that they send to a cloud gateway. The cloud gateway makes the data available for processing by other back-end services from where data is delivered to other line-of-business applications or to human operators through a dashboard or other presentation device.



> **NOTE**
>
> For an in-depth discussion of IoT architecture, see the Microsoft Azure IoT Reference Architecture.

**Device connectivity**

In this IoT solution architecture, devices send telemetry, such as sensor readings from a pumping station, to a cloud endpoint for storage and processing. In a predictive maintenance scenario, the solution back end might use the stream of sensor data to determine when a specific pump requires maintenance. Devices can also receive and respond to cloud-to-device messages by reading messages from a cloud endpoint. For example, in the predictive maintenance scenario the solution back end might send messages to other pumps in the pumping station to begin rerouting flows just before maintenance is due to start. This procedure would make sure the maintenance engineer could get started as soon as she arrives.

One of the biggest challenges facing IoT projects is how to reliably and securely connect devices to the solution back end. IoT devices have different characteristics as compared to other clients such as browsers and mobile apps. IoT devices:

- Are often embedded systems with no human operator.
- Can be deployed in remote locations, where physical access is expensive.
- May only be reachable through the solution back end. There is no other way to interact with the device.
- May have limited power and processing resources.
- May have intermittent, slow, or expensive network connectivity.
- May need to use proprietary, custom, or industry-specific application protocols.
- Can be created using a large set of popular hardware and software platforms.

In addition to the requirements above, any IoT solution must also deliver scale, security, and reliability. The resulting set of connectivity requirements is hard and time-consuming to implement using traditional technologies such as web containers and messaging brokers. Azure IoT Hub and the Azure IoT device SDKs make it easier to implement solutions that meet these requirements.

A device can communicate directly with a cloud gateway endpoint, or if the device cannot use any of the communications protocols that the cloud gateway supports, it can connect through an intermediate gateway. For example, the Azure IoT protocol gateway can perform protocol translation if devices cannot use any of the protocols that IoT Hub supports.

**Data processing and analytics**

In the cloud, an IoT solution back end is where most of the data processing occurs, such as filtering and aggregating telemetry and routing it to other services. The IoT solution back end:

- Receives telemetry at scale from your devices and determines how to process and store that data.
- May enable you to send commands from the cloud to specific device.
- Provides device registration capabilities that enable you to provision devices and to control which devices are permitted to connect to your infrastructure.
- Enables you to track the state of your devices and monitor their activities.

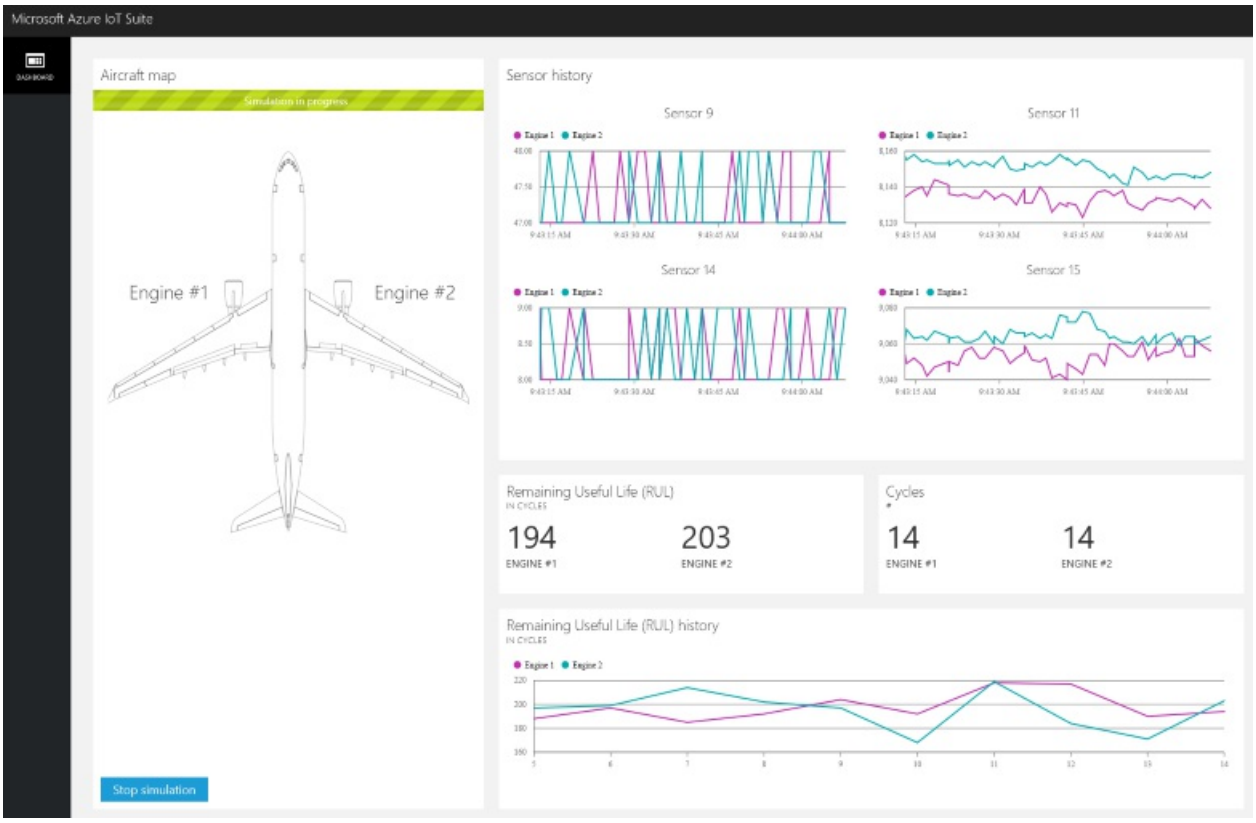
In the predictive maintenance scenario, the solution back end stores historical telemetry data. The solution back end can use this data to use to identify patterns that indicate maintenance is due on a specific pump.

IoT solutions can include automatic feedback loops. For example, an analytics module in the solution back end can identify from telemetry that the temperature of a specific device is above normal operating levels. The solution can then send a command to the device, instructing it to take corrective action.

**Presentation and business connectivity**

The presentation and business connectivity layer allows end users to interact with the IoT solution and the devices.

It enables users to view and analyze the data collected from their devices. These views can take the form of dashboards or BI reports that can display both historical data or near real-time data. For example, an operator can check on the status of particular pumping station and see any alerts raised by the system. This layer also allows integration of the IoT solution back end with existing line-of-business applications to tie into enterprise business processes or workflows. For example, the predictive maintenance solution can integrate with a scheduling system that books an engineer to visit a pumping station when the solution identifies a pump in need of maintenance.




## Azure IoT Suite

The Microsoft Azure IoT Suite is an enterprise-grade solution that enables you to get started quickly through a set of extensible preconfigured solutions. These solutions address common IoT scenarios, such as remote monitoring, predictive maintenance, and connected factory. These solutions are implementations of the IoT solution architecture outlined in this article.

The preconfigured solutions are complete, working, end-to-end solutions that include:

- Simulated devices to get you started.
- Preconfigured Azure services such as Azure IoT Hub, Azure Event Hubs, Azure Stream Analytics, Azure Machine Learning, and Azure storage.
- Solution-specific management consoles.

The preconfigured solutions contain proven, production-ready code that you can customize and extend to implement your own specific IoT scenarios.

You may also be interested in the Azure IoT Hub service that many of the preconfigured solutions use. Azure IoT Hub provides the secure and reliable bi-directional communications between devices and the cloud used in the preconfigured solution architecture.

## Next steps

Explore these resources to continue learning about IoT Suite and the preconfigured solutions:

- What is Azure IoT Suite?

- What are the Azure IoT Suite preconfigured solutions?

# Overview of Azure IoT Suite

7/24/2017 • 2 min to read • Edit Online

The Azure internet of things (IoT) services offer a broad range of capabilities. These enterprise grade services enable you to:

- Collect data from devices
- Analyze data streams in-motion
- Store and query large data sets
- Visualize both real-time and historical data
- Integrate with back-office systems
- Manage your devices

To deliver these capabilities, Azure IoT Suite packages together multiple Azure services with custom extensions as *preconfigured solutions*. These preconfigured solutions are base implementations of common IoT solution patterns that help to reduce the time you take to deliver your IoT solutions. Using the IoT software development kits, you can customize and extend these solutions to meet your own requirements. You can also use these solutions as examples or templates when you are developing new IoT solutions.

The following video provides an introduction to Azure IoT Suite:

## Azure IoT services in Azure IoT Suite

The preconfigured solutions typically use the following services:

- Core to Azure IoT Suite is the Azure IoT Hub service. This service provides the device-to-cloud and cloud-to-device messaging capabilities and acts as the gateway to the cloud and the other key IoT Suite services. The service enables you to receive messages from your devices at scale, and send commands to your devices. The service also enables you to manage your devices. For example, you can configure, reboot, or perform a factory reset on one or more devices connected to the hub.
- Azure Stream Analytics provides in-motion data analysis. IoT Suite uses this service to process incoming telemetry, perform aggregation, and detect events. The preconfigured solutions also use stream analytics to process informational messages that contain data such as metadata or command responses from devices. The solutions use Stream Analytics to process the messages from your devices and deliver those messages to other services.
- Azure Storage and Azure Cosmos DB provide the data storage capabilities. The preconfigured solutions use blob storage to store telemetry and to make it available for analysis. The solutions use Cosmos DB to store device metadata and enable the device management capabilities of the solutions.
- Azure Web Apps and Microsoft Power BI provide the data visualization capabilities. The flexibility of Power BI enables you to quickly build your own interactive dashboards that use IoT Suite data.

For an overview of the architecture of a typical IoT solution, see Microsoft Azure and the Internet of Things (IoT).

## Preconfigured solutions

IoT Suite includes preconfigured solutions that enable you to quickly get started with and to explore the common IoT scenarios, such as:

- Remote monitoring
- Predictive maintenance
- Connected factory

You can deploy these solutions to your Azure subscription and then run a complete, end-to-end IoT scenario.

## Next steps

Now that you have an overview of what IoT Suite can do and what are its main components, you can learn more about the preconfigured solutions in IoT Suite. For more information, see What are the Azure IoT preconfigured solutions?

# What are the Azure IoT Suite preconfigured solutions?

7/25/2017 • 7 min to read • Edit Online

The Azure IoT Suite preconfigured solutions are implementations of common IoT solution patterns that you can deploy to Azure using your subscription. You can use the preconfigured solutions:

- As a starting point for your own IoT solutions.
- To learn about common patterns in IoT solution design and development.

Each preconfigured solution is a complete, end-to-end implementation that uses simulated devices to generate telemetry.

You can download the complete source code to customize and extend the solution to meet your specific IoT requirements.

> **NOTE**
> To deploy one of the preconfigured solutions, visit Microsoft Azure IoT Suite. The article Get started with the IoT preconfigured solutions provides more information about how to deploy and run one of the solutions.

The following table shows how the solutions map to specific IoT features:

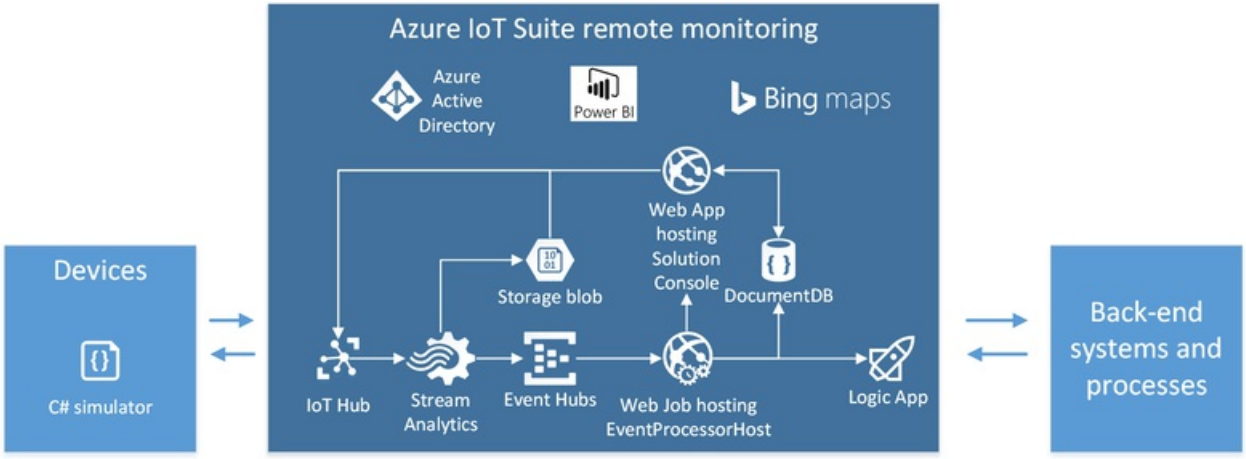
| SOLUTION | DATA INGESTION | DEVICE IDENTITY | DEVICE MANAGEMENT | COMMAND AND CONTROL | RULES AND ACTIONS | PREDICTIVE ANALYTICS |
|---|---|---|---|---|---|---|
| Remote monitoring | Yes | Yes | Yes | Yes | Yes | - |
| Predictive maintenance | Yes | Yes | - | Yes | Yes | Yes |
| Connected factory | Yes | Yes | Yes | Yes | Yes | - |

- *Data ingestion*: Ingress of data at scale to the cloud.
- *Device identity*: Manage unique device identities and control device access to the solution.
- *Device management*: Manage device metadata and perform operations such as device reboots and firmware upgrades.
- *Command and control*: To cause the device to take an action, send messages to a device from the cloud.
- *Rules and actions*: To act on specific device-to-cloud data, the solution back end uses rules.
- *Predictive analytics*: The solution back end analyzes device-to-cloud data to predict when specific actions should take place. For example, analyzing aircraft engine telemetry to determine when engine maintenance is due.

## Remote Monitoring preconfigured solution overview

We have chosen to discuss the remote monitoring preconfigured solution in this article because it illustrates many common design elements that the other solutions share.

The following diagram illustrates the key elements of the remote monitoring solution. The following sections provide more information about these elements.




## Devices

When you deploy the remote monitoring preconfigured solution, four simulated devices are pre-provisioned in the solution that simulate a cooling device. These simulated devices have a built-in temperature and humidity model that emits telemetry. These simulated devices are included to:

- Illustrate the end-to-end flow of data through the solution.
- Provide a convenient source of telemetry.
- Provide a target for methods or commands if you are a back-end developer using the solution as a starting point for a custom implementation.

The simulated devices in the solution can respond to the following cloud-to-device communications:

- *Methods (direct methods)*: A two-way communication method where a connected device is expected to respond immediately.
- *Commands (cloud-to-device messages)*: A one-way communication method where a device retrieves the command from a durable queue.

For a comparison of these different approaches, see Cloud-to-device communications guidance.

When a device first connects to IoT Hub in the preconfigured solution, it sends a device information message to the hub. This message enumerates the methods the device can respond to. In the remote monitoring preconfigured solution, simulated devices support these methods:

- *Initiate Firmware Update*: this method initiates an asynchronous task on the device to perform a firmware update. The asynchronous task uses reported properties to deliver status updates to the solution dashboard.
- *Reboot*: this method causes the simulated device to reboot.
- *FactoryReset*: this method triggers a factory reset on the simulated device.

When a device first connects to IoT Hub in the preconfigured solution, it sends a device information message to the hub. This message enumerates the commands the device can respond to. In the remote monitoring preconfigured solution, simulated devices support these commands:

- *Ping Device*: The device responds to this command with an acknowledgement. This command is useful for checking that the device is still active and listening.
- *Start Telemetry*: Instructs the device to start sending telemetry.
- *Stop Telemetry*: Instructs the device to stop sending telemetry.
- *Change Set Point Temperature*: Controls the simulated temperature telemetry values the device sends. This command is useful for testing back-end logic.

- *Diagnostic Telemetry*: Controls if the device should send the external temperature as telemetry.
- *Change Device State*: Sets the device state metadata property that the device reports. This command is useful for testing back-end logic.

You can add more simulated devices to the solution that emit the same telemetry and respond to the same methods and commands.

In addition to responding to commands and methods, the solution uses device twins. Devices use device twins to report property values to the solution back end. The solution dashboard uses device twins to set to new desired property values on devices. For example, during the firmware update process the simulated device reports the status of the update using reported properties.

## IoT Hub

In this preconfigured solution, the IoT Hub instance corresponds to the *Cloud Gateway* in a typical IoT solution architecture.

An IoT hub receives telemetry from the devices at a single endpoint. An IoT hub also maintains device-specific endpoints where each device can retrieve the commands that are sent to it.

The IoT hub makes the received telemetry available through the service-side telemetry read endpoint.

The device management capability of IoT Hub enables you to manage your device properties from the solution portal and schedule jobs that perform operations such as:

- Rebooting devices
- Changing device states
- Firmware updates

## Azure Stream Analytics

The preconfigured solution uses three Azure Stream Analytics (ASA) jobs to filter the telemetry stream from the devices:

- *DeviceInfo job* - outputs data to an Event hub that routes device registration-specific messages to the solution device registry. This device registry is an Azure Cosmos DB database. These messages are sent when a device first connects or in response to a **Change device state** command.
- *Telemetry job* - sends all raw telemetry to Azure blob storage for cold storage and calculates telemetry aggregations that display in the solution dashboard.
- *Rules job* - filters the telemetry stream for values that exceed any rule thresholds and outputs the data to an Event hub. When a rule fires, the solution portal dashboard view displays this event as a new row in the alarm history table. These rules can also trigger an action based on the settings defined on the **Rules** and **Actions** views in the solution portal.

In this preconfigured solution, the ASA jobs form part of to the **IoT solution back end** in a typical IoT solution architecture.

## Event processor

In this preconfigured solution, the event processor forms part of the **IoT solution back end** in a typical IoT solution architecture.

The **DeviceInfo** and **Rules** ASA jobs send their output to Event hubs for delivery to other back-end services. The solution uses an EventProcessorHost instance, running in a WebJob, to read the messages from these Event hubs. The **EventProcessorHost** uses:

- The **DeviceInfo** data to update the device data in the Cosmos DB database.
- The **Rules** data to invoke the Logic app and update the alerts display in the solution portal.

## Device identity registry, device twin, and Cosmos DB

Every IoT hub includes a device identity registry that stores device keys. IoT Hub uses this information authenticate devices - a device must be registered and have a valid key before it can connect to the hub.

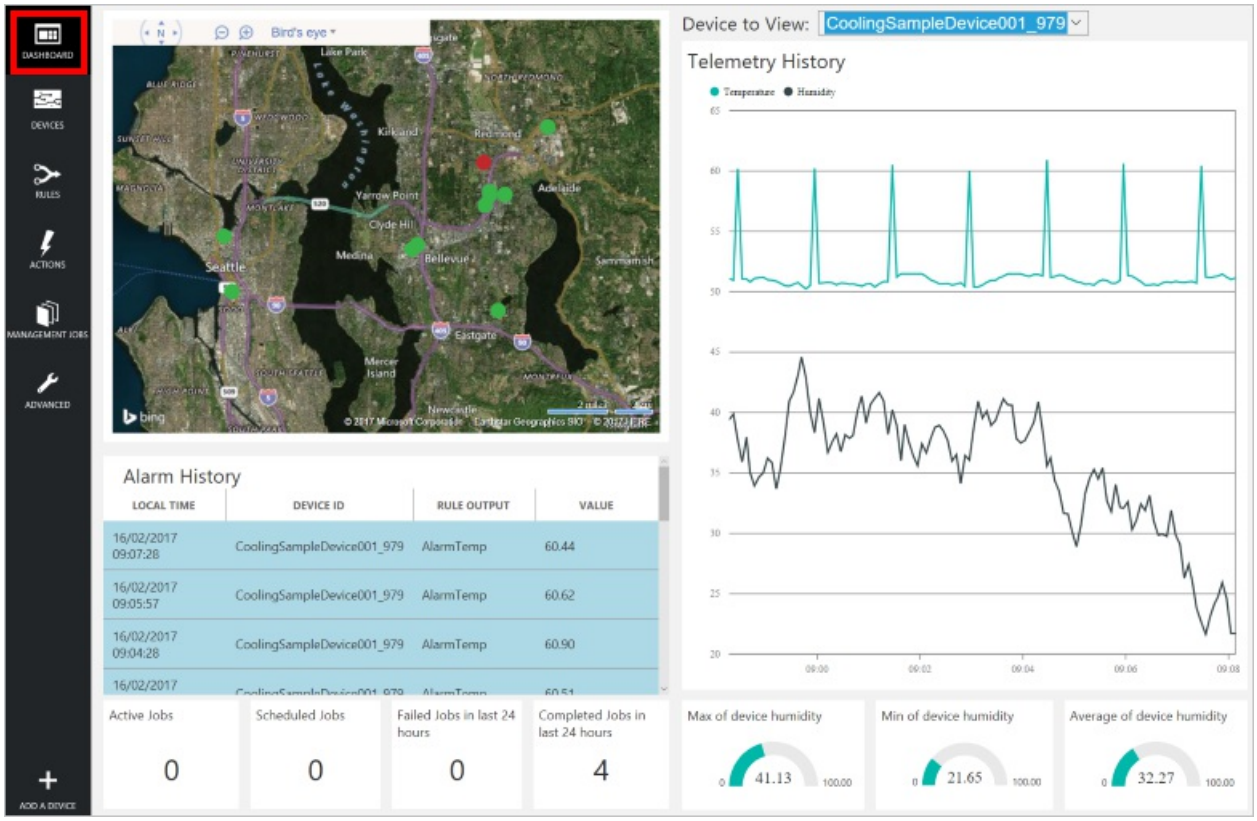
A device twin is a JSON document managed by the IoT Hub. A device twin for a device contains:

- Reported properties sent by the device to the hub. You can view these properties in the solution portal.
- Desired properties that you want to send to the device. You can set these properties in the solution portal.
- Tags that exist only in the device twin and not on the device. You can use these tags to filter lists of devices in the solution portal.

This solution uses device twins to manage device metadata. The solution also uses a Cosmos DB database to store additional solution-specific device data such as the commands supported by each device and the command history.

The solution must also keep the information in the device identity registry synchronized with the contents of the Cosmos DB database. The **EventProcessorHost** uses the data from **DeviceInfo** stream analytics job to manage the synchronization.

## Solution portal




The solution portal is a web-based UI that is deployed to the cloud as part of the preconfigured solution. It enables you to:

- View telemetry and alarm history in a dashboard.
- Provision new devices.
- Manage and monitor devices.
- Send commands to specific devices.
- Invoke methods on specific devices.

- Manage rules and actions.
- Schedule jobs to run on one or more devices.

In this preconfigured solution, the solution portal forms part of the **IoT solution back end** and part of the **Processing and business connectivity** in the typical IoT solution architecture.

## Next steps

For more information about IoT solution architectures, see Microsoft Azure IoT services: Reference Architecture.

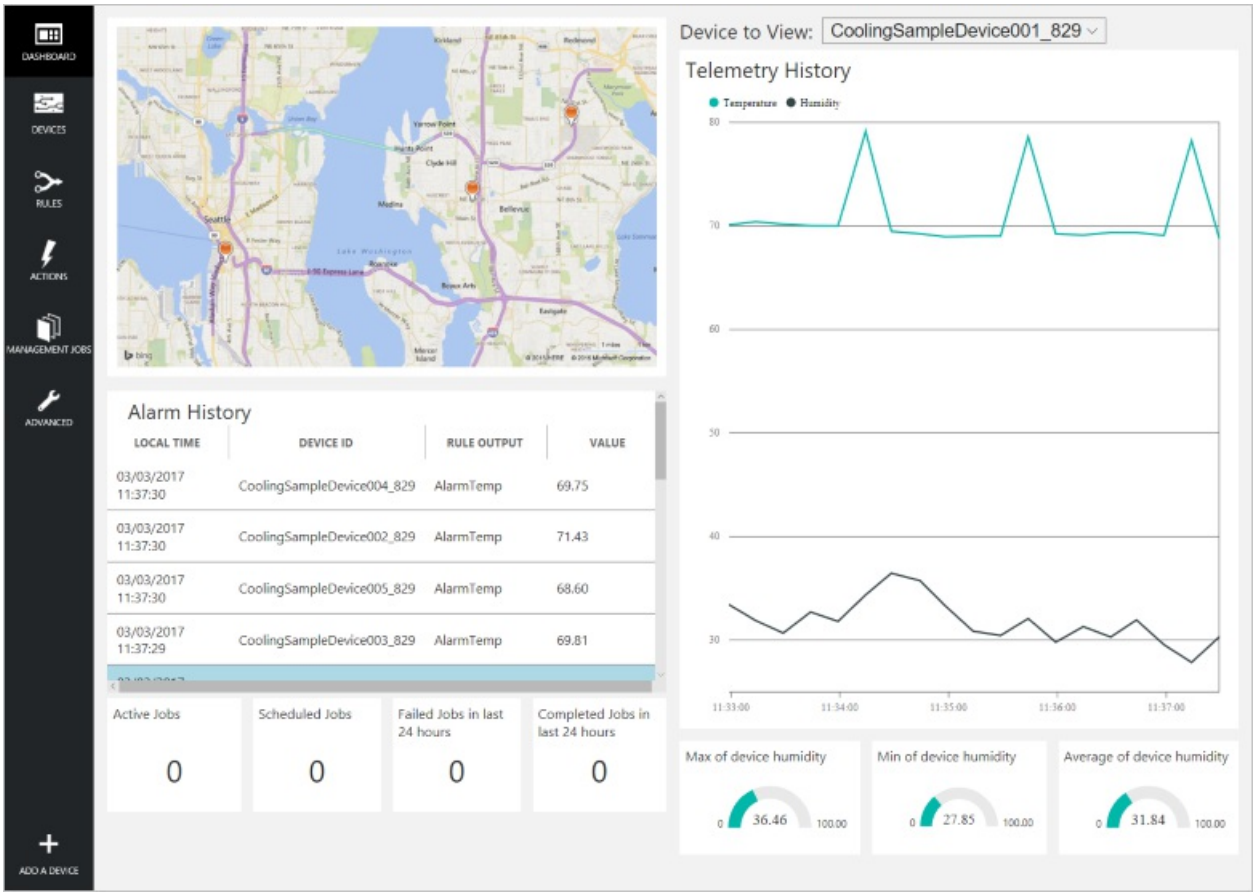
Now you know what a preconfigured solution is, you can get started by deploying the *remote monitoring* preconfigured solution: Get started with the preconfigured solutions.

# Get started with the preconfigured solutions

8/24/2017 • 14 min to read • Edit Online

Azure IoT Suite preconfigured solutions combine multiple Azure IoT services to deliver end-to-end solutions that implement common IoT business scenarios. The *remote monitoring* preconfigured solution connects to and monitors your devices. You can use the solution to analyze the stream of data from your devices and to improve business outcomes by making processes respond automatically to that stream of data.

This tutorial shows you how to provision the remote monitoring preconfigured solution. It also walks you through the basic features of the preconfigured solution. You can access many of these features from the solution *dashboard* that deploys as part of the preconfigured solution:




To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click + to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.

5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

## Scenario overview

When you deploy the remote monitoring preconfigured solution, it is prepopulated with resources that enable you to step through a common remote monitoring scenario. In this scenario, several devices connected to the solution are reporting unexpected temperature values. The following sections show you how to:

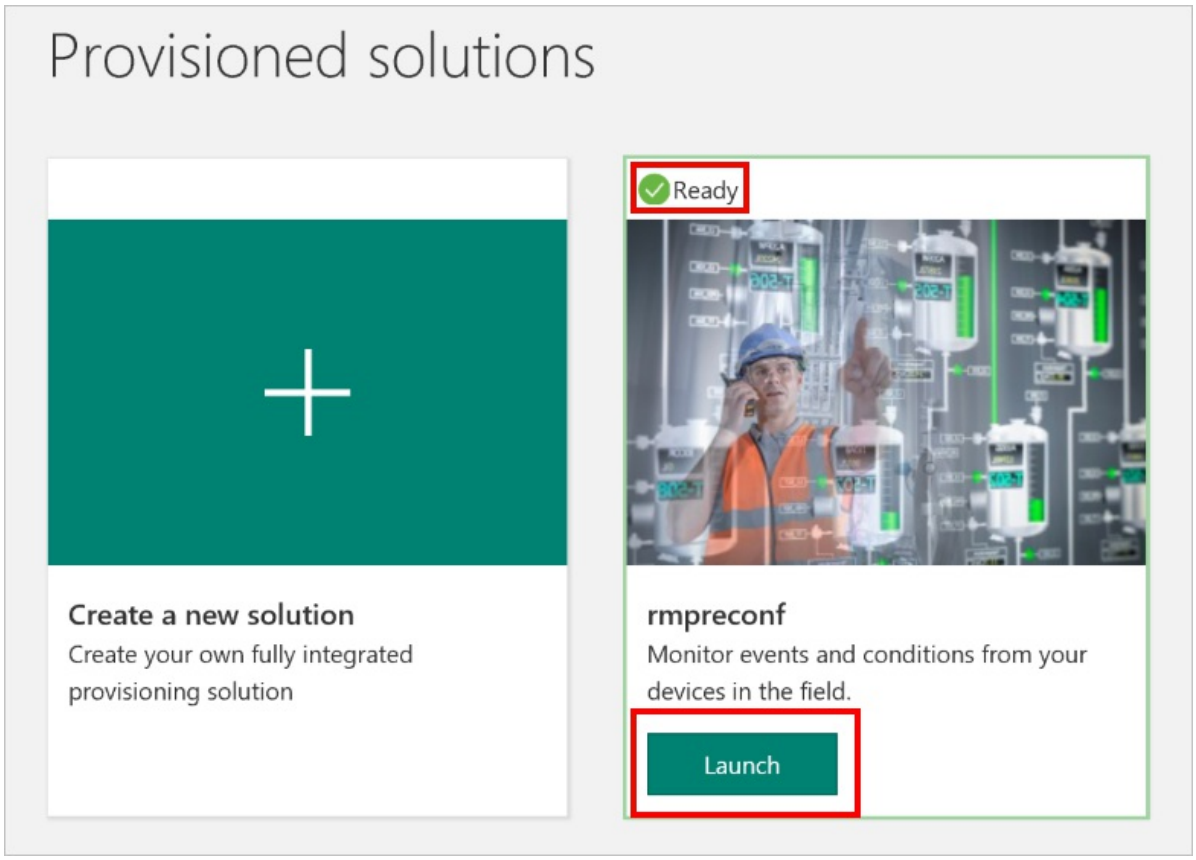
- Identify the devices sending unexpected temperature values.
- Configure these devices to send more detailed telemetry.
- Fix the problem by updating the firmware on these devices.
- Verify that your action has resolved the issue.

A key feature of this scenario is that you can perform all these actions remotely from the solution dashboard. You do not need physical access to the devices.

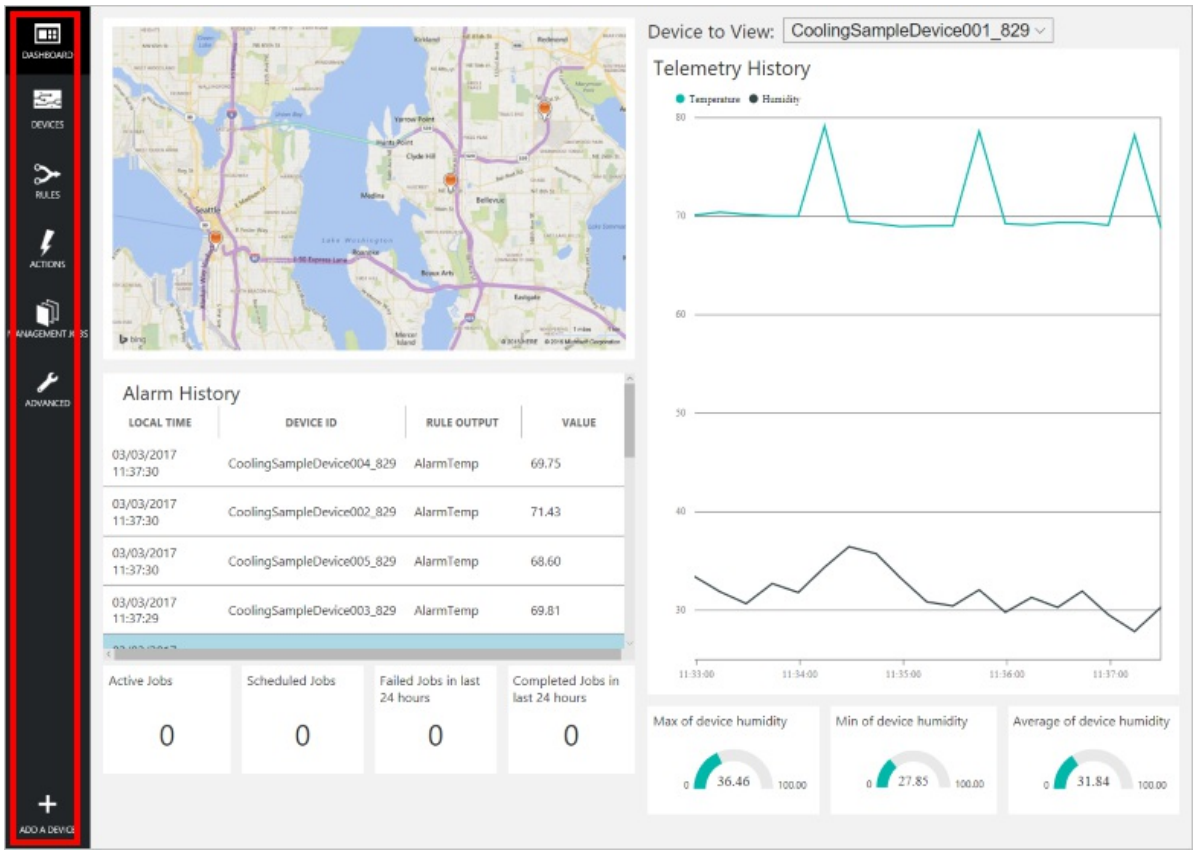
## View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and configure rules.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.




The dashboard displays the following information:

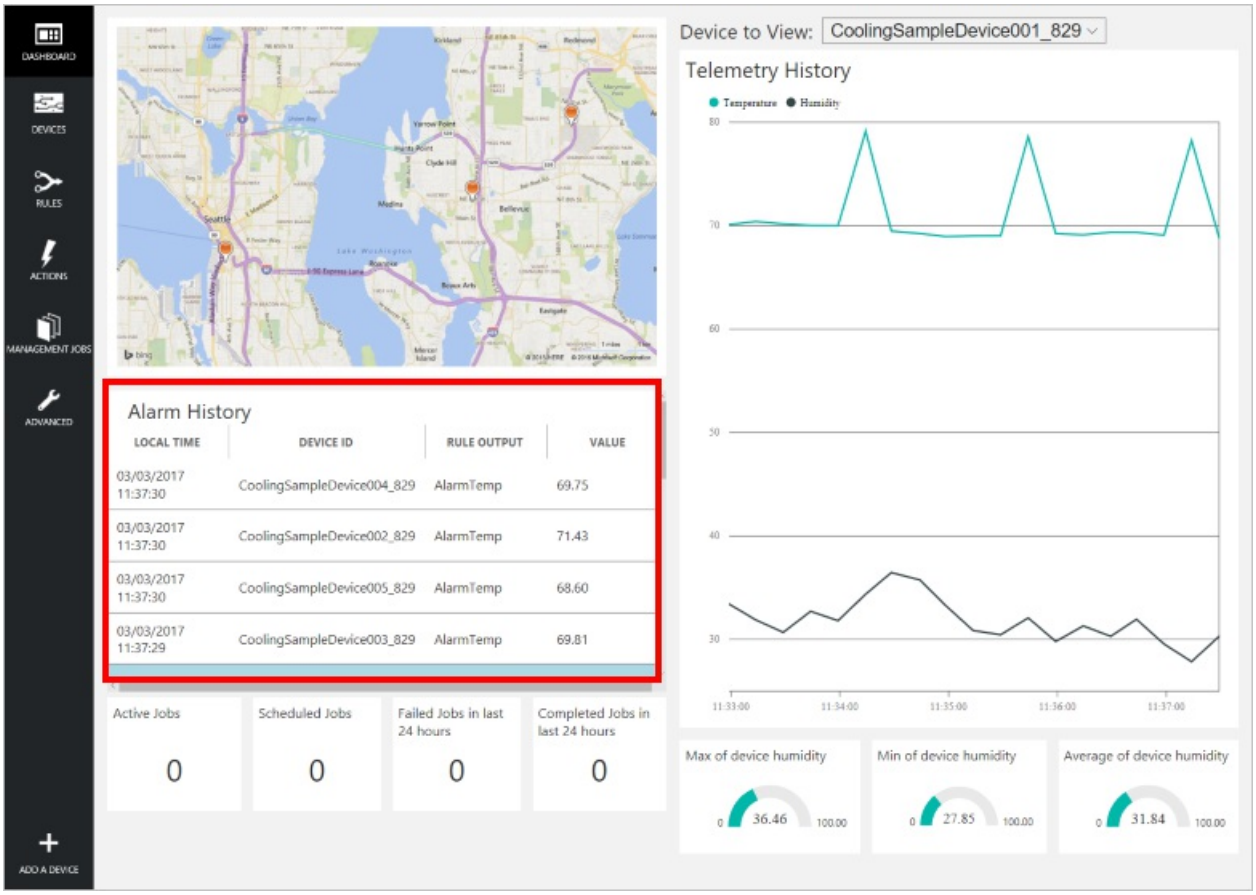
- A map that displays the location of each device connected to the solution. When you first run the solution, there are 25 simulated devices. The simulated devices are implemented as Azure WebJobs, and the solution uses the Bing Maps API to plot information on the map. See the FAQ to learn how to make the map dynamic.
- A **Telemetry History** panel that plots humidity and temperature telemetry from a selected device in near real

time and displays aggregate data such as maximum, minimum, and average humidity.

- An **Alarm History** panel that shows recent alarm events when a telemetry value has exceeded a threshold. You can define your own alarms in addition to the examples created by the preconfigured solution.
- A **Jobs** panel that displays information about scheduled jobs. You can schedule your own jobs on **Management jobs** page.

## View alarms

The alarm history panel shows you that five devices are reporting higher than expected telemetry values.

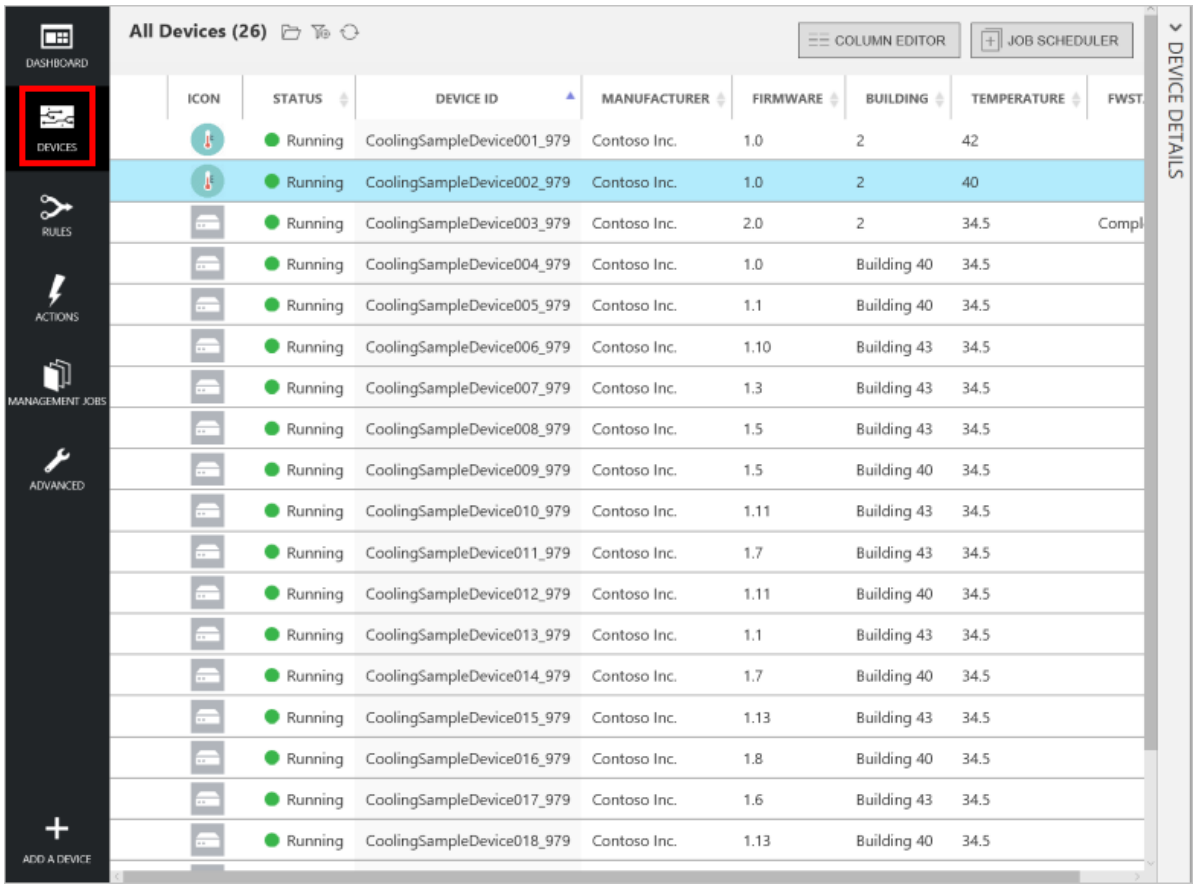



> **NOTE**
>
> These alarms are generated by a rule that is included in the preconfigured solution. This rule generates an alert when the temperature value sent by a device exceeds 60. You can define your own rules and actions by choosing Rules and Actions in the left-hand menu.

## View devices

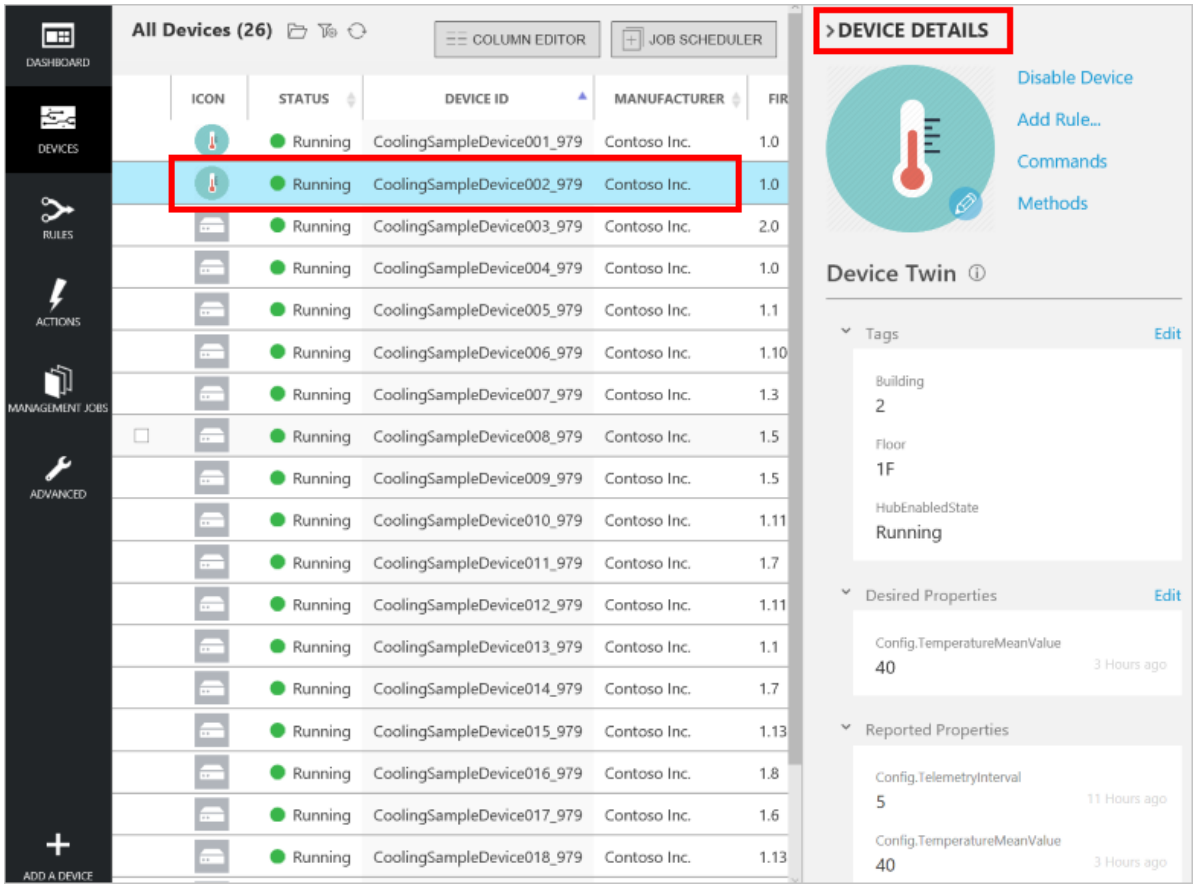
The *devices* list shows all the registered devices in the solution. From the device list you can view and edit device metadata, add or remove devices, and invoke methods on devices. You can filter and sort the list of devices in the device list. You can also customize the columns shown in the device list.

1. Choose **Devices** to show the device list for this solution.




2. The device list initially shows 25 simulated devices created by the provisioning process. You can add additional simulated and physical devices to the solution.

3. To view the details of a device, choose a device in the device list.




The **Device Details** panel contains six sections:

- A collection of links that enable you to customize the device icon, disable the device, add a rule, invoke a

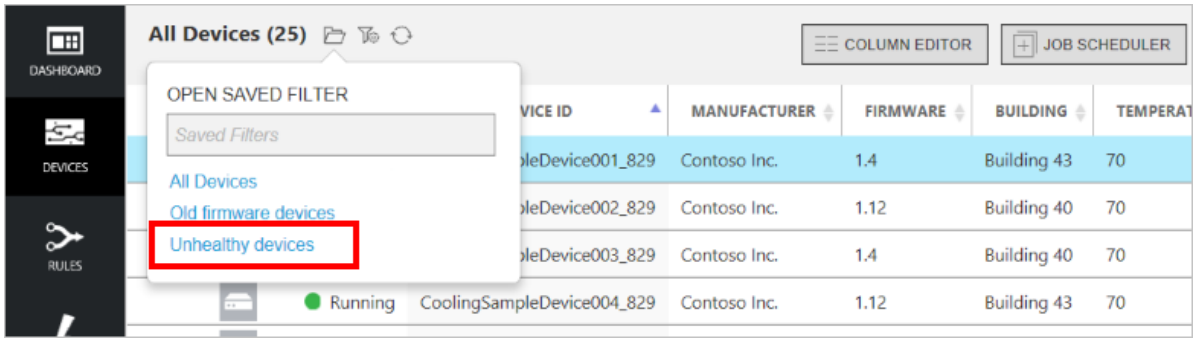
method, or send a command. For a comparison of commands (device-to-cloud messages) and methods (direct methods), see Cloud-to-device communications guidance.

- The **Device Twin - Tags** section enables you to edit tag values for the device. You can display tag values in the device list and use tag values to filter the device list.
- The **Device Twin - Desired Properties** section enables you to set property values to be sent to the device.
- The **Device Twin - Reported Properties** section shows property values sent from the device.
- The **Device Properties** section shows information from the identity registry such as the device id and authentication keys.
- The **Recent Jobs** section shows information about any jobs that have recently targeted this device.

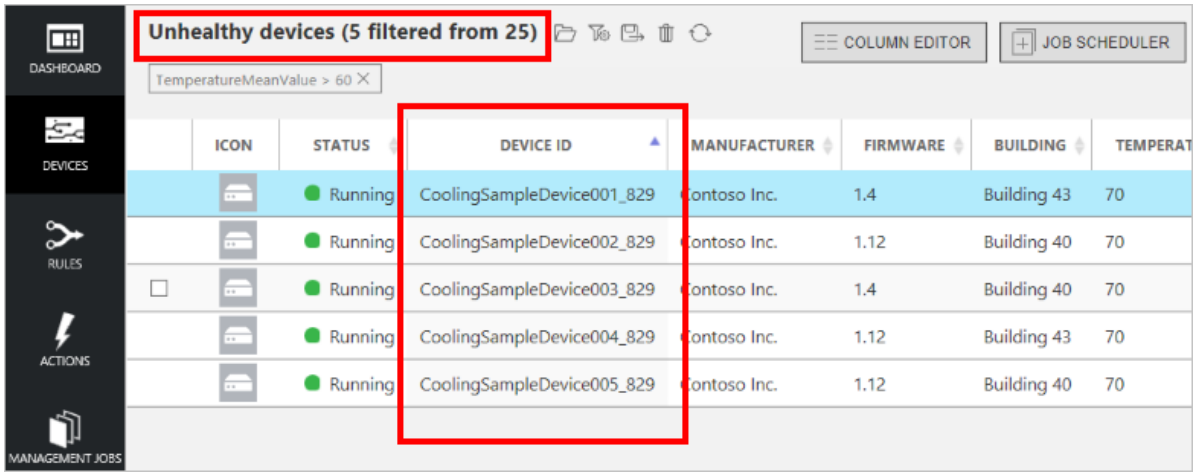
## Filter the device list

You can use a filter to display only those devices that are sending unexpected temperature values. The remote monitoring preconfigured solution includes the **Unhealthy devices** filter to show devices with a mean temperature value greater than 60. You can also create your own filters.

1. Choose **Open saved filter** to display a list of available filters. Then choose **Unhealthy devices** to apply the filter:




2. The device list now shows only devices with a mean temperature value greater than 60.



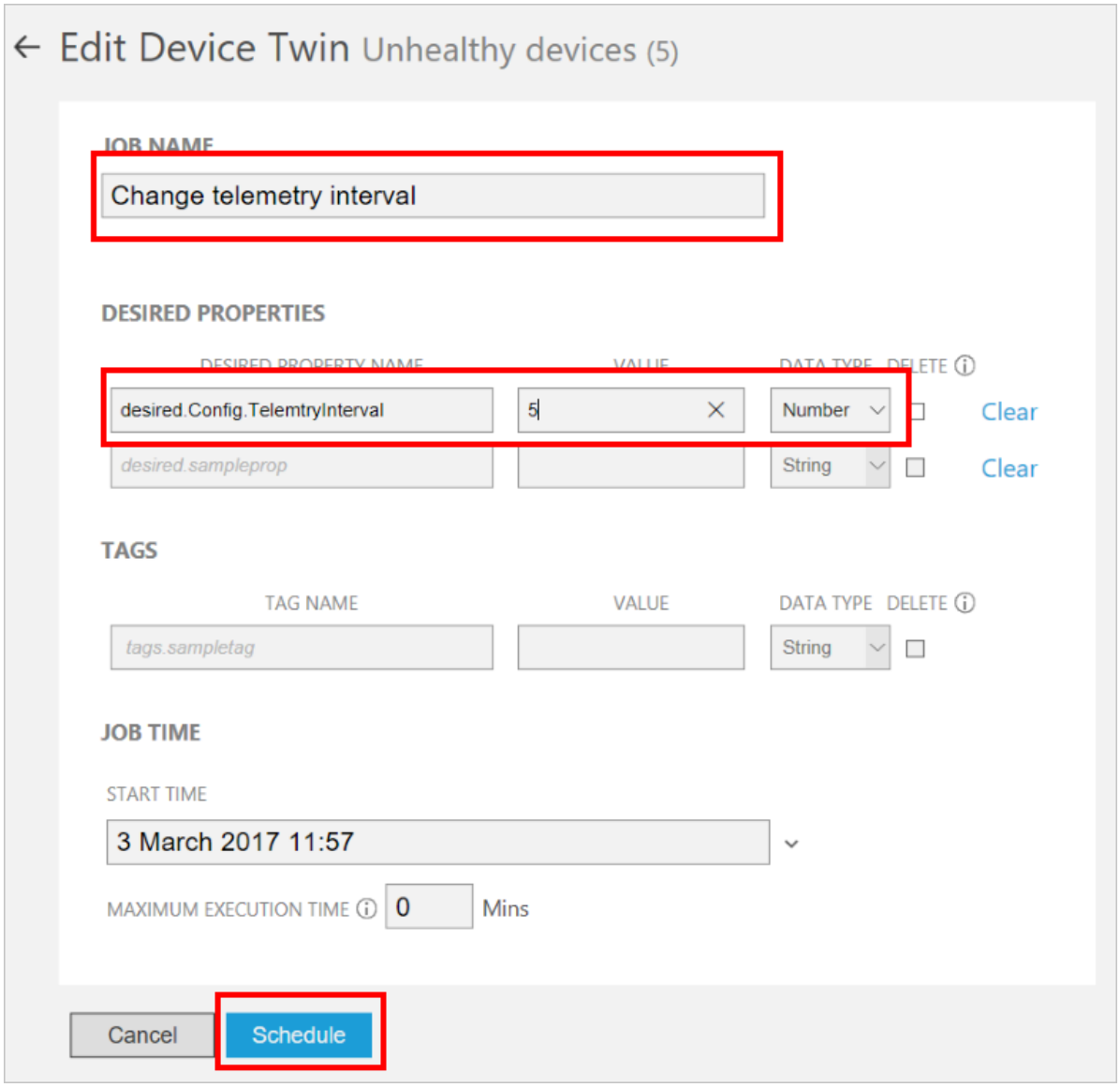

## Update desired properties

You have now identified a set of devices that may need remediation. However, you decide that the data frequency of 15 seconds is not sufficient for a clear diagnosis of the issue. Changing the telemetry frequency to five seconds to provide you with more data points to better diagnose the issue. You can push this configuration change to your remote devices from the solution portal. You can make the change once, evaluate the impact, and then act on the results.

Follow these steps to run a job that changes the **TelemetryInterval** desired property for the affected devices. When the devices receive the new **TelemetryInterval** property value, they change their configuration to send

telemetry every five seconds instead of every 15 seconds:

1. While you are showing the list of unhealthy devices in the device list, choose **Job Scheduler**, then **Edit Device Twin**.
2. Call the job **Change telemetry interval**.
3. Change the value of the **Desired Property** name **desired.Config.TelemetryInterval** to five seconds.
4. Choose **Schedule**.




5. You can monitor the progress of the job on the **Management Jobs** page in the portal.

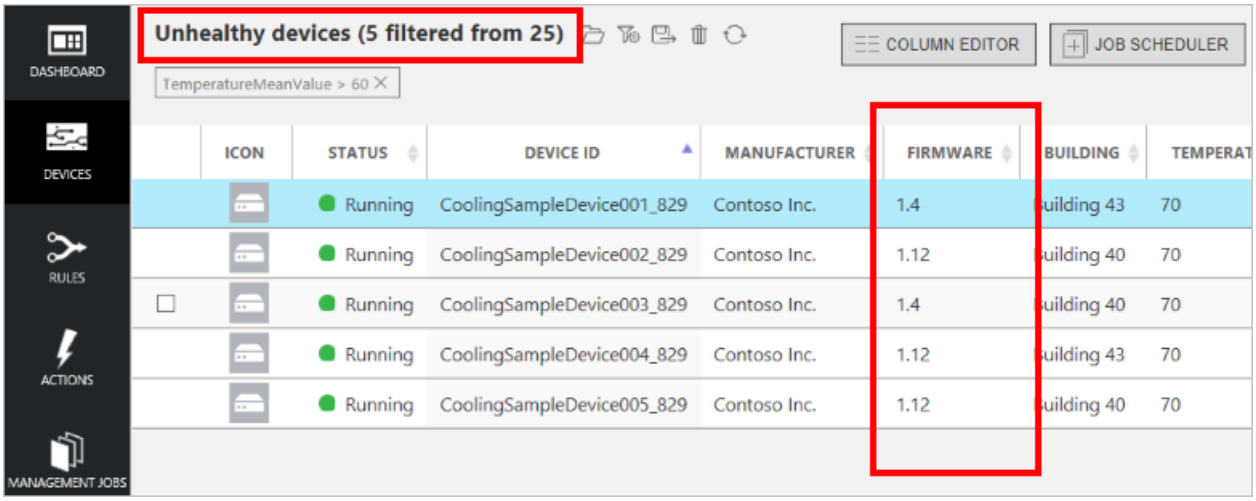
> **NOTE**
> If you want to change a desired property value for an individual device, use the **Desired Properties** section in the **Device Details** panel instead of running a job.

This job sets the value of the **TelemetryInterval** desired property in the device twin for all the devices selected by the filter. The devices retrieve this value from the device twin and update their behavior. When a device retrieves and processes a desired property from a device twin, it sets the corresponding reported value property.

## Invoke methods

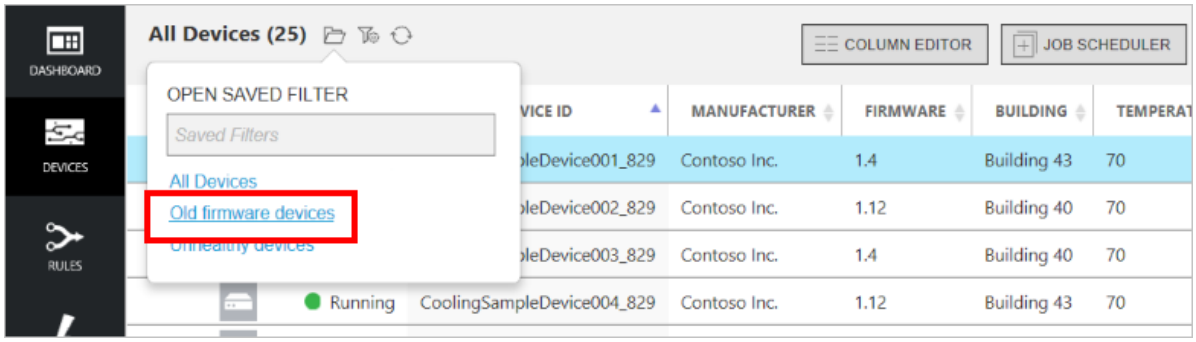
While the job runs, you notice in the list of unhealthy devices that all these devices have old (less than version 1.6)

firmware versions.

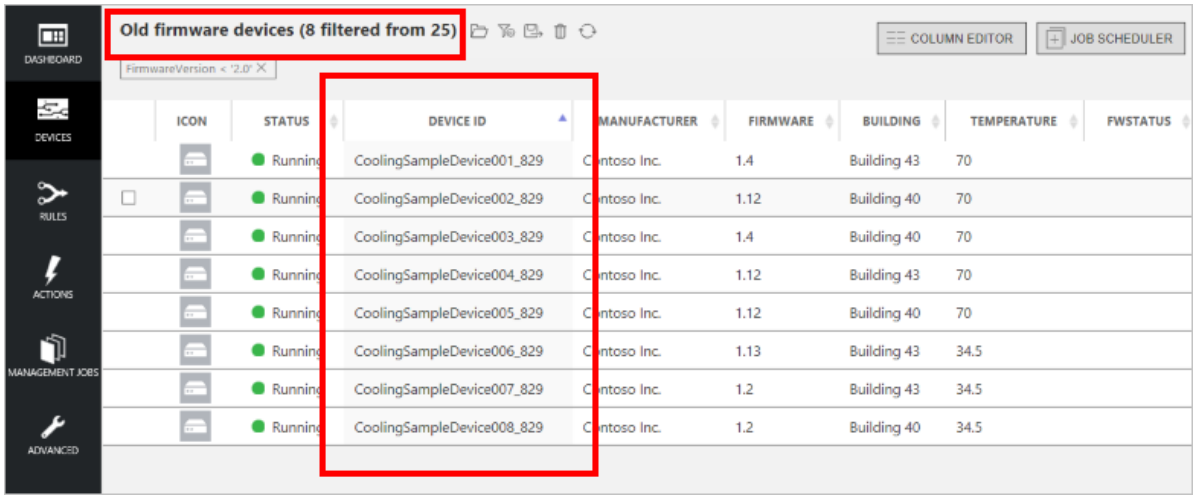



This firmware version may be the root cause of the unexpected temperature values because you know that other healthy devices were recently updated to version 2.0. You can use the built-in **Old firmware devices** filter to identify any devices with old firmware versions. From the portal, you can then remotely update all the devices still running old firmware versions:

1. Choose **Open saved filter** to display a list of available filters. Then choose **Old firmware devices** to apply the filter:



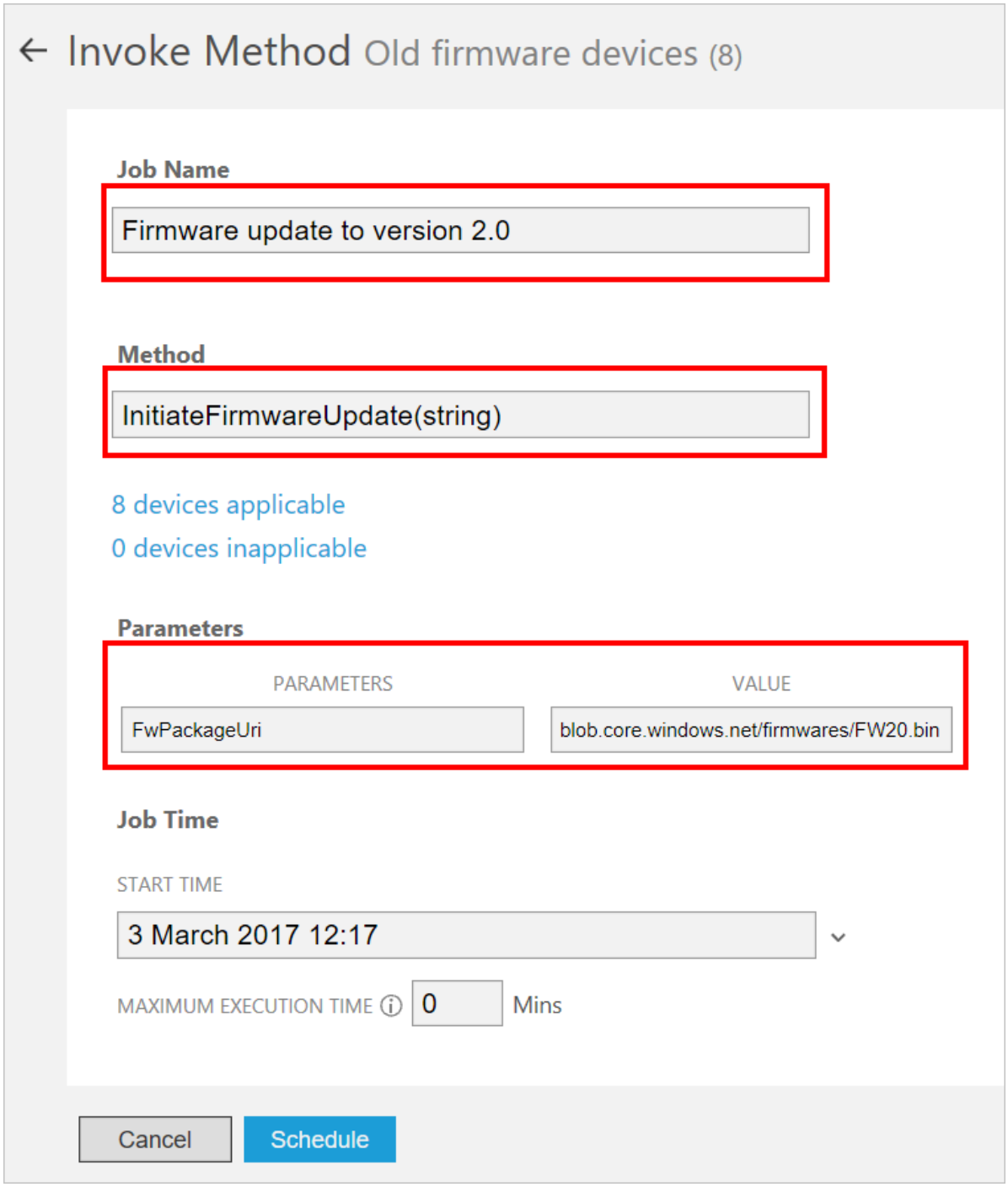

2. The device list now shows only devices with old firmware versions. This list includes the five devices identified by the **Unhealthy devices** filter and three additional devices:




3. Choose **Job Scheduler**, then **Invoke Method**.
4. Set **Job Name** to **Firmware update to version 2.0**.
5. Choose **InitiateFirmwareUpdate** as the **Method**.
6. Set the **FwPackageUri** parameter to

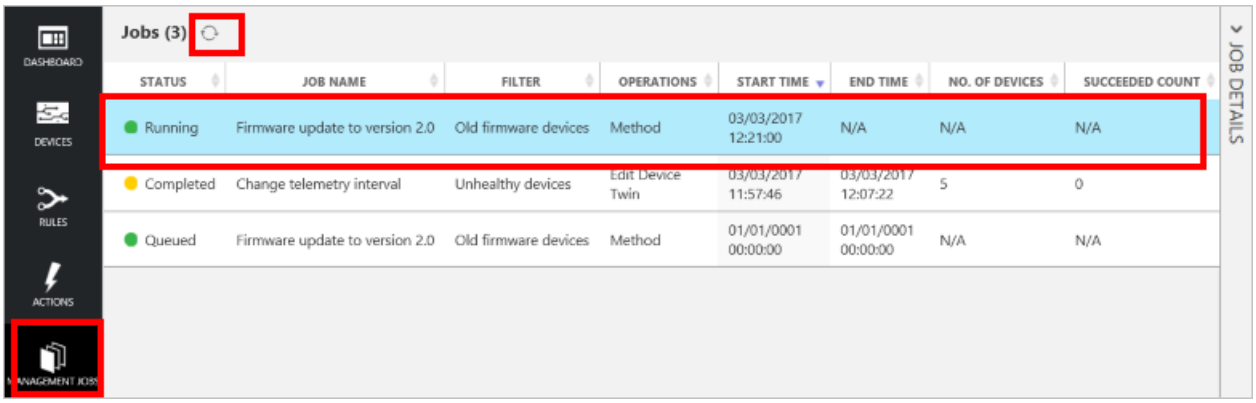
**https://iotrmassets.blob.core.windows.net/firmwares/FW20.bin**.

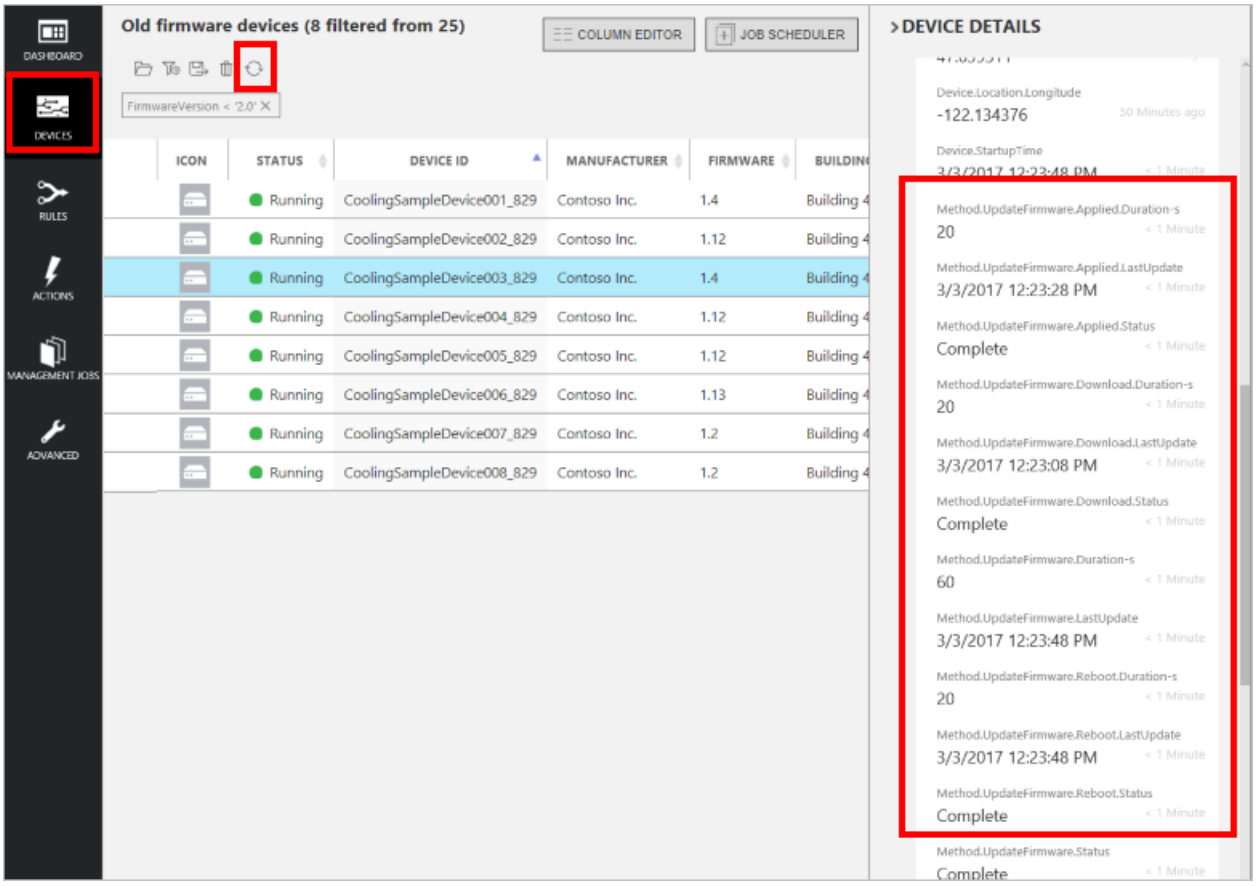
7. Choose **Schedule**. The default is for the job to run now.

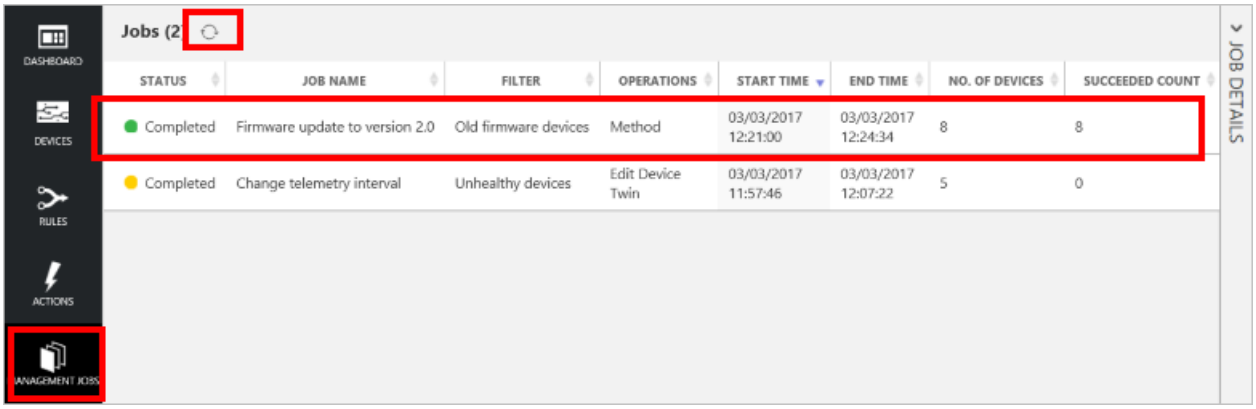



> **NOTE**
> If you want to invoke a method on an individual device, choose **Methods** in the **Device Details** panel instead of running a job.

This job invokes the **InitiateFirmwareUpdate** direct method on all the devices selected by the filter. Devices respond immediately to IoT Hub and then initiate the firmware update process asynchronously. The devices provide status information about the firmware update process through reported property values, as shown in the following screenshots. Choose the **Refresh** icon to update the information in the device and job lists:



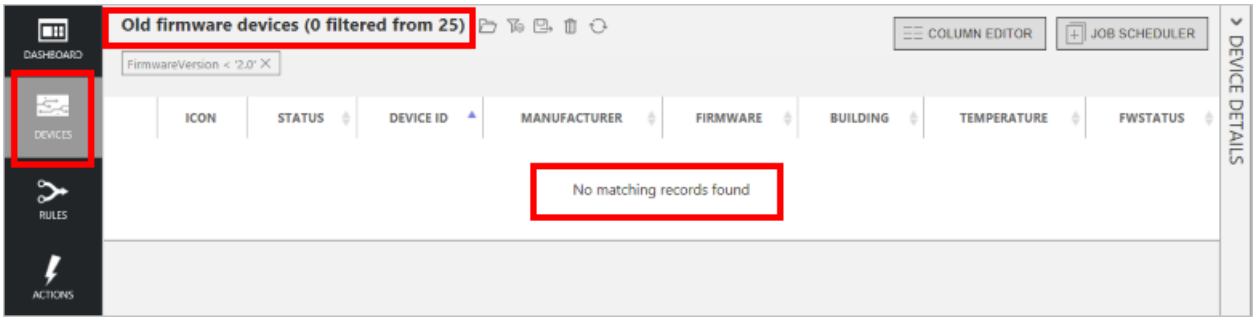

**NOTE**

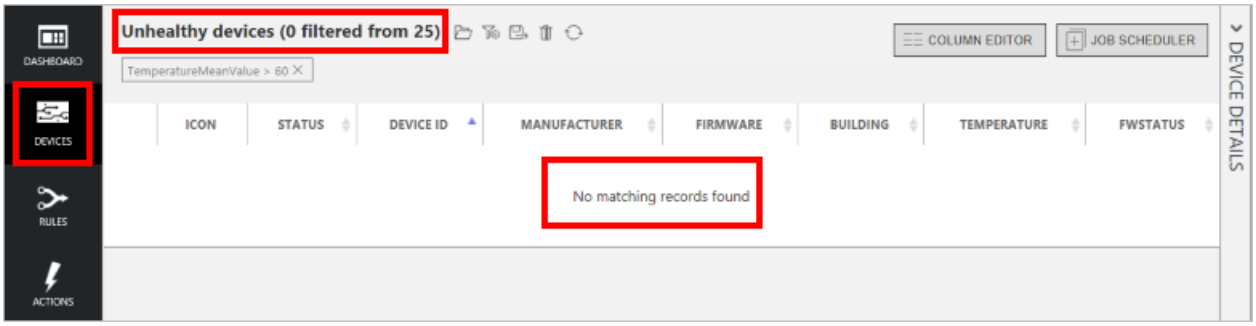
In a production environment, you can schedule jobs to run during a designated maintenance window.

## Scenario review

In this scenario, you identified a potential issue with some of your remote devices using the alarm history on the dashboard and a filter. You then used the filter and a job to remotely configure the devices to provide more

information to help diagnose the issue. Finally, you used a filter and a job to schedule maintenance on the affected devices. If you return to the dashboard, you can check that there are no longer any alarms coming from devices in your solution. You can use a filter to verify that the firmware is up-to-date on all the devices in your solution and that there are no more unhealthy devices:



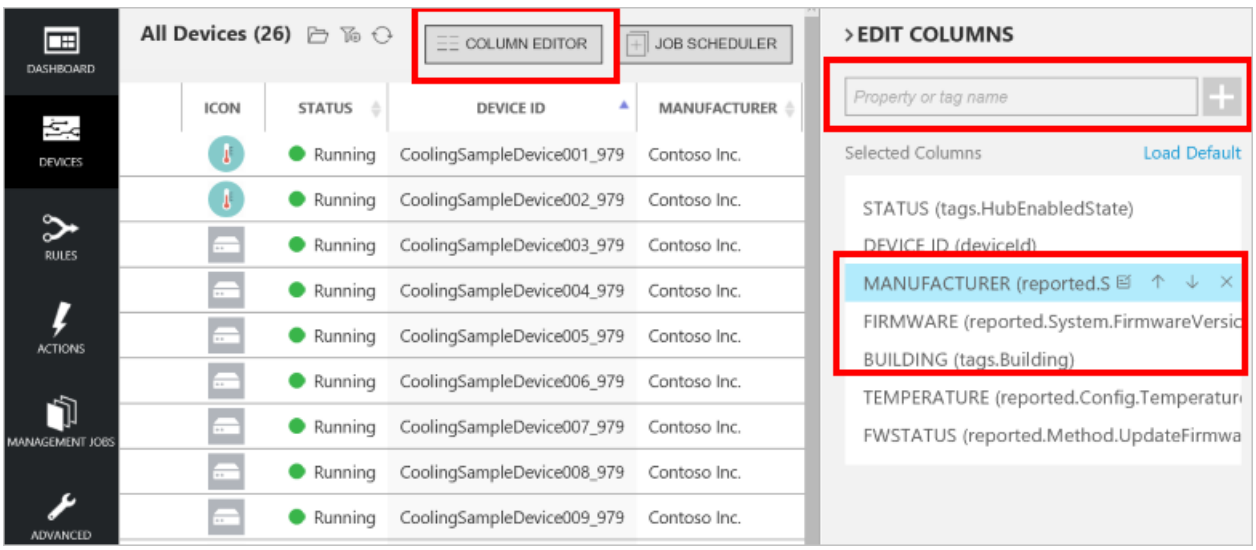




# Other features

The following sections describe some additional features of the remote monitoring preconfigured solution that are not described as part of the previous scenario.

### Customize columns

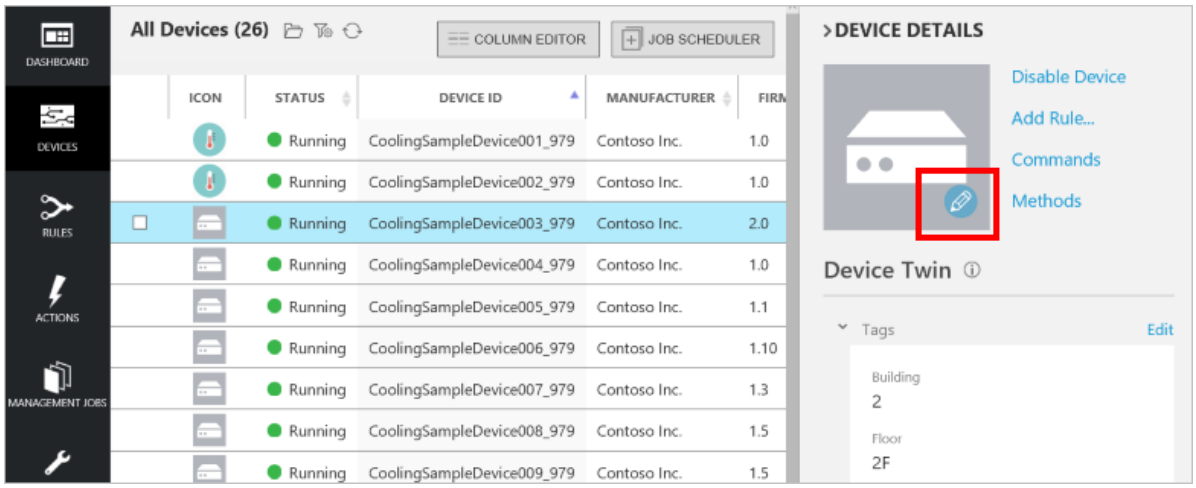
You can customize the information shown in the device list by choosing **Column editor**. You can add and remove columns that display reported property and tag values. You can also reorder and rename columns:



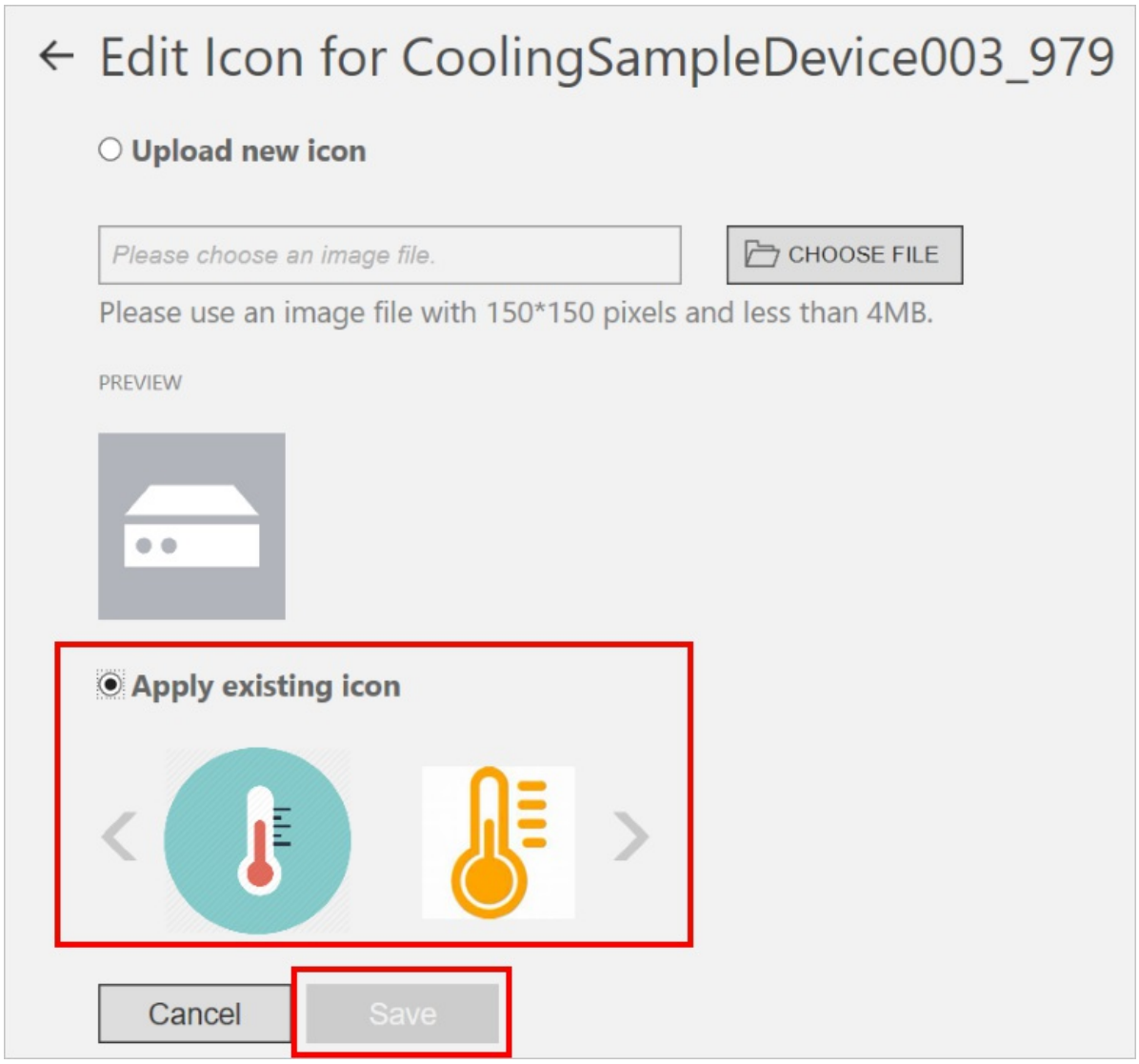

### Customize the device icon

You can customize the device icon displayed in the device list from the **Device Details** panel as follows:

1. Choose the pencil icon to open the **Edit image** panel for a device:




2. Either upload a new image or use one of the existing images and then choose **Save**:




3. The image you selected now displays in the **Icon** column for the device.

> **NOTE**
> The image is stored in blob storage. A tag in the device twin contains a link to the image in blob storage.

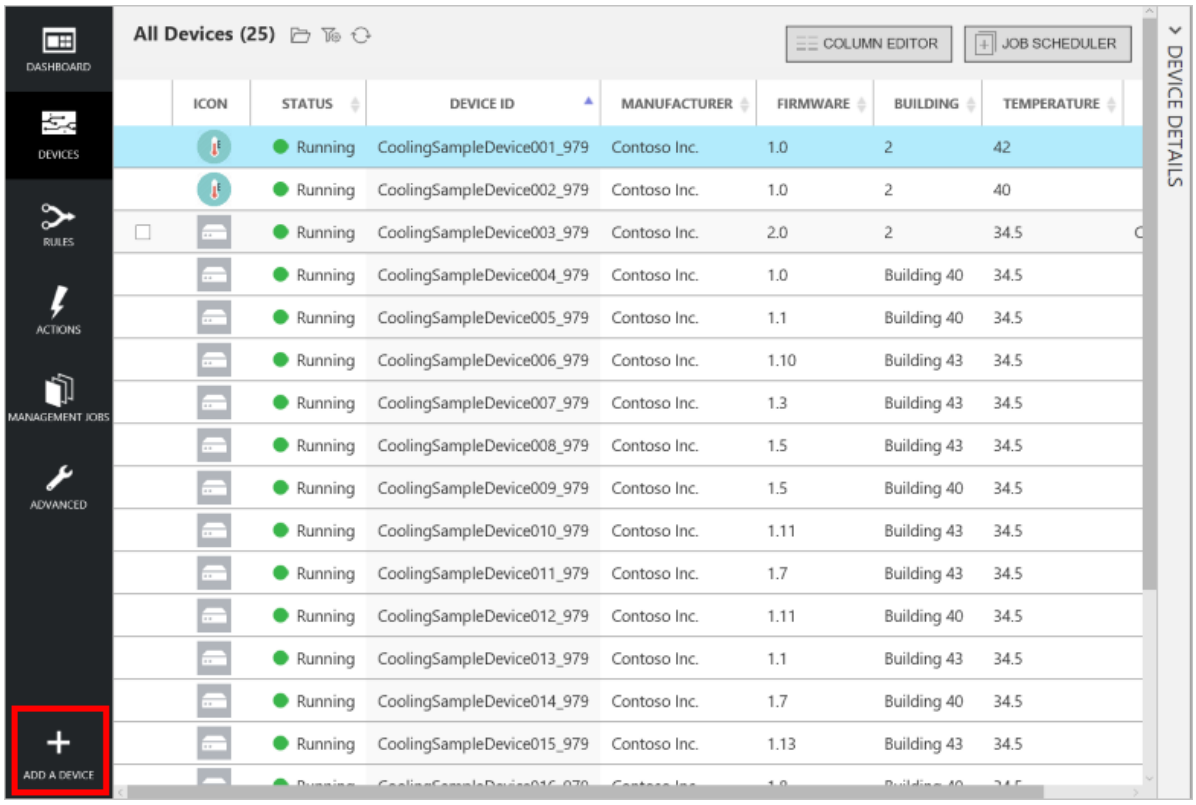
### Add a device

When you deploy the preconfigured solution, you automatically provision 25 sample devices that you can see in the device list. These devices are *simulated devices* running in an Azure WebJob. Simulated devices make it easy

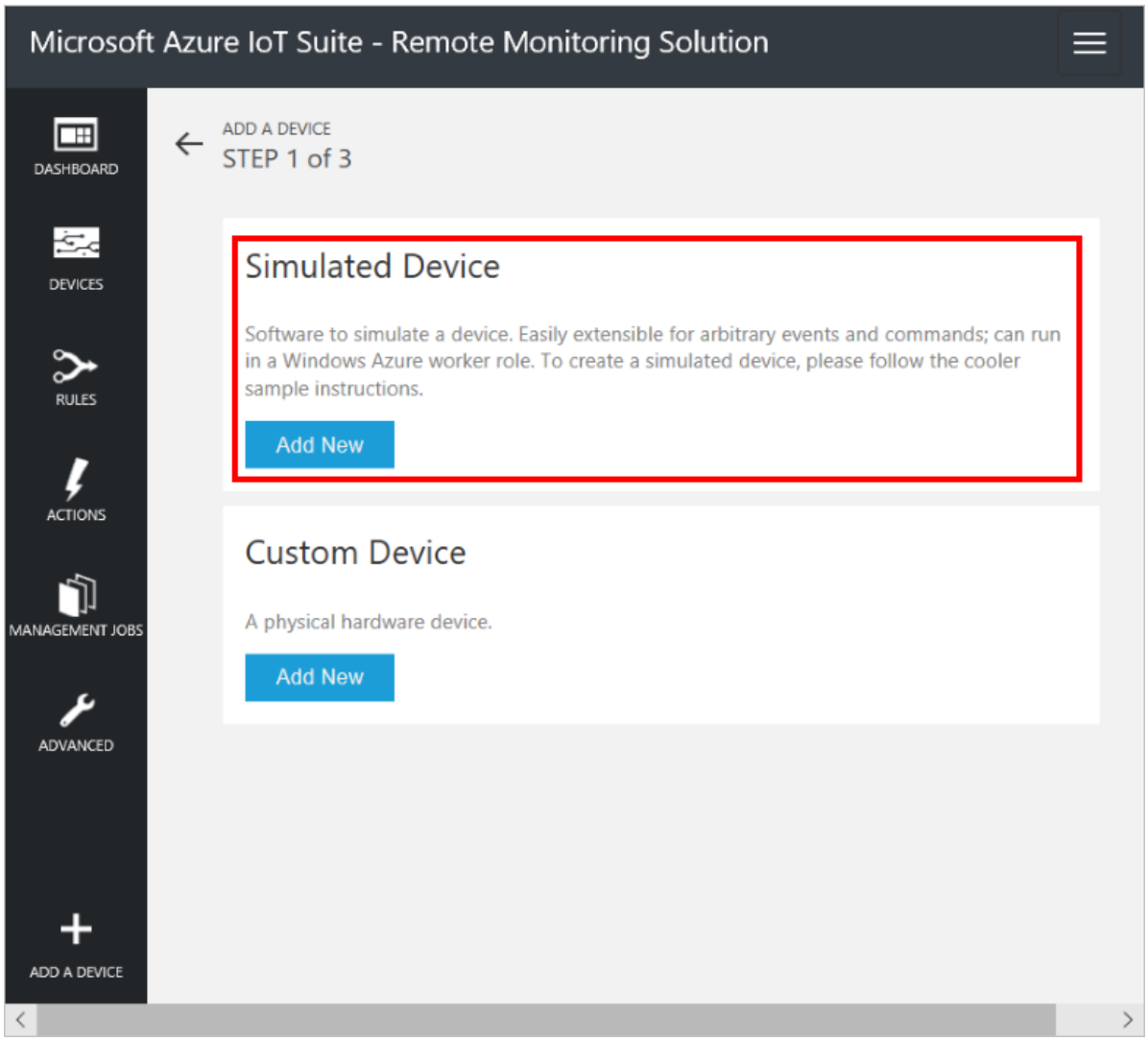
for you to experiment with the preconfigured solution without the need to deploy real, physical devices. If you do want to connect a real device to the solution, see the Connect your device to the remote monitoring preconfigured solution tutorial.

The following steps show you how to add a simulated device to the solution:

1. Navigate back to the device list.
2. To add a device, choose **+ Add A Device** in the bottom left corner.

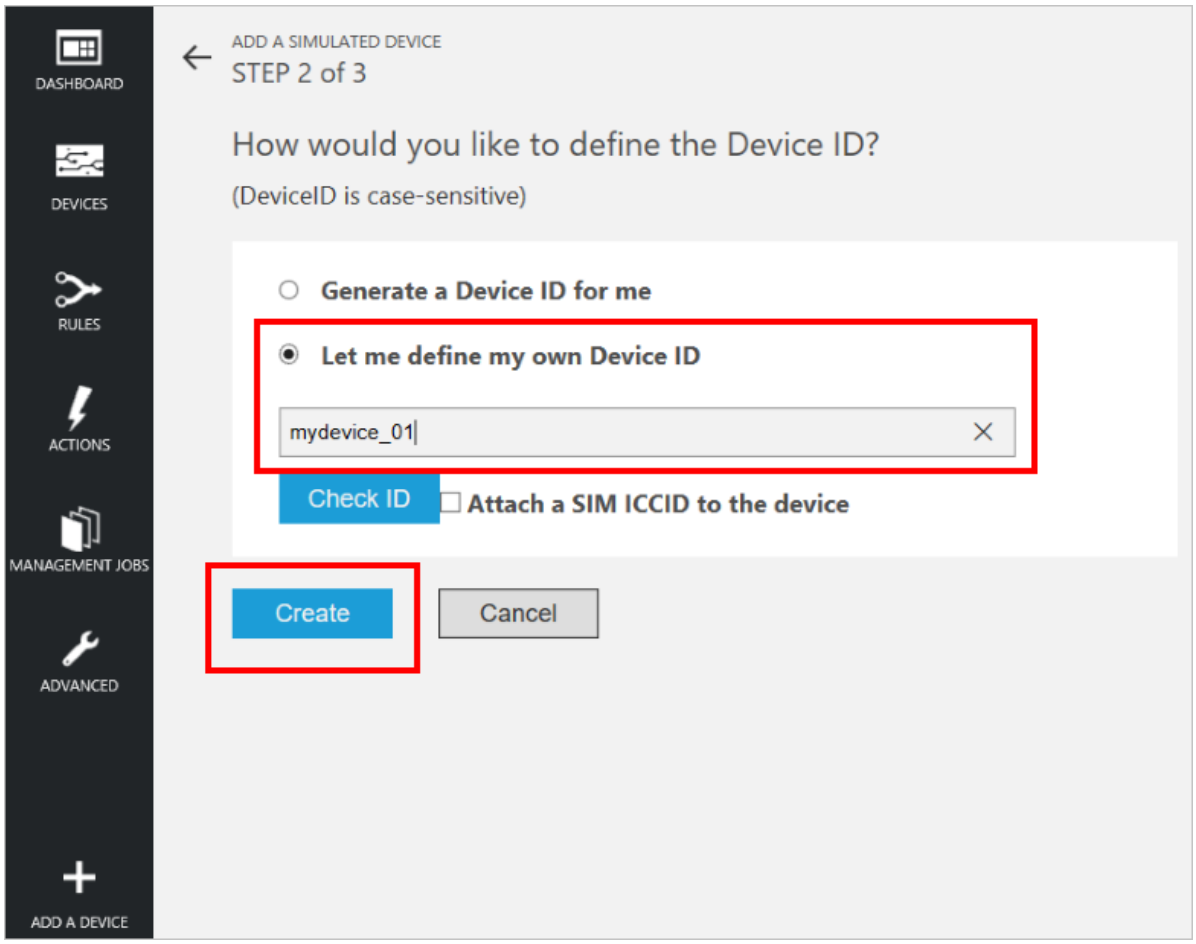



3. Choose **Add New** on the **Simulated Device** tile.



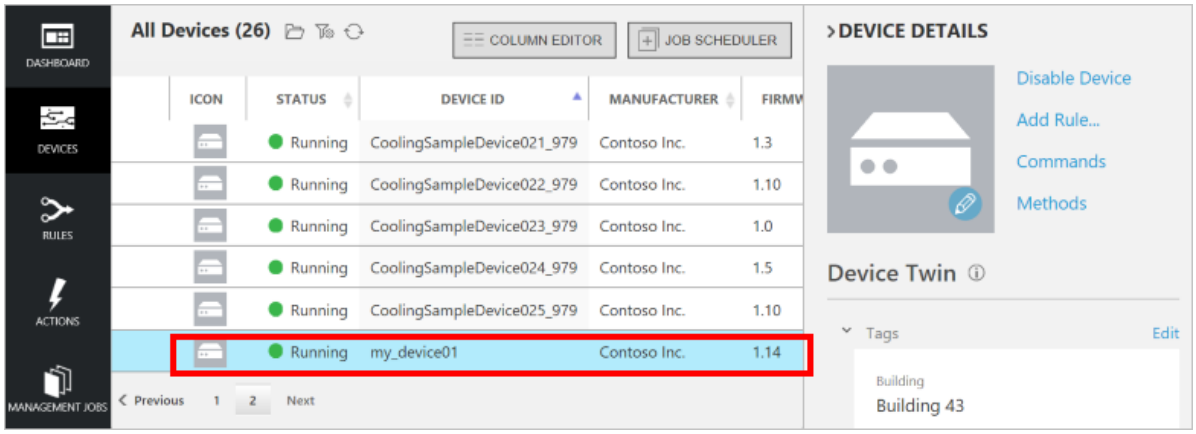

In addition to creating a new simulated device, you can also add a physical device if you choose to create a **Custom Device**. To learn more about connecting physical devices to the solution, see Connect your device to the IoT Suite remote monitoring preconfigured solution.

4. Select **Let me define my own Device ID**, and enter a unique device ID name such as **mydevice_01**.
5. Choose **Create**.

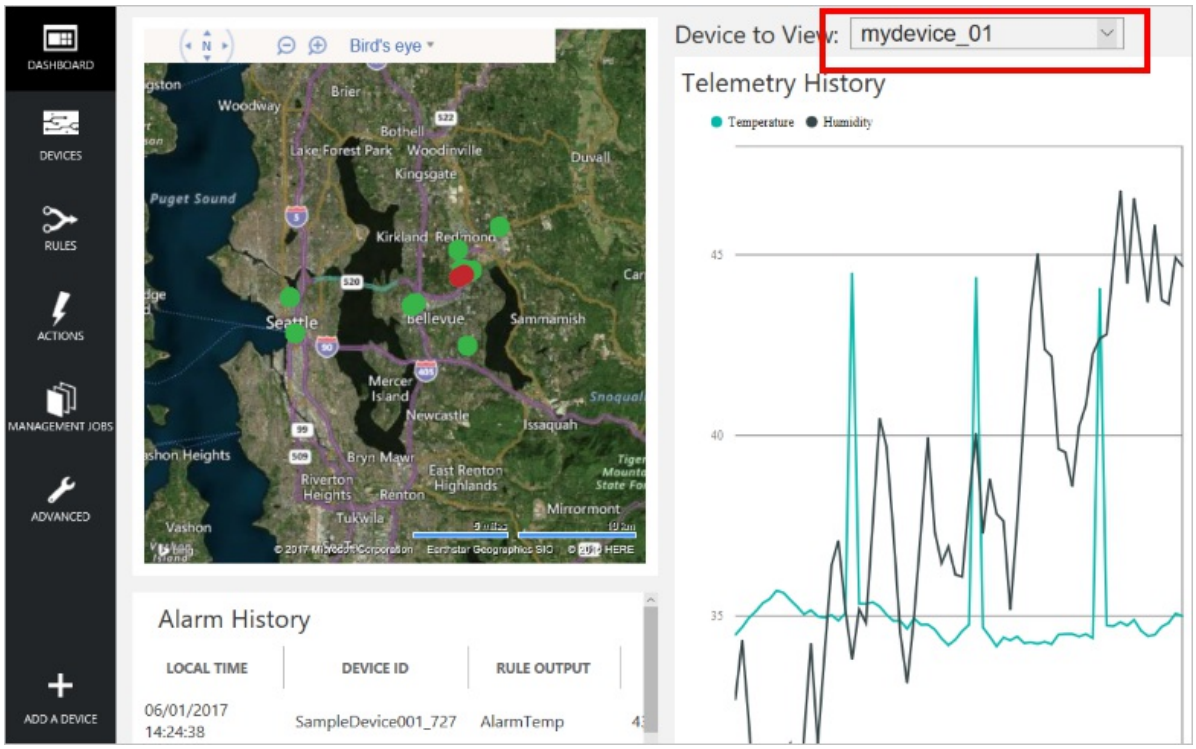



6. In step 3 of **Add a simulated device**, choose **Done** to return to the device list.

7. You can view your device **Running** in the device list.

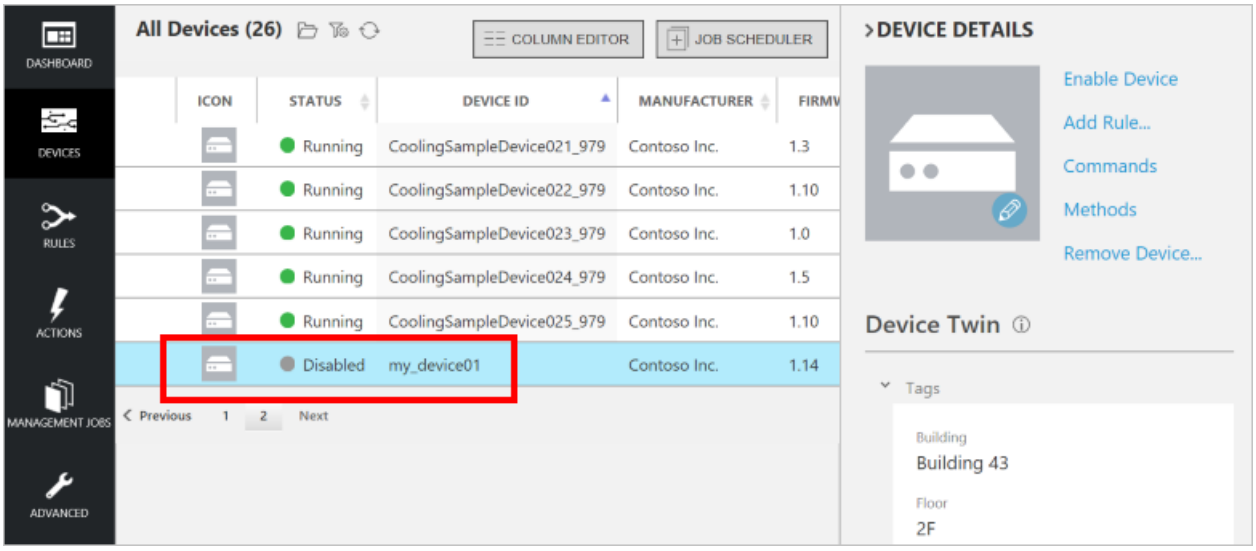



8. You can also view the simulated telemetry from your new device on the dashboard:




**Disable and delete a device**

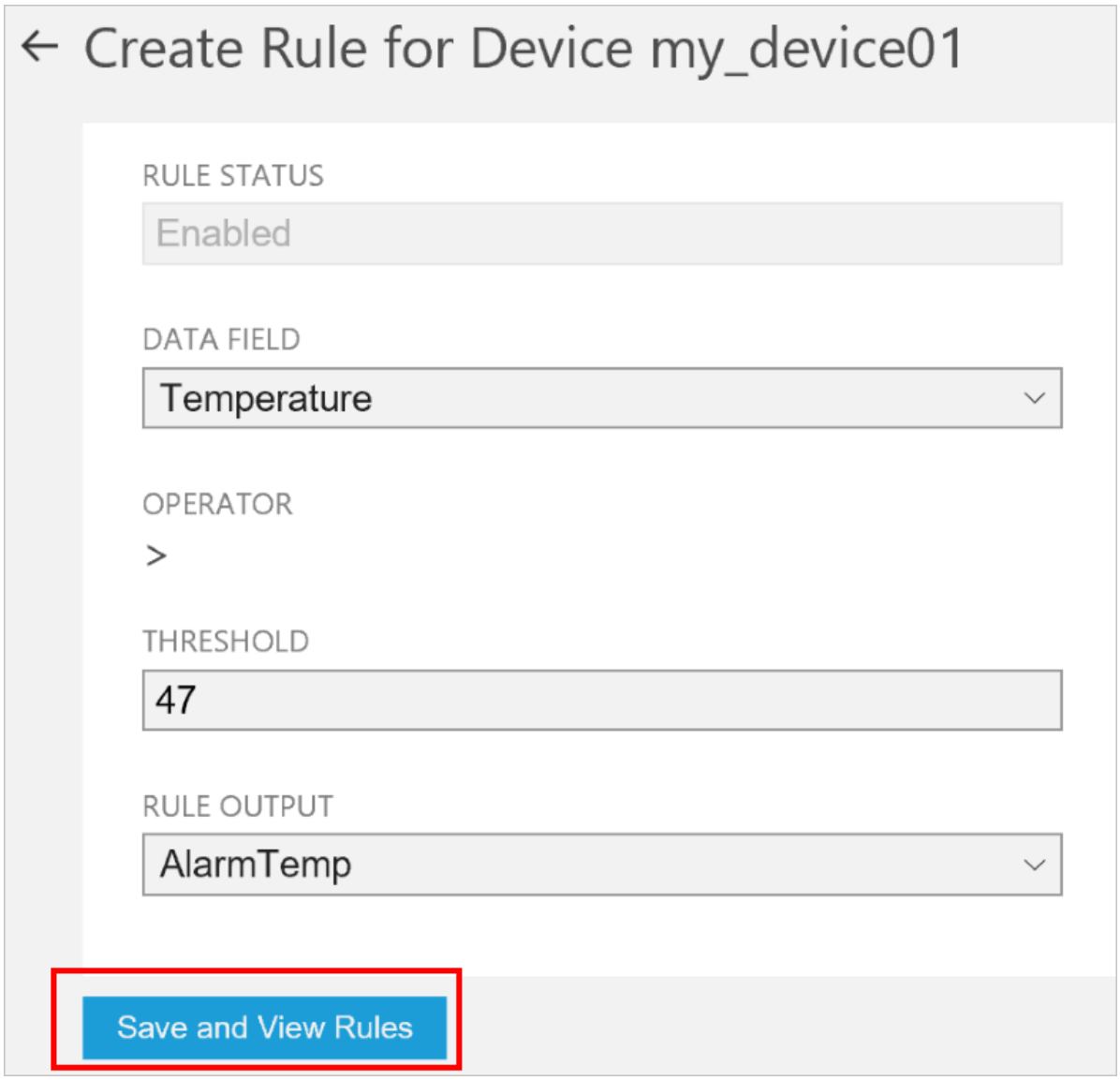
You can disable a device, and after it is disabled you can remove it:




**Add a rule**

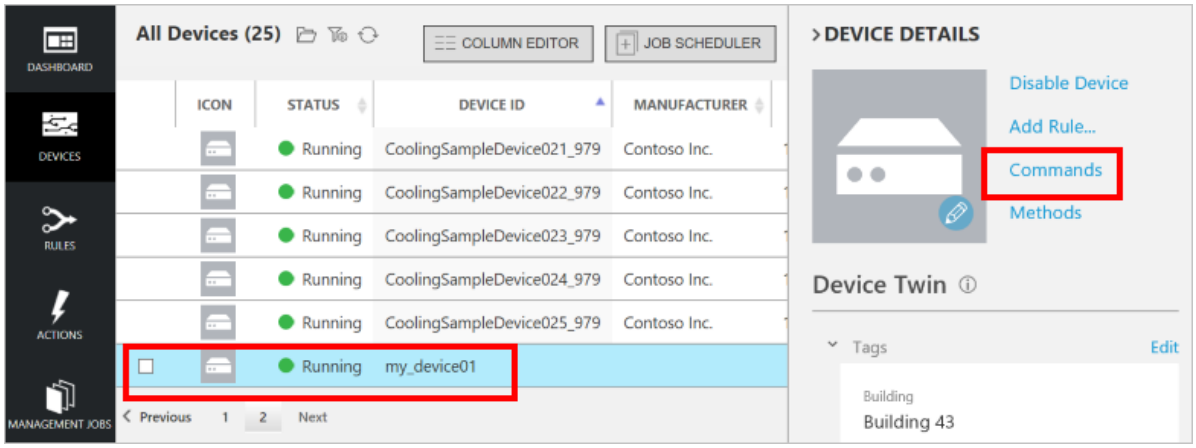
There are no rules for the new device you just added. In this section, you add a rule that triggers an alarm when the temperature reported by the new device exceeds 47 degrees. Before you start, notice that the telemetry history for the new device on the dashboard shows the device temperature never exceeds 45 degrees.

1. Navigate back to the device list.
2. To add a rule for the device, select your new device in the **Devices List**, and then choose **Add rule**.
3. Create a rule that uses **Temperature** as the data field and uses **AlarmTemp** as the output when the temperature exceeds 47 degrees:

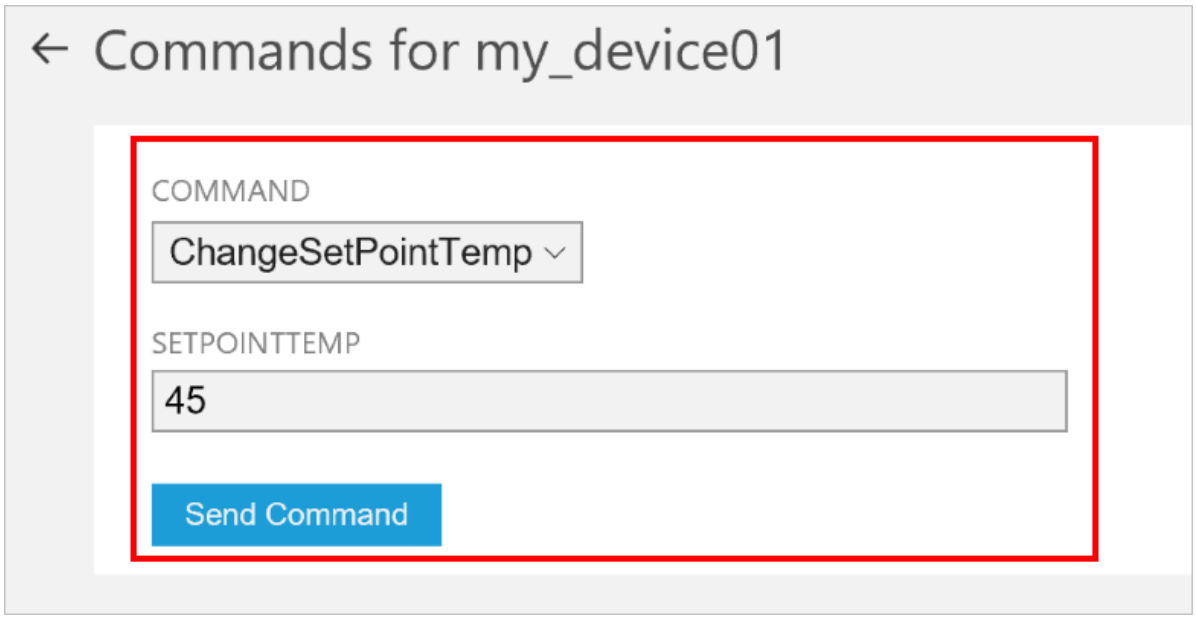



4. To save your changes, choose **Save and View Rules**.

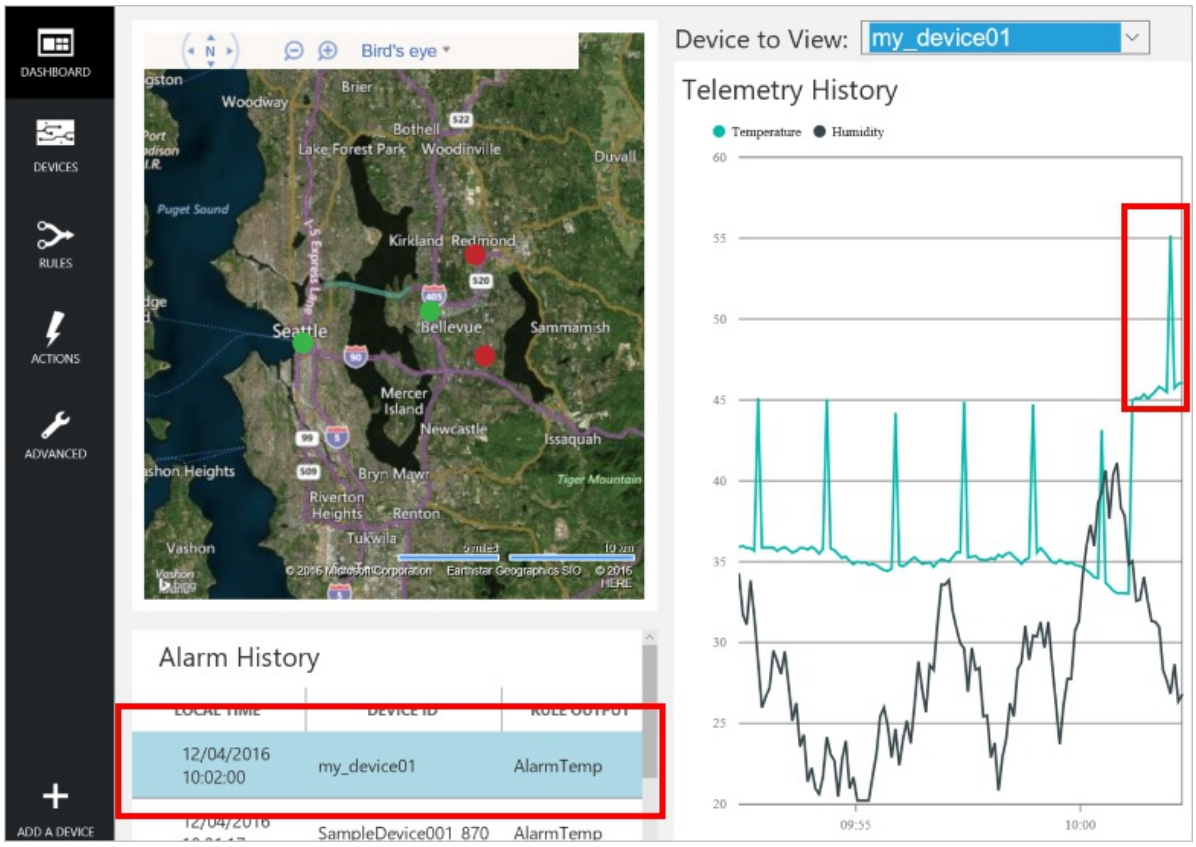
5. Choose **Commands** in the device details pane for the new device.



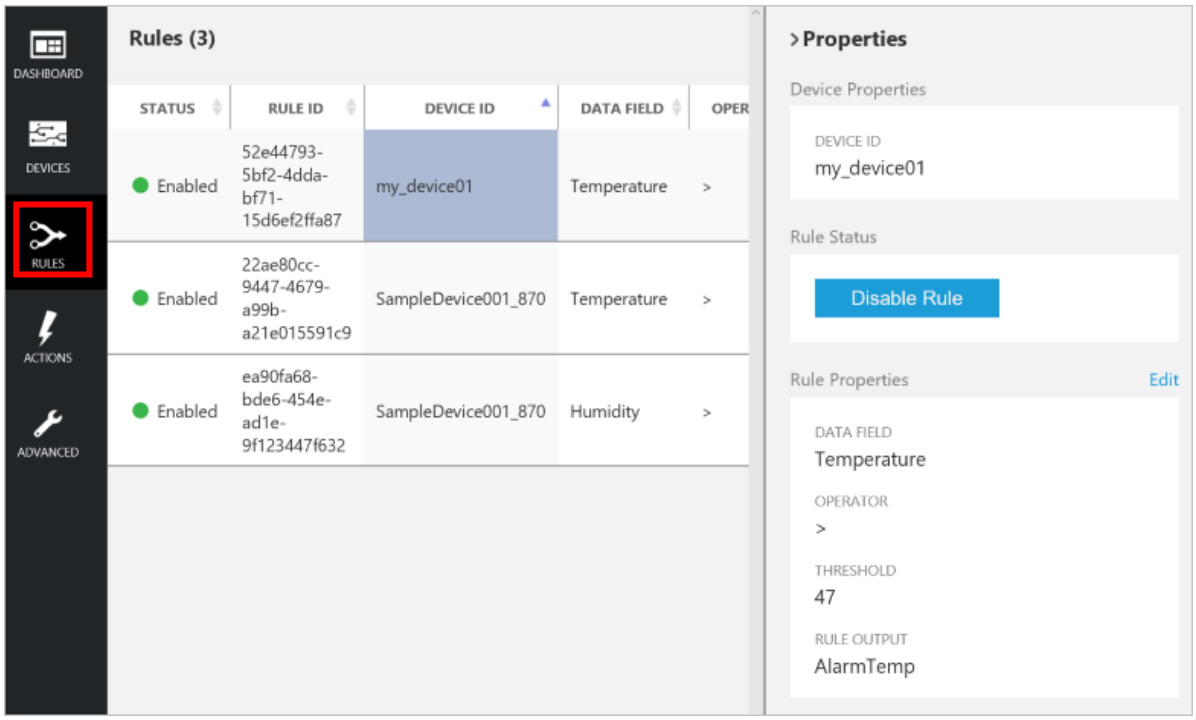

6. Select **ChangeSetPointTemp** from the command list and set **SetPointTemp** to 45. Then choose **Send Command**:



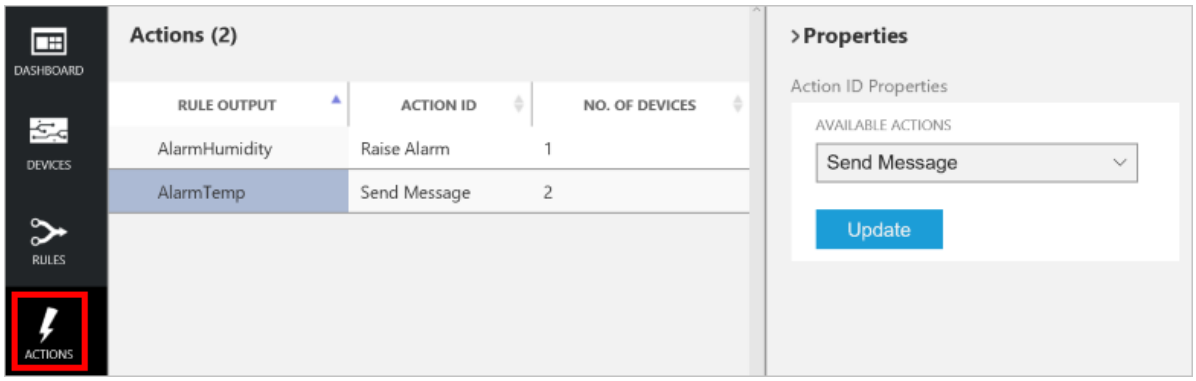

7. Navigate back to the dashboard. After a short time, you will see a new entry in the **Alarm History** pane when the temperature reported by your new device exceeds the 47-degree threshold:




8. You can review and edit all your rules on the **Rules** page of the dashboard:




9. You can review and edit all the actions that can be taken in response to a rule on the **Actions** page of the dashboard:

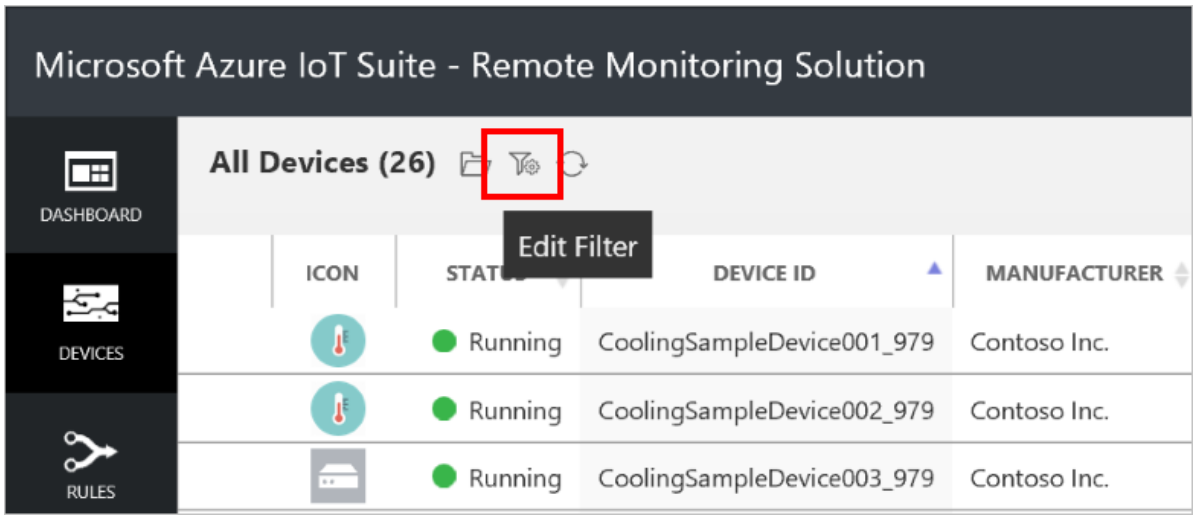



> **NOTE**
> It is possible to define actions that can send an email message or SMS in response to a rule or integrate with a line-of-business system through a Logic App. For more information, see the Connect Logic App to your Azure IoT Suite Remote Monitoring preconfigured solution.

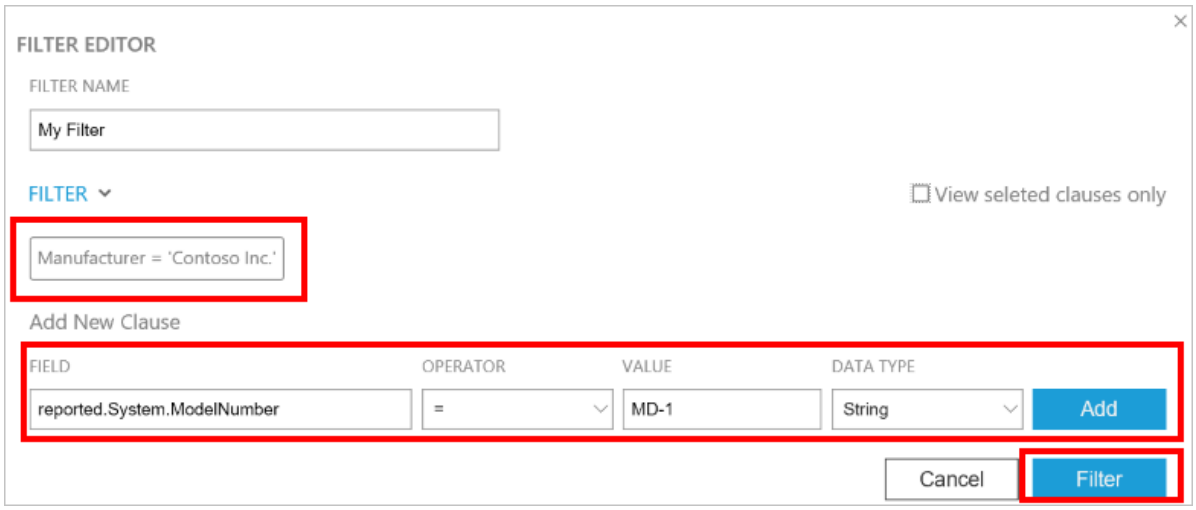
### Manage filters

In the device list, you can create, save, and reload filters to display a customized list of devices connected to your hub. To create a filter:

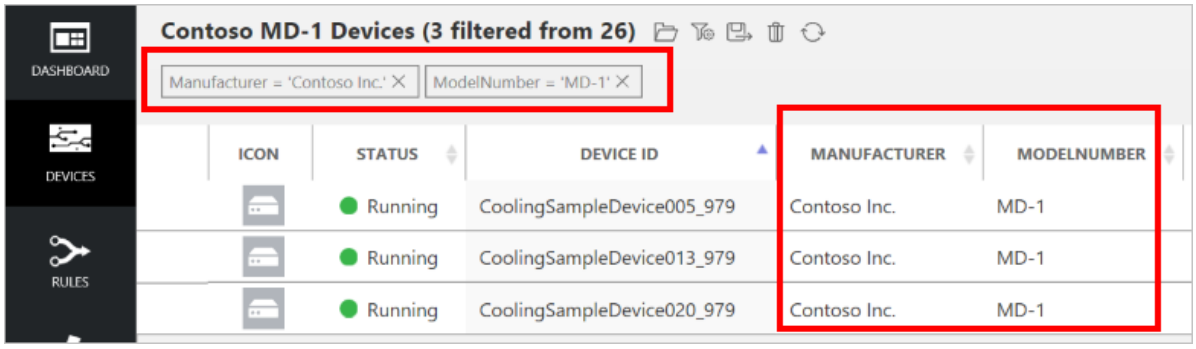
1. Choose the edit filter icon above the list of devices:



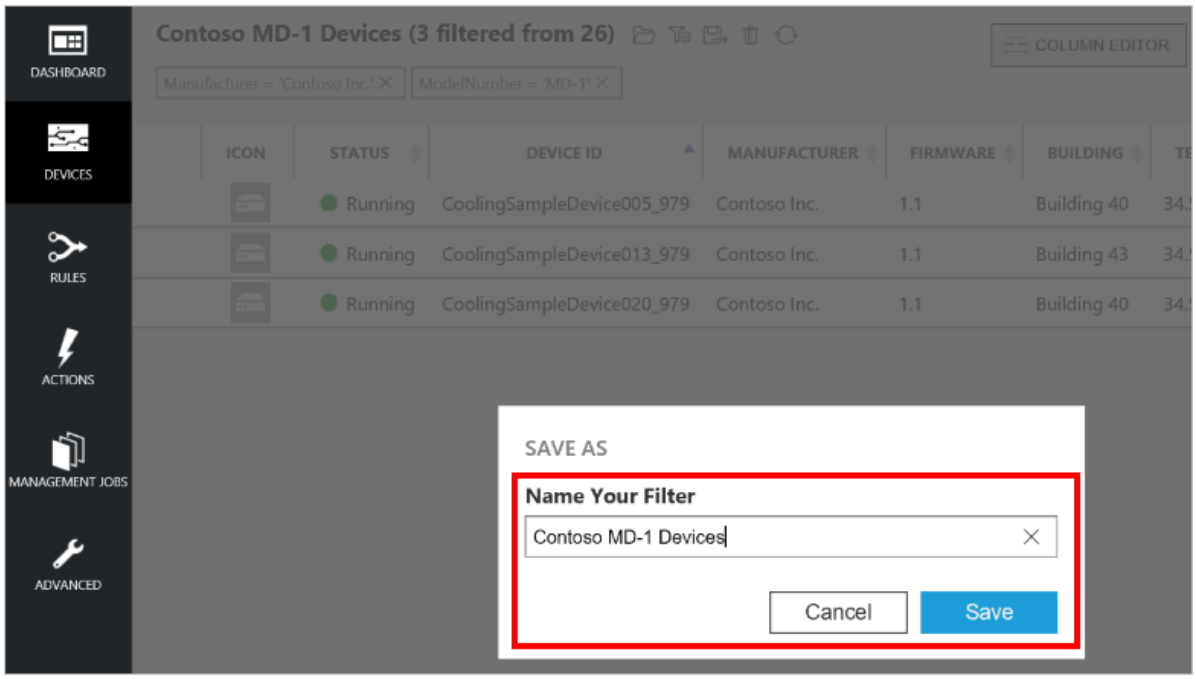

2. In the **Filter editor**, add the fields, operators, and values to filter the device list. You can add multiple clauses to refine your filter. Choose **Filter** to apply the filter:



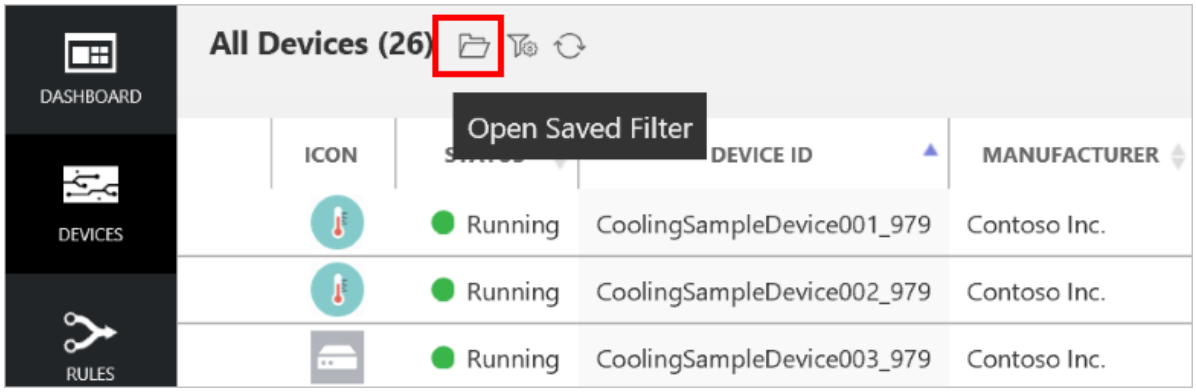

3. In this example, the list is filtered by manufacturer and model number:




4. To save your filter with a custom name, choose the **Save as** icon:




5. To reapply a filter you saved previously, choose the **Open saved filter** icon:



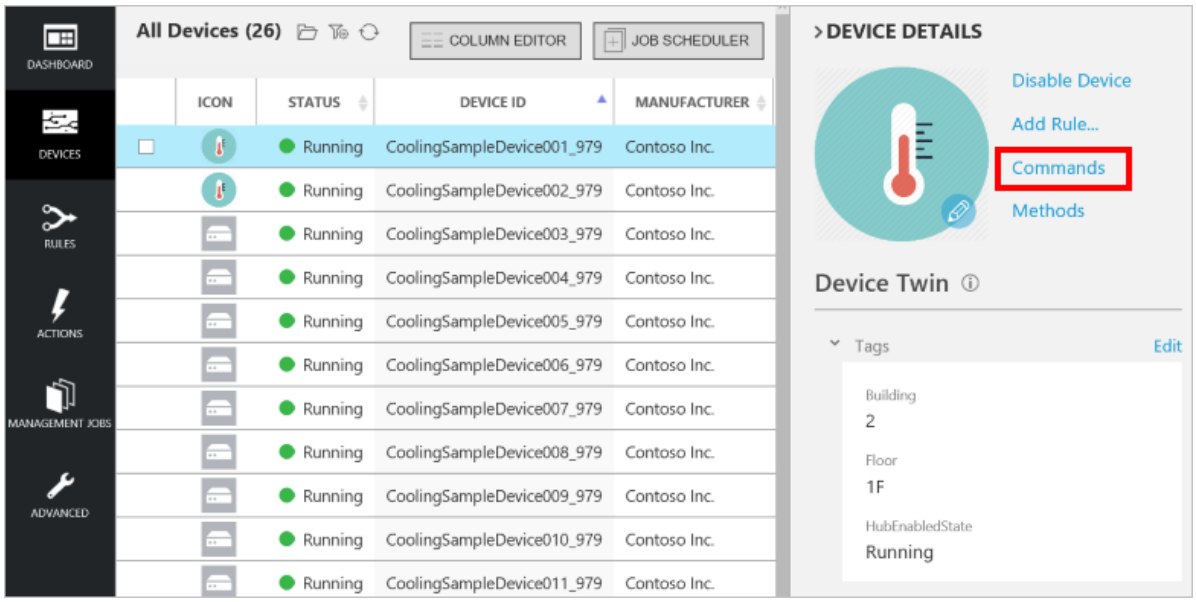

You can create filters based on device id, device state, desired properties, reported properties, and tags. You add your own custom tags to a device in the **Tags** section of the **Device Details** panel, or run a job to update tags on multiple devices.

> **NOTE**
> In the **Filter editor**, you can use the **Advanced view** to edit the query text directly.

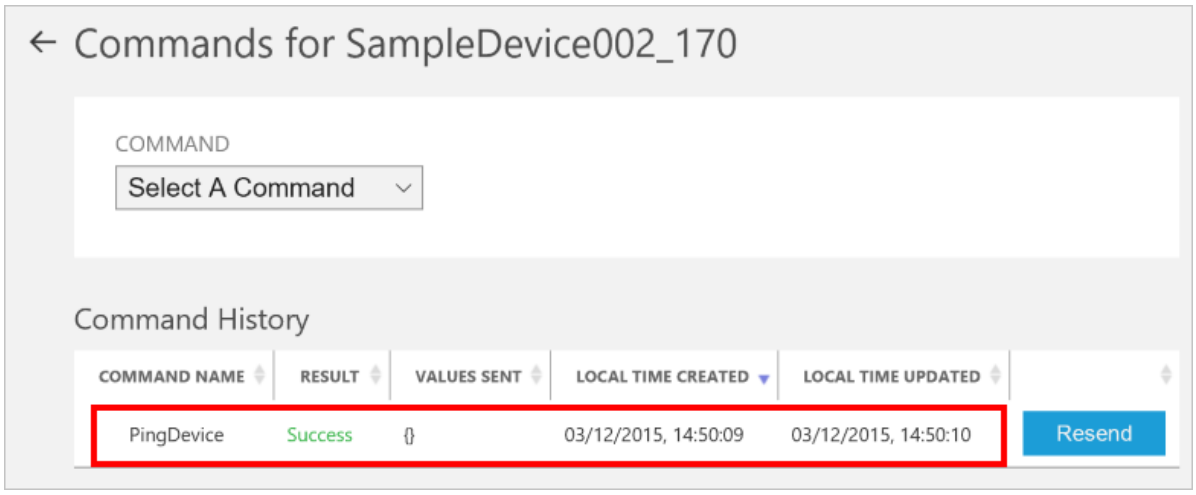
### Commands

From the **Device Details** panel, you can send commands to the device. When a device first starts, it sends information about the commands it supports to the solution. For a discussion of the differences between commands and methods, see Azure IoT Hub cloud-to-device options.

1. Choose **Commands** in the **Device Details** panel for the selected device:



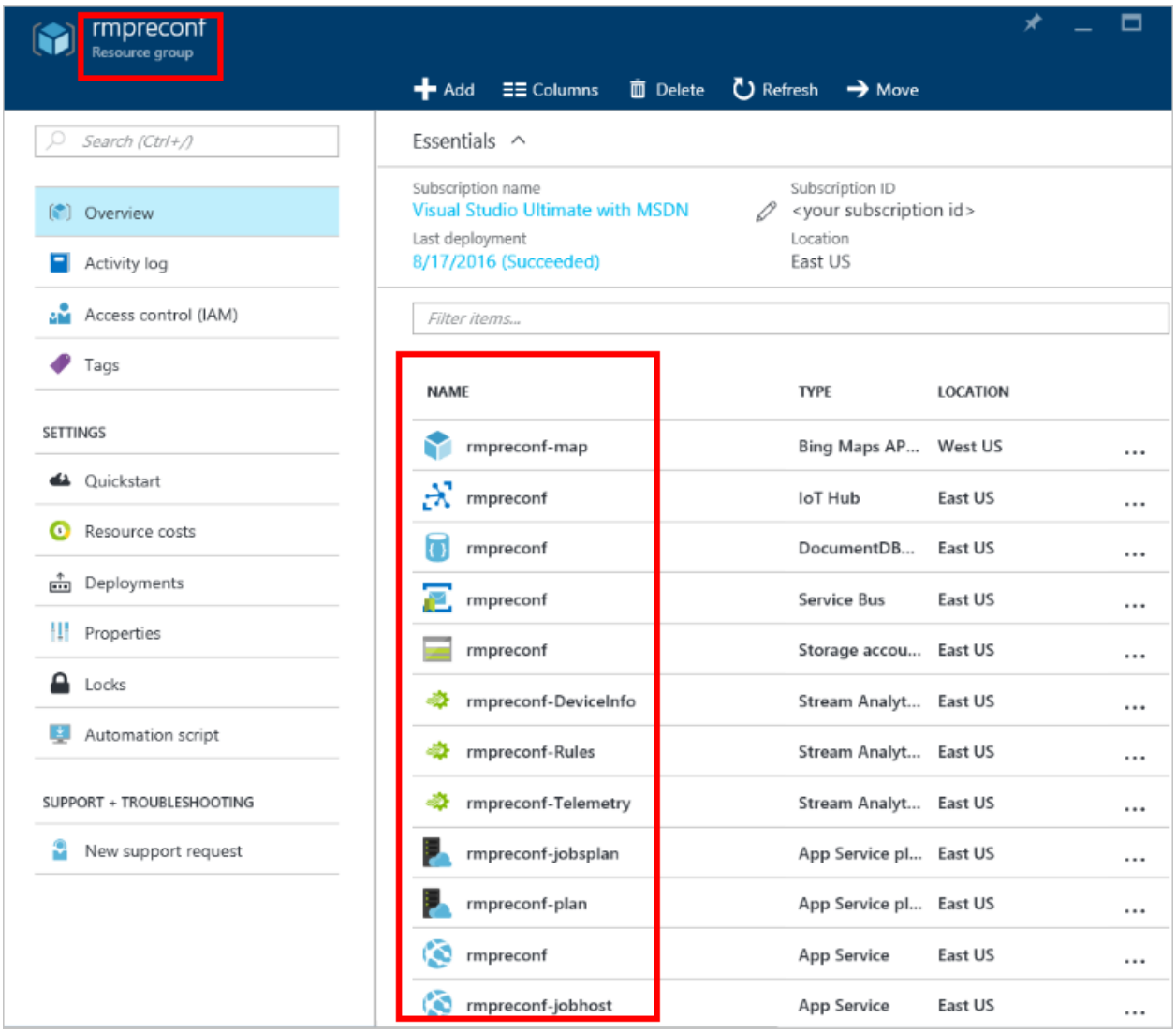

2. Select **PingDevice** from the command list.
3. Choose **Send Command**.
4. You can see the status of the command in the command history.




The solution tracks the status of each command it sends. Initially the result is **Pending**. When the device reports that it has executed the command, the result is set to **Success**.

## Behind the scenes

When you deploy a preconfigured solution, the deployment process creates multiple resources in the Azure subscription you selected. You can view these resources in the Azure portal. The deployment process creates a **resource group** with a name based on the name you choose for your preconfigured solution:




You can view the settings of each resource by selecting it in the list of resources in the resource group.

You can also view the source code for the preconfigured solution. The remote monitoring preconfigured solution source code is in the azure-iot-remote-monitoring GitHub repository:

- The **DeviceAdministration** folder contains the source code for the dashboard.
- The **Simulator** folder contains the source code for the simulated device.
- The **EventProcessor** folder contains the source code for the back-end process that handles the incoming telemetry.

When you are done, you can delete the preconfigured solution from your Azure subscription on the azureiotsuite.com site. This site enables you to easily delete all the resources that were provisioned when you created the preconfigured solution.

> **NOTE**
> To ensure that you delete everything related to the preconfigured solution, delete it on the azureiotsuite.com site and do not delete the resource group in the portal.

## Next Steps

Now that you've deployed a working preconfigured solution, you can continue getting started with IoT Suite by reading the following articles:

- Remote monitoring preconfigured solution walkthrough

- Connect your device to the remote monitoring preconfigured solution
- Permissions on the azureiotsuite.com site

# Permissions on the azureiotsuite.com site

6/27/2017 • 6 min to read • Edit Online

## What happens when you sign in

The first time you sign in at azureiotsuite.com, the site determines the permission levels you have based on the currently selected Azure Active Directory (AAD) tenant and Azure subscription.

1. First, to populate the list of tenants seen next to your username, the site finds out from Azure which AAD tenants you belong to. Currently, the site can only obtain user tokens for one tenant at a time. Therefore, when you switch tenants using the dropdown in the top right corner, the site logs you in to that tenant to obtain the tokens for that tenant.
2. Next, the site finds out from Azure which subscriptions you have associated with the selected tenant. You see the available subscriptions when you create a new preconfigured solution.
3. Finally, the site retrieves all the resources in the subscriptions and resource groups tagged as preconfigured solutions and populates the tiles on the home page.

The following sections describe the roles that control access to the preconfigured solutions.

## AAD roles

The AAD roles control the ability provision preconfigured solutions and manage users in a preconfigured solution.

You can find more information about administrator roles in AAD in Assigning administrator roles in Azure AD. The current article focuses on the **Global Administrator** and the **User** directory roles as used by the preconfigured solutions.

#### Global administrator

There can be many global administrators per AAD tenant:

- When you create an AAD tenant, you are by default the global administrator of that tenant.
- The global administrator can provision a preconfigured solution and is assigned an **Admin** role for the application inside their AAD tenant.
- If another user in the same AAD tenant creates an application, the default role granted to the global administrator is **ReadOnly**.
- A global administrator can assign users to roles for applications using the Azure portal.

#### Domain user

There can be many domain users per AAD tenant:

- A domain user can provision a preconfigured solution through the azureiotsuite.com site. By default, the domain user is granted the **Admin** role in the provisioned application.
- A domain user can create an application using the build.cmd script in the azure-iot-remote-monitoring, azure-iot-predictive-maintenance, or azure-iot-connected-factory repository. However, the default role granted to the domain user is **ReadOnly**, because a domain user does not have permission to assign roles.
- If another user in the AAD tenant creates an application, the domain user is assigned the **ReadOnly** role by default for that application.
- A domain user cannot assign roles for applications; therefore a domain user cannot add users or roles for users for an application even if they provisioned it.

**Guest User**

There can be many guest users per AAD tenant. Guest users have a limited set of rights in the AAD tenant. As a result, guest users cannot provision a preconfigured solution in the AAD tenant.

For more information about users and roles in AAD, see the following resources:

- Create users in Azure AD
- Assign users to apps

# Azure subscription administrator roles

The Azure admin roles control the ability to map an Azure subscription to an AD tenant.

Find out more about the Azure admin roles in the article How to add or change Azure Co-Administrator, Service Administrator, and Account Administrator.

# Application roles

The application roles control access to devices in your preconfigured solution.

There are two defined roles and one implicit role defined in a provisioned application:

- **Admin:** Has full control to add, manage, remove devices, and modify settings.
- **ReadOnly:** Can view devices, rules, actions, jobs, and telemetry.

You can find the permissions assigned to each role in the RolePermissions.cs source file.

**Changing application roles for a user**

You can use the following procedure to make a user in your Active Directory an administrator of your preconfigured solution.

You must be an AAD global administrator to change roles for a user:

1. Go to the Azure portal.
2. Select **Azure Active Directory**.
3. Make sure you are using the directory you chose on azureiotsuite.com when you provisioned your solution. If you have multiple directories associated with your subscription, you can switch between them if you click your account name at the top-right of the portal.
4. Click **Enterprise applications**, then **All applications**.
5. Show **All applications** with **Any** status. Then search for an application with name of your preconfigured solution.
6. Click the name of the application that matches your preconfigured solution name.
7. Click **Users and groups**.
8. Select the user you want to switch roles.
9. Click **Assign** and select the role (such as **Admin**) you'd like to assign to the user, click the check mark.

# FAQ

**I'm a service administrator and I'd like to change the directory mapping between my subscription and a specific AAD tenant. How do I complete this task?**

1. Go to the Azure classic portal, click **Settings** in the list of services on the left-hand side.
2. Select the subscription you'd like to change the directory mapping to.
3. Click **Edit Directory**.
4. Select the **Directory** you would like to use in the dropdown. Click the forward arrow.

5. Confirm the directory mapping and affected co-administrators. If you are moving from another directory, all the co-administrators from the original directory are removed.

**I'm a domain user/member on the AAD tenant and I've created a preconfigured solution. How do I get assigned a role for my application?**

Ask a global administrator to make you a global administrator on the AAD tenant and then assign roles to users yourself. Alternatively, ask a global administrator to assign you a role directly. If you'd like to change the AAD tenant your preconfigured solution has been deployed to, see the next question.

**How do I switch the AAD tenant my remote monitoring preconfigured solution and application are assigned to?**

You can run a cloud deployment from https://github.com/Azure/azure-iot-remote-monitoring and redeploy with a newly created AAD tenant. Because you are, by default, a global administrator when you create an AAD tenant, you have permissions to add users and assign roles to those users.

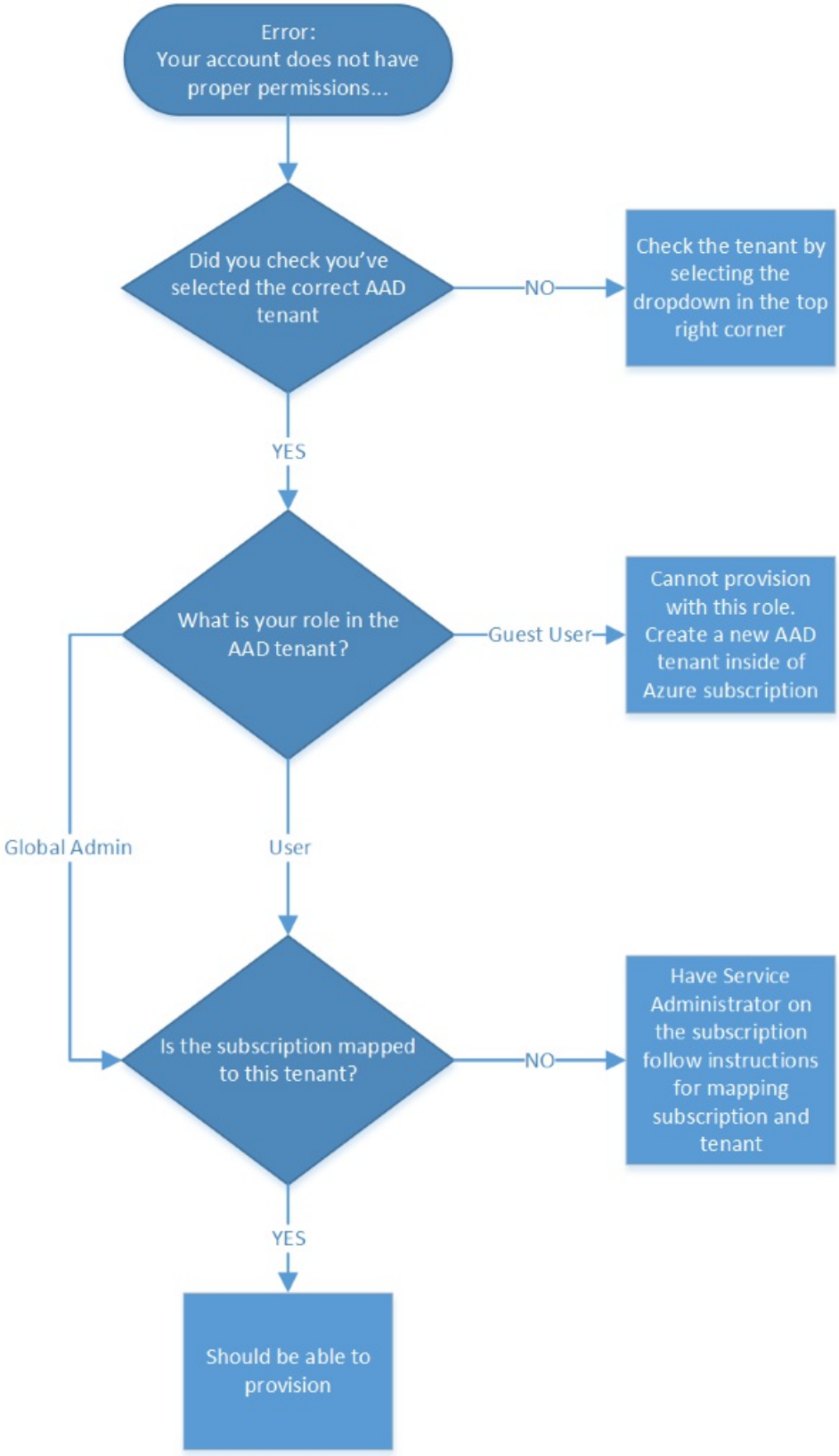
1. Create an AAD directory in the Azure portal.
2. Go to https://github.com/Azure/azure-iot-remote-monitoring.
3. Run `build.cmd cloud [debug | release] {name of previously deployed remote monitoring solution}` (For example, `build.cmd cloud debug myRMSolution`)
4. When prompted, set the **tenantid** to be your newly created tenant instead of your previous tenant.

**I want to change a Service Administrator or Co-Administrator when logged in with an organisational account**

See the support article Changing Service Administrator and Co-Administrator when logged in with an organisational account.

**Why am I seeing this error? "Your account does not have the proper permissions to create a solution. Please check with your account administrator or try with a different account."**

Look at the following diagram for guidance:




**NOTE**

If you continue to see the error after validating you are a global administrator on the AAD tenant and a co-administrator on the subscription, have your account administrator remove the user and reassign necessary permissions in this order. First, add the user as a global administrator and then add user as a co-administrator on the Azure subscription. If issues persist, contact Help & Support.

**Why am I seeing this error when I have an Azure subscription? "An Azure subscription is required to create pre-configured solutions. You can create a free trial account in just a couple of minutes."**

If you're certain you have an Azure subscription, validate the tenant mapping for your subscription and ensure

the correct tenant is selected in the dropdown. If you've validated the desired tenant is correct, follow the preceding diagram and validate the mapping of your subscription and this AAD tenant.

## Next steps

To continue learning about IoT Suite, see how you can customize a preconfigured solution.

# Predictive maintenance preconfigured solution overview

7/24/2017 • 6 min to read • Edit Online

The *predictive maintenance* preconfigured solution is one of the Microsoft Azure IoT Suite preconfigured solutions. This solution integrates real-time device telemetry collection with a predictive model created using Azure Machine Learning.

With Azure IoT Suite, you can quickly and easily connect to and monitor assets, and analyze telemetry in real time in dashboards and visualizations. In the predictive maintenance solution, the dashboards and visualizations provide you with new intelligence that can drive efficiencies and enhance revenue streams.

## The Scenario

Fabrikam is a regional airline that focuses on great customer experience at competitive prices. One cause of flight delays is maintenance issues and aircraft engine maintenance is particularly challenging. Fabrikam must avoid engine failure during flight at all costs, so it inspects its engines regularly and schedules maintenance according to a plan. However, aircraft engines don't always wear the same. Some unnecessary maintenance is performed on engines. More importantly, issues arise which can ground an aircraft until maintenance is performed. If an aircraft is at a location where the right technicians or spare parts are not available, these issues can be especially costly.

The engines of Fabrikam's aircraft are instrumented with sensors that monitor engine conditions during flight. Fabrikam uses the predictive maintenance solution to collect the sensor data collected during the flight. After accumulating years of engine operational and failure data, Fabrikam's data scientists have modeled a way to predict the Remaining Useful Life (RUL) of an aircraft engine. The model uses a correlation between data from four of the engine sensors and engine wear that leads to eventual failure. While Fabrikam continues to perform regular inspections to ensure safety, it can now use the models to compute the RUL for each engine after every flight. The model uses the telemetry collected from the engines during the flight. Fabrikam can now predict future points of failure and plan for maintenance and repair in advance.

> **NOTE**
> The solution model uses actual engine wear data.

By predicting the point when maintenance is required, Fabrikam can optimize its operations to reduce costs.

Maintenance coordinators work with schedulers to:

- Plan maintenance to coincide with an aircraft stopping at a particular location.
- Ensure sufficient time is available for the aircraft to be out of service without causing schedule disruption.
- To schedule technicians to ensure that aircraft are serviced efficiently without wait time.

Inventory control managers receive maintenance plans, so they can optimize their ordering process and spare parts inventory.

These activities enable Fabrikam to minimize aircraft ground time and reduce operating costs while ensuring the safety of passengers and crew.

To understand how Azure IoT Suite provides the capabilities customers need to realize the potential of predictive maintenance, review this infographic.

# How the predictive maintenance solution is built

The solution uses an existing Azure Machine Learning model available as a template to show these capabilities working from device telemetry collected through IoT Suite services. Microsoft has built a regression model of an aircraft engine based on publically available data[1], and step-by-step guidance on how to use the model.

The Azure IoT predictive maintenance solution uses the regression model created from this template. The model is deployed into your Azure subscription and exposed through an automatically generated API. The solution includes a subset of the testing data representing 4 (of 100 total) engines and the 4 (of 21 total) sensor data streams. This data is sufficient to provide an accurate result from the trained model.

*[1] A. Saxena and K. Goebel (2008). "Turbofan Engine Degradation Simulation Data Set", NASA Ames Prognostics Data Repository (http://ti.arc.nasa.gov/tech/dash/pcoe/prognostic-data-repository/), NASA Ames Research Center, Moffett Field, CA*

# Get started with predictive maintenance

This tutorial shows you how to provision the predictive maintenance solution. It also walks you through the basic features of the predictive maintenance solution. You can access many of these features through the solution dashboard that deploys along with the preconfigured solution.

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** the **Predictive maintenance** tile.
3. Enter a **Solution name** for your predictive maintenance preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane. From this pane, you can launch the solution dashboard and access the Machine Learning workspace.

> **NOTE**
> If you encounter issues deploying the preconfigured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

# View the solution

This section guides you through the solution UI.

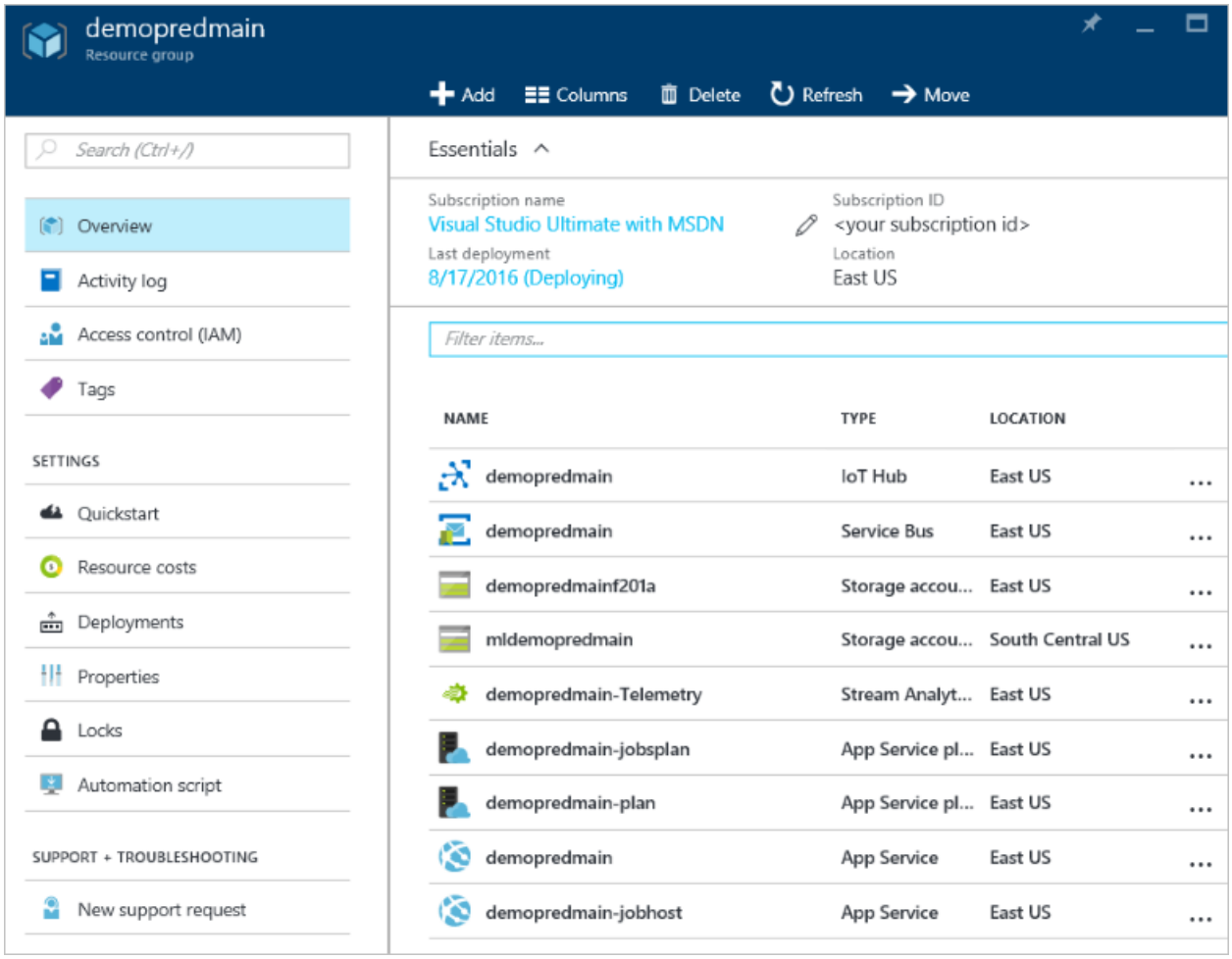
**Predictive Maintenance Dashboard**

This page in the web application uses PowerBI JavaScript controls (see the PowerBI-visuals repository) to visualize:

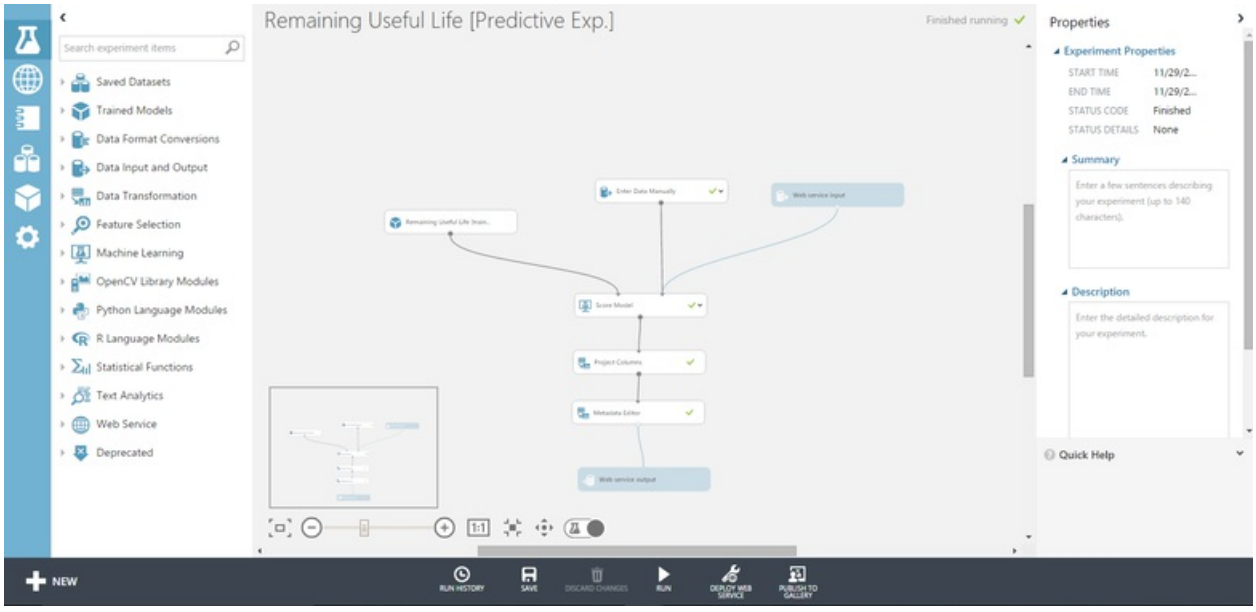
- The output data from the Stream Analytics jobs in blob storage.
- The RUL and cycle count per aircraft engine.

**Observing the behavior of the cloud solution**

In the Azure portal, navigate to the resource group with the solution name you chose to view your provisioned resources.

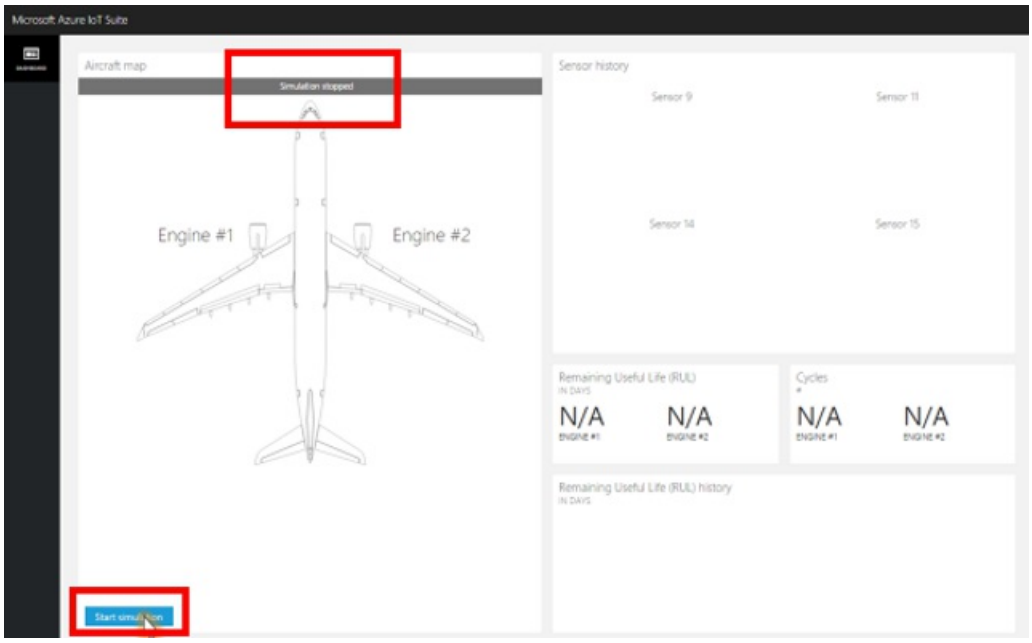



When you provision the preconfigured solution, you receive an email with a link to the Machine Learning workspace. You can also navigate to the Machine Learning workspace from the azureiotsuite.com page for your provisioned solution. A tile is available on this page when the solution is in the **Ready** state.

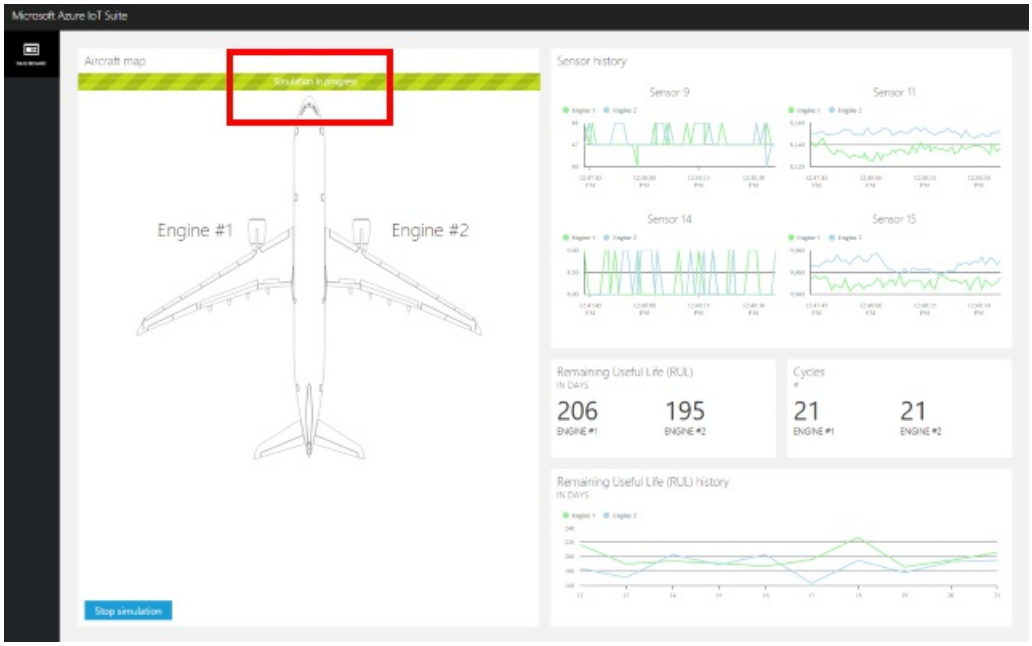



In the solution portal, you can see that the sample is provisioned with four simulated devices to represent two

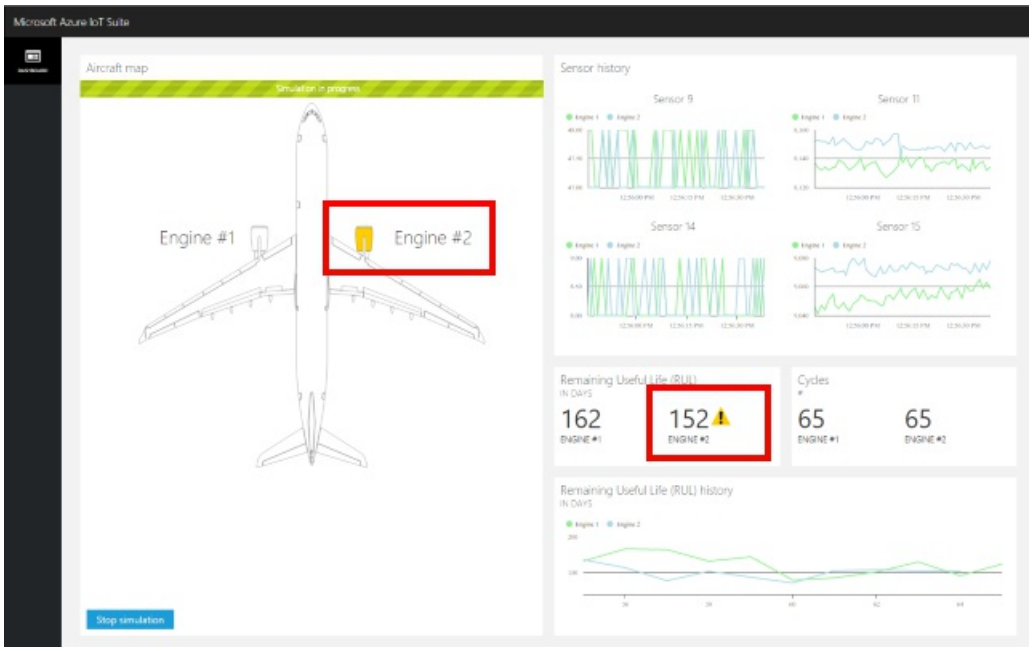
aircraft with two engines per aircraft, each with four sensors. When you first navigate to the solution portal, the simulation is stopped.




Click **Start simulation** to begin the simulation. The sensor history, RUL, Cycles, and RUL history populate the dashboard.




When RUL is less than 160 (an arbitrary threshold chosen for demonstration purposes), the solution portal displays a warning symbol next to the RUL display. The solution portal also highlights the aircraft engine in yellow. Notice how the RUL values have a general downward trend overall, but tend to bounce up and down. This behavior results from the varying cycle lengths and the model accuracy.




The full simulation takes around 35 minutes to complete 148 cycles. The 160 RUL threshold is met for the first time at around 5 minutes and both engines hit the threshold at around 8 minutes.

The simulation runs through the complete dataset for 148 cycles and settles on final RUL and cycle values.

You can stop the simulation at any point, but clicking **Start Simulation** replays the simulation from the start of the dataset.

## Next steps

To learn more about how Azure IoT enables predictive maintenance scenarios, read Capture value from the Internet of Things.

Take a walkthrough of the predictive maintenance solution.

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

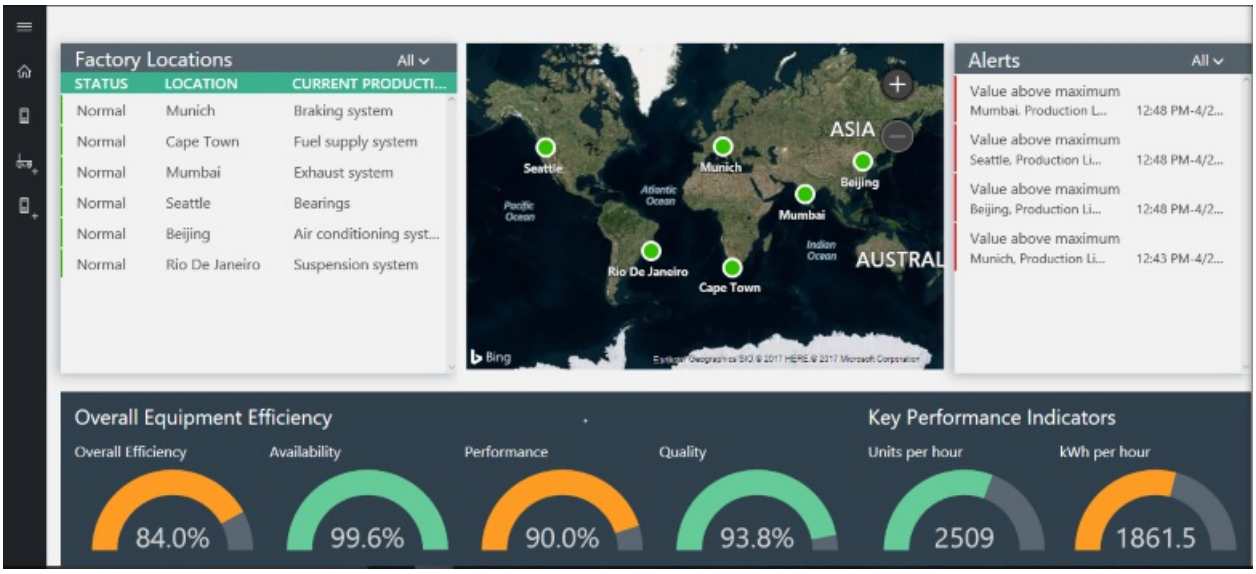
- Frequently asked questions for IoT Suite
- IoT security from the ground up

# Get started with the connected factory preconfigured solution

8/24/2017 • 12 min to read • Edit Online

Azure IoT Suite preconfigured solutions combine multiple Azure IoT services to deliver end-to-end solutions that implement common IoT business scenarios. The *connected factory* preconfigured solution connects to and monitors your industrial devices. You can use the solution to analyze the stream of data from your devices and to drive operational productivity and profitability.

This tutorial shows you how to provision the connected factory preconfigured solution. It also walks you through the basic features of the preconfigured solution. You can access many of these features from the solution *dashboard* that deploys as part of the preconfigured solution:




To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

## Provision the solution

1. Log on to azureiotsuite.com using your Azure account credentials, and click "**+**" to create a solution.
2. Click **Select** on the **Connected factory** tile.
3. Enter a **Solution name** for your connected factory preconfigured solution.
4. Select the **Subscription** and **Region** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**While you wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
>
> If you encounter issues deploying the preconfigured solution, review Permissions on the azureiotsuite.com site and the Connected factory FAQ. If the issues persist, create a service ticket on the portal.

Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

## Scenario overview

When you deploy the connected factory preconfigured solution, it is prepopulated with resources that enable you to step through a common industrial scenario. In this scenario, several factories connected to the solution report the data values required to compute overall equipment efficiency (OEE) and key performance indicators (KPIs). The following sections show you how to:

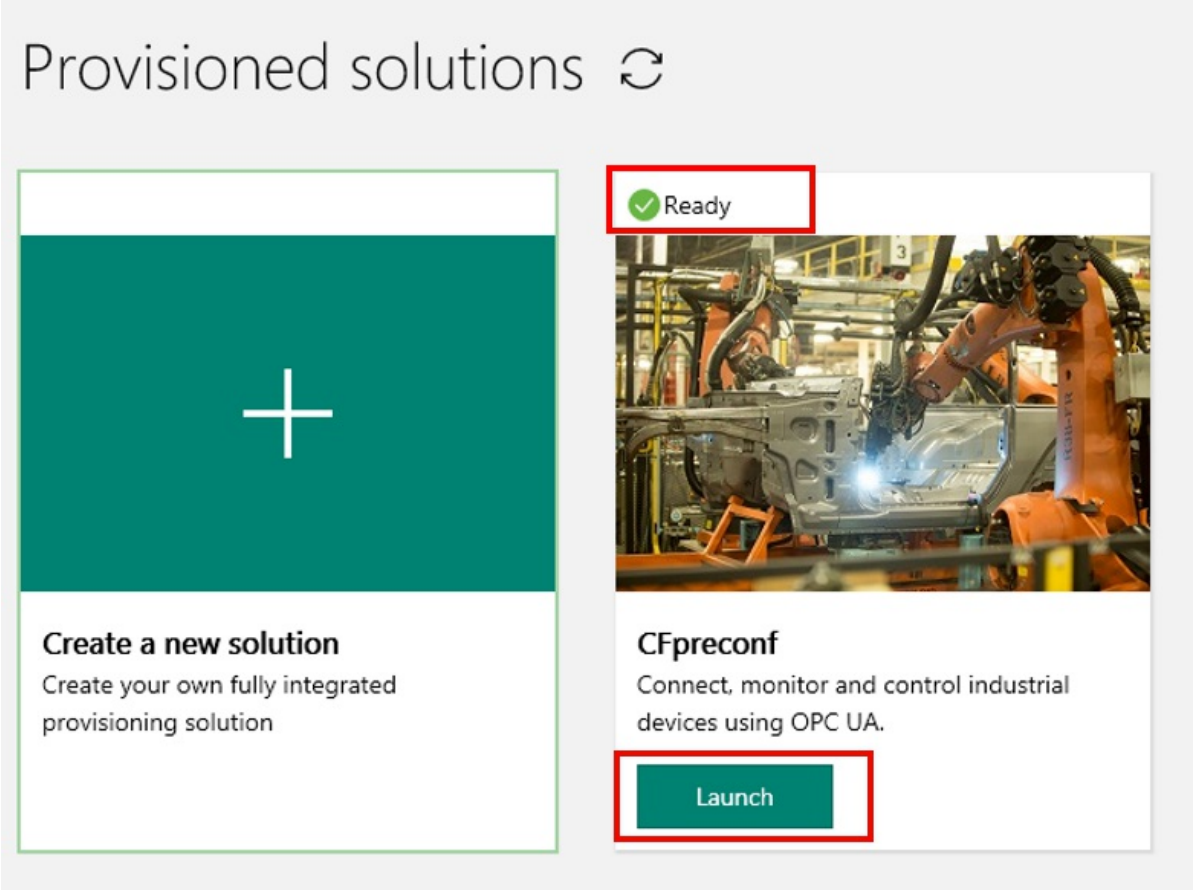
- Monitor factory, production lines, station OEE, and KPI values
- Analyze the telemetry data generated from these devices using Azure Time Series Insights
- Act on alerts to fix issues

A key feature of this scenario is that you can perform all these actions remotely from the solution dashboard. You do not need physical access to the devices.

## View the solution dashboard

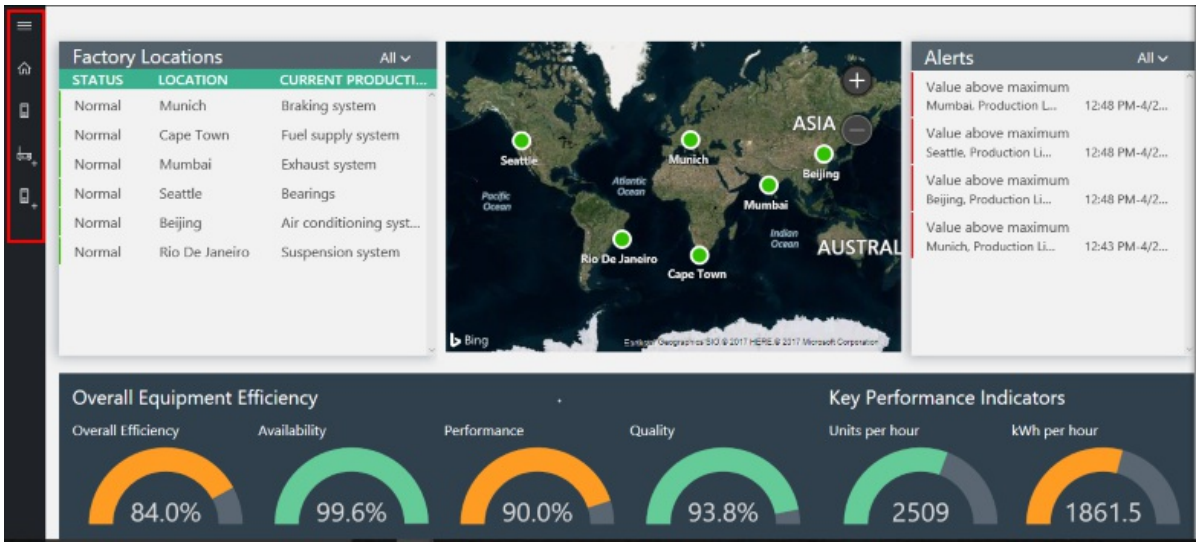
The solution dashboard enables you to manage the deployed solution. It is a hierarchical representation of a global factory configuration. For example, you can view OEE and KPIs, publish new nodes for telemetry and action alerts.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your connected factory solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. To navigate to other areas of the portal, use the menu

on the left-hand side of the page.

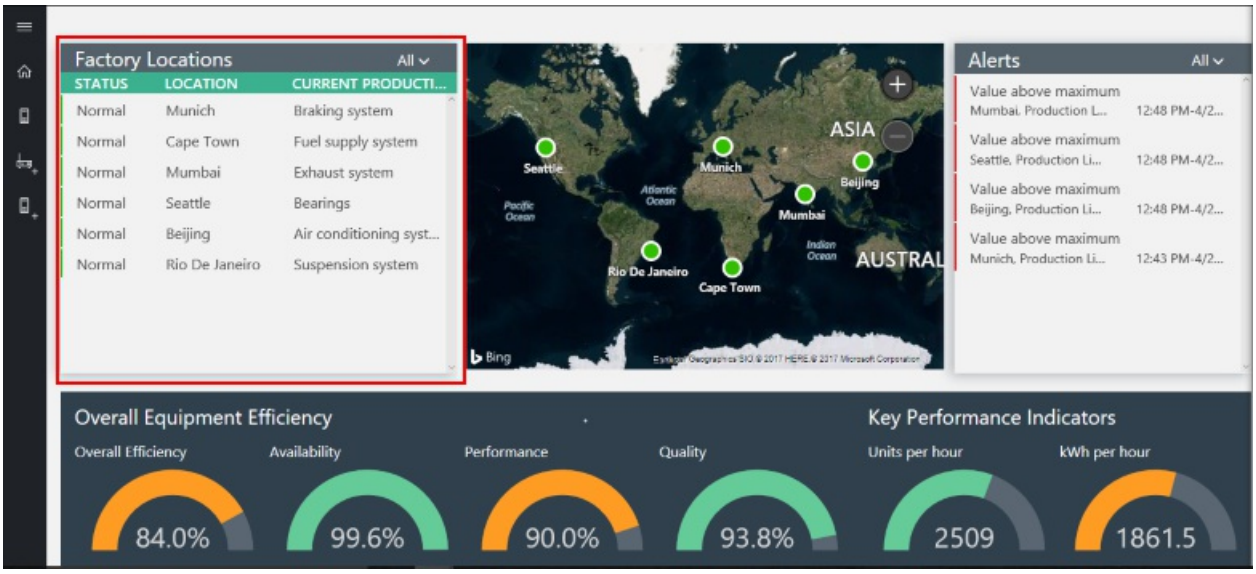



The dashboard displays the following information:

- A **Factory list** panel that shows the status, location, and current production configuration in the solution. When you first run the solution, there are a number of simulated devices. The production line simulation is composed of three real OPC UA servers per production line that perform simulated tasks and share data. For more information about OPC UA, see the Connected factory FAQ.
- A **map** that displays the location of each device connected to the solution. The solution can use the Bing Maps API to plot information on the map. If your subscription is enabled for Bing Maps Enterprise API, then this feature is used automatically. If not, see the FAQ to learn how to make the map dynamic.
- An **Alerts** panel that displays alerts generated when a telemetry or OEE/KPI value exceeds a specific threshold.
- An **Overall Equipment Efficiency** panel that shows the OEE values for the whole enterprise, or the factory/production line/station you are viewing. This value is aggregated from the station view to the enterprise level. The OEE figure and its constituent elements can be further analyzed.
- **Key Performance Indicators** panel that displays the number of units produced and energy used by the whole enterprise or the factory/production line/station you are viewing. These values are aggregated from a station view to the enterprise level.

## View factories

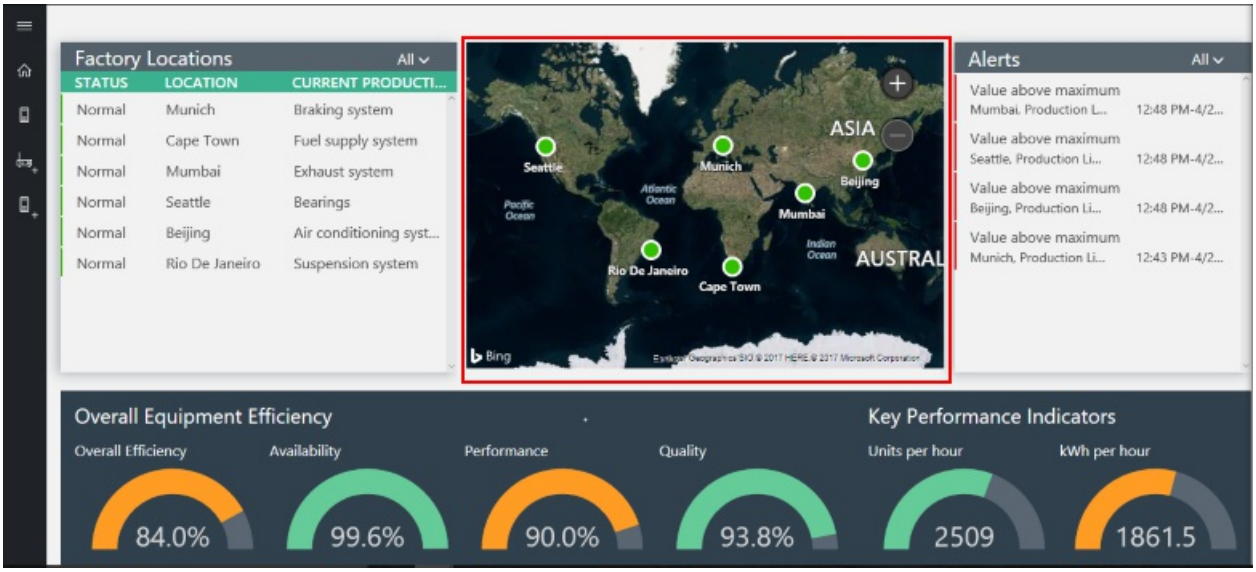
The *Factories* panel shows you the geographical location of all the factories in the solution, their status, and current production configuration. From the locations list, you can navigate to the other levels in the solution hierarchy. The rows in the list are hyperlinks that link details of the production lines at that location. It is then possible to drill into the production line details and down to the station level view. You can also apply a filter to the list.




1. The **Factory panel** shows the factory list for this solution.
2. The factory list initially shows six factories created by the provisioning process. You can add additional simulated and physical devices to the solution.
3. To view the details of a factory, click the row in the factory list.
4. To view the details of a production line, click the row in the list.
5. To view the published OPC UA nodes of a station on the production line, click the row in the list.
6. To view details on a specific node in the station, click the row in the list. This action launches the context panel with Time Series Insights visualizations. Click these graphs to do further analysis in the Time Series Insights explorer environment.

## View map

If your subscription has access to the Bing Maps API, the *Factories* map shows you the geographical location and status of all the factories in the solution. To drill into the location details, click the locations displayed on the map.




## View alerts

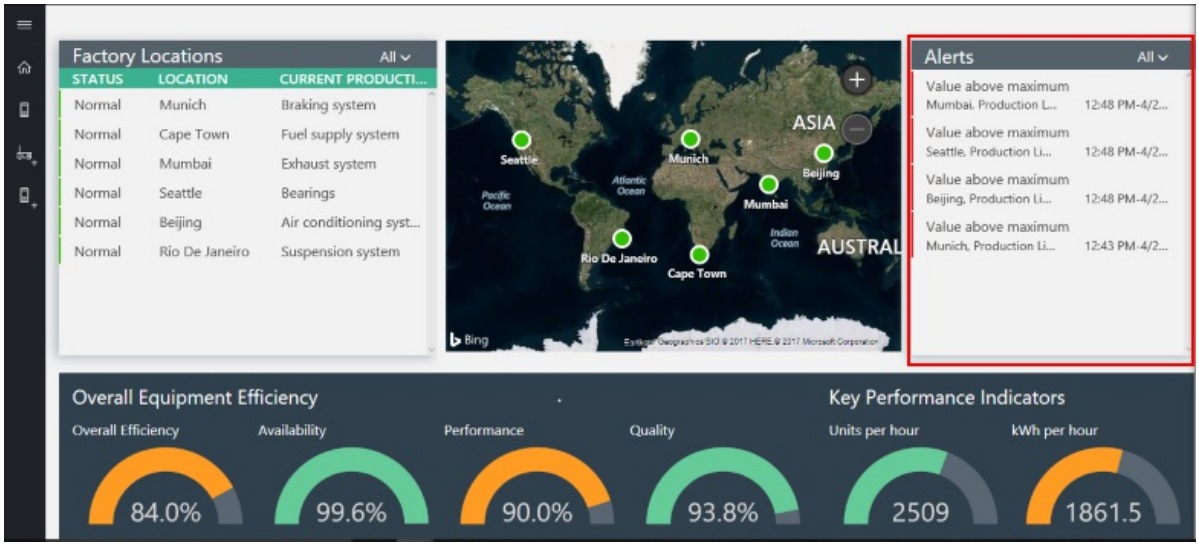
The **Alert** panel shows you alerts generated due to a reported value or a calculated OEE/KPI value exceeding its configured threshold. This panel displays alerts at each level of the hierarchy, from the station level view to the global view. The alerts contain a description of the alert, date, time, location, and number of occurrences. You can

gain insights in to the data that caused the alert using the Time Series Insights data. The Time Series Insights data is visualized in the alerts where applicable. If you are an Administrator, you can take default actions on the alerts such as:

- Close the alert.
- Acknowledge the alert.

Optionally, you can take more complex actions. For example, for the Pressure OPC UA Node of the Assembly you could:

- Show supporting information in a web page in a new browser window.
- Mitigate the cause of the alert by calling an OPC UA method on the device.
- Suppress the availability of the default actions.

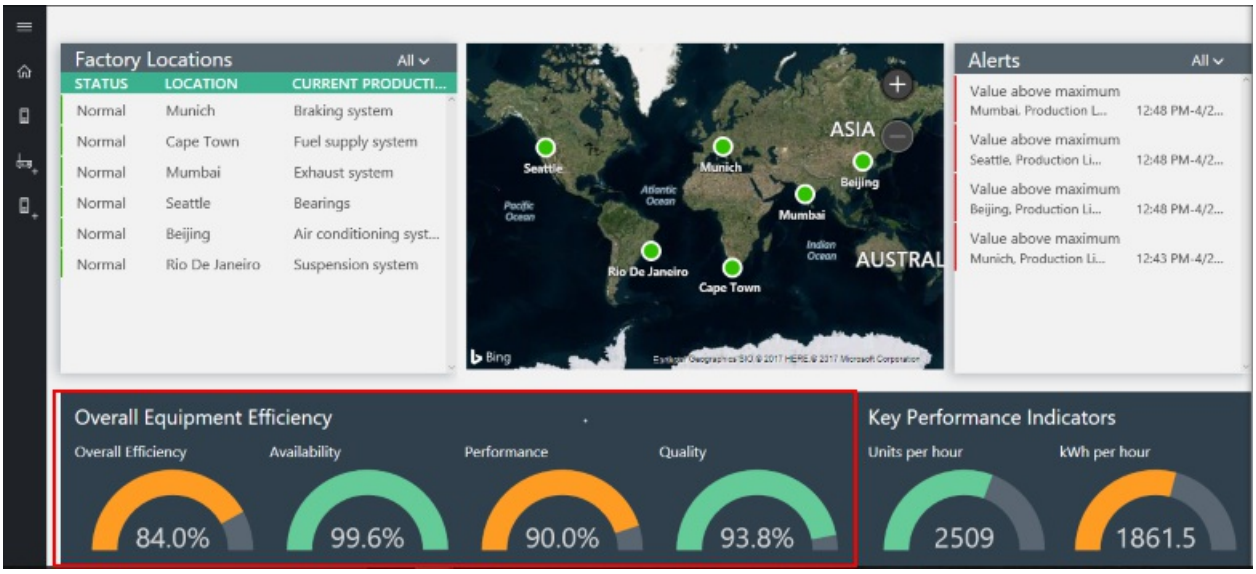



> **NOTE**
>
> These alerts are generated by rules that are specified in a configuration file in the preconfigured solution. These rules can generate alerts when the OEE or KPI figures or OPC UA Node values are exceeding their configured threshold.

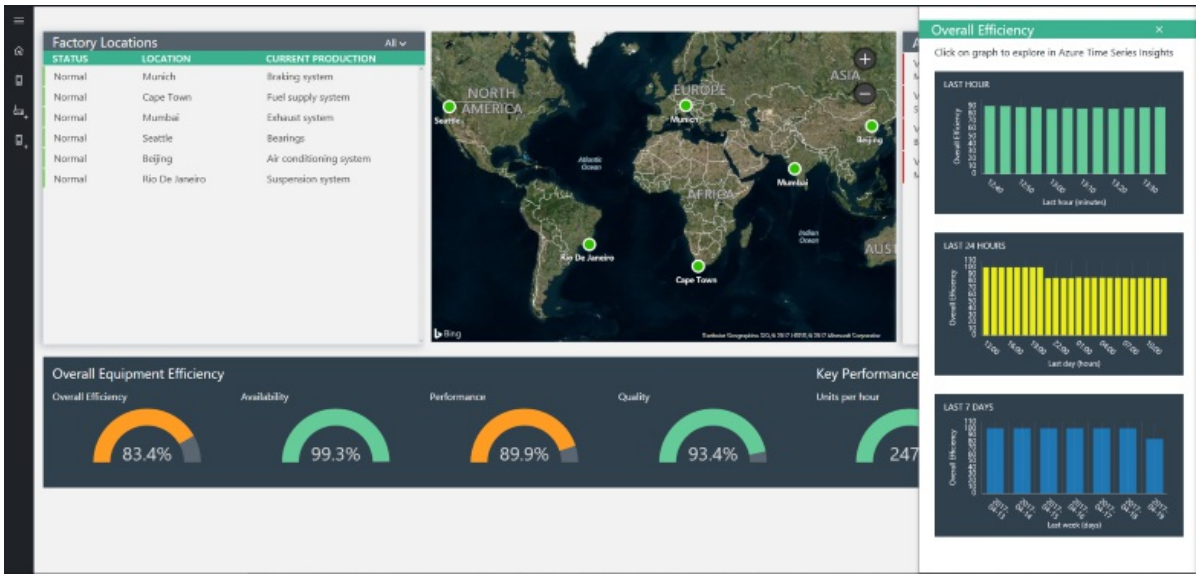
1. The **Alerts panel** shows the alerts generated in this solution.
2. To view the details of an alert, click the alert in the alerts panel.
3. To further analyze the alert data, click the graph in the alert panel to open the Time Series Insights explorer environment.
4. To address the alert, several actions are available in the alert panel. Choose the appropriate option for you and click the execute action command button.

## View overall equipment efficiency

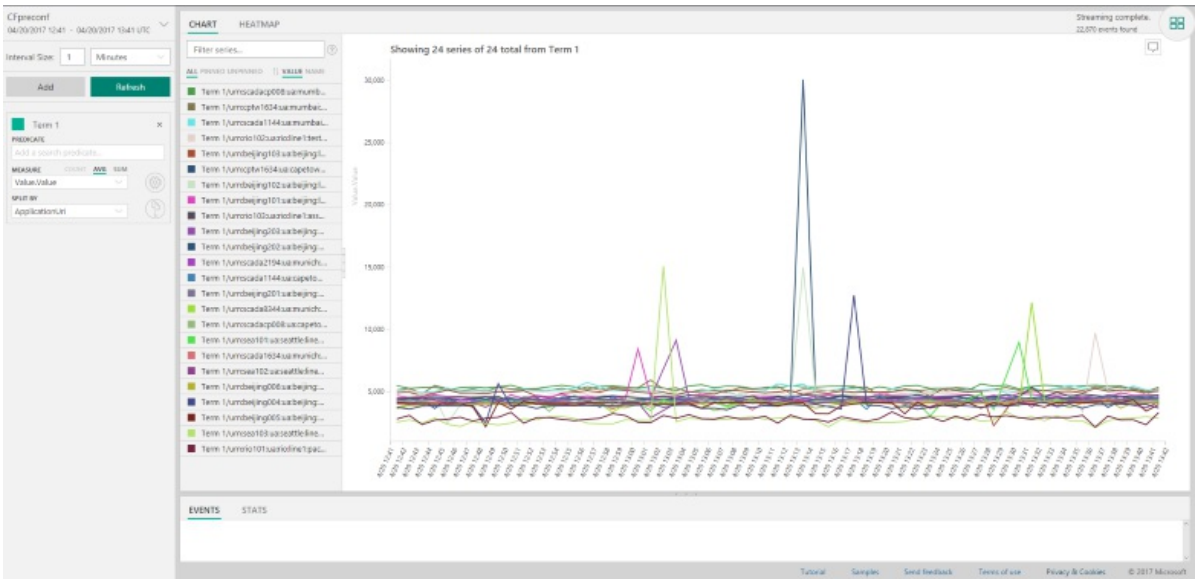
OEE rates the efficiency of the manufacturing process using a key production-related operational parameters. OEE is an industry standard measure calculated by multiplying the availability rate, performance rate, and quality rate: OEE = availability x performance x quality.




1. To view OEE for any level in the hierarchy, navigate to the specific view you require. The OEE for that view displays in the panel along with each of the elements that make up the OEE percentage.
2. To further analyze the OEE for any level in the hierarchy data, click either the OEE, availability, performance, or quality percentage. A context panel appears with Time Series Insights powered visualizations that shows data from the last hour, last 24 hours, and last 7 days.

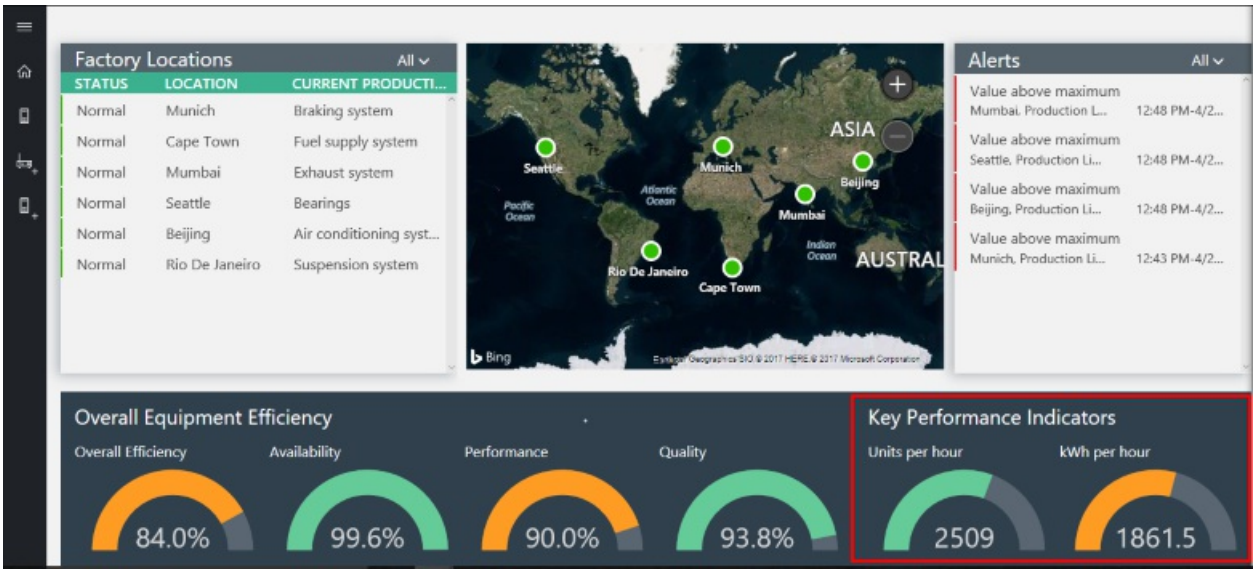



3. To further analyze the alert data, click the graph in the alert panel. This action opens the Time Series Insights explorer environment.




## View Key Performance Indicators

The solution provides two key performance indicators, *units per hour* and *energy used in kWh*.




1. To view units per hour or energy used for any level in the hierarchy, navigate to the specific view you require. The units per hour and energy used display in the panel.

2. To analyze units per hour or energy used for any level in the hierarchy further, click the gauge in the **Key Performance Indicators** panel. A context panel appears with Time Series Insights powered visualizations enabling you to view data from the last hour, the last 24 hours, and last 7 days.

## Scenario review

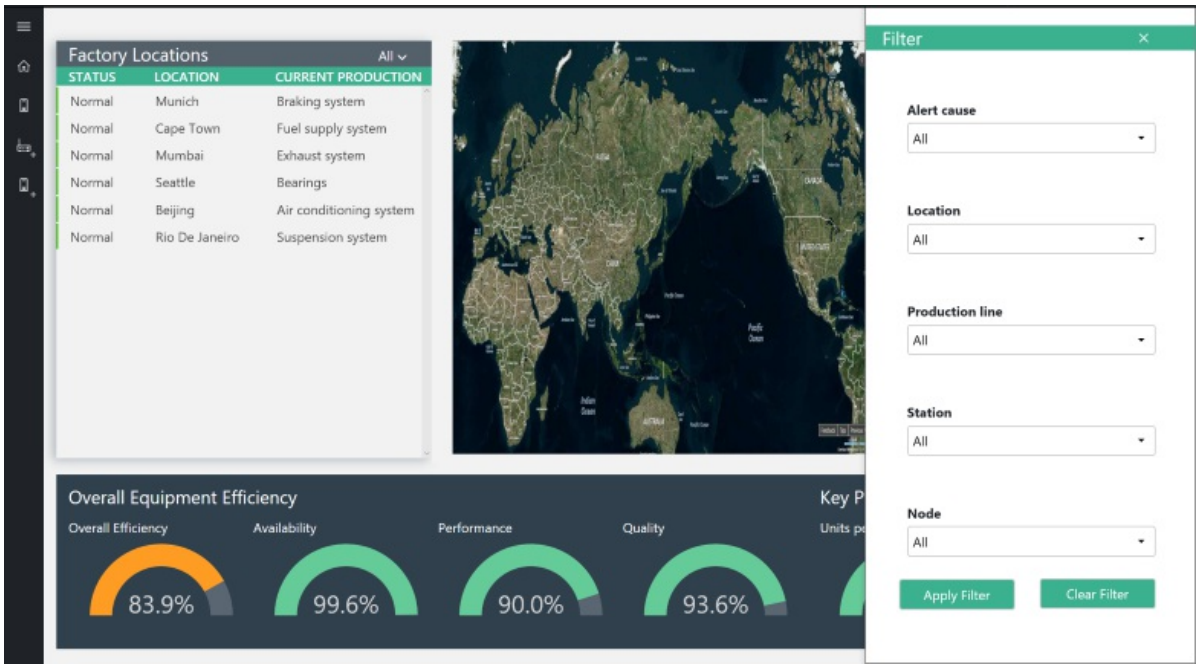
In this scenario, you monitored your factories OEE and KPIs values, in the dashboard. You then used Time Series Insights to provide more information to help drill further into the telemetry data for OEE and KPIs to help with detecting anomalies. You also used the alert panel to view issues with your factories and you used the actions available to you to resolve the alert.

## Other features

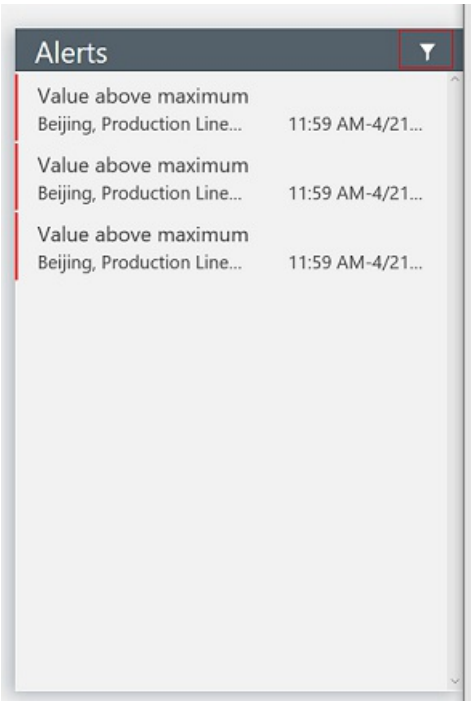
The following sections describe some additional features of the connected factory solution that are not described in the previous scenario.

# Apply filters

1. Click the **chevron** to display a list of available filters in either the factory locations panel or the alerts panel.
2. The filters panel is displayed for you.




3. Choose the filter that you require. It is also possible to type free text into the filter fields.
4. The filter is then applied for you. The filter state is also shown in the dashboard via a funnel that displays in the factories and alerts tables.




> **NOTE**
> An active filter does not affect the displayed OEE and KPI values, it only filters the list contents.

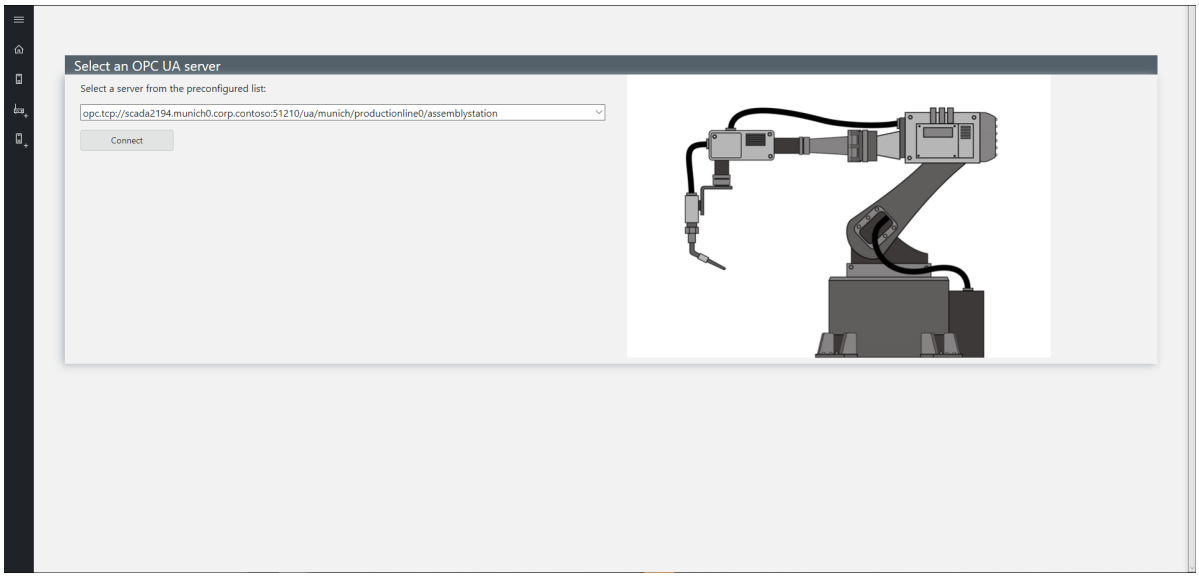
5. To clear a filter, click the funnel and click filter in the filter context panel. The text **All** is displayed in the factories and alerts tables.

# Browse an OPC UA server

When you deploy the preconfigured solution, you automatically provision simulated OPC UA servers that you can browse via the solution browser. These servers are *simulated OPC UA servers*. Simulated servers make it easy for you to experiment with the preconfigured solution without the need to deploy real, physical servers. If you do want to connect a real OPC UA server to the solution, see the Connect your OPC UA device to the connected factory preconfigured solution tutorial.

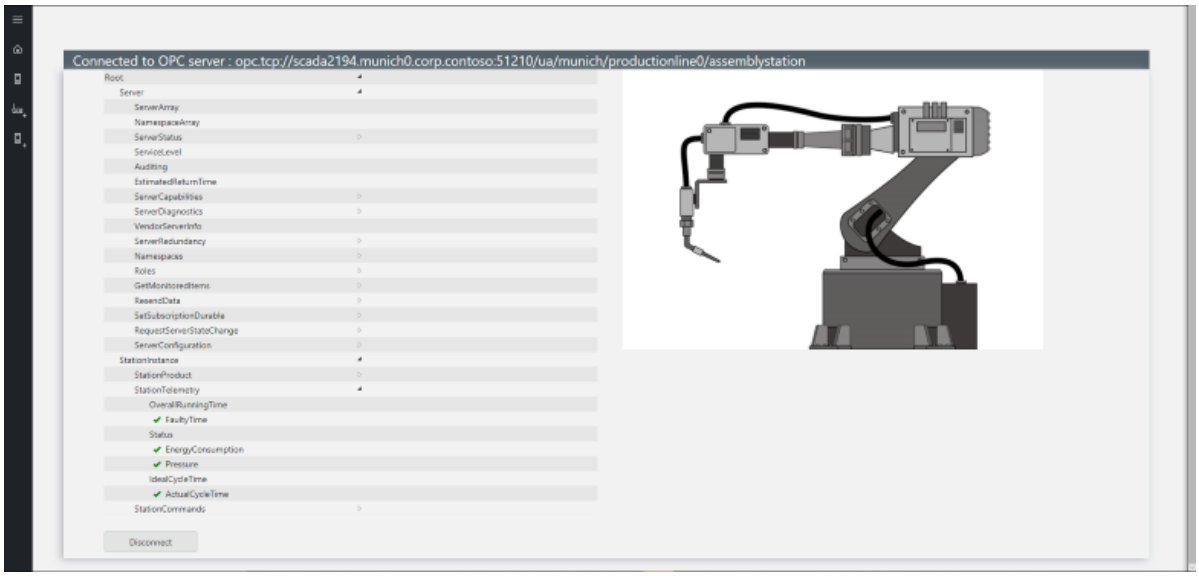
1. Click the **factory icon** in the dashboard navigation bar.


2. Choose one of the servers from the preconfigured list. This list shows the servers that are deployed for you in the preconfigured solution.




3. Click **Connect**, a security dialog displays. For the simulation, it is safe to click **Proceed**

4. To expand any of the nodes in the server tree, click it. Nodes that are publishing telemetry have a tick mark beside them.

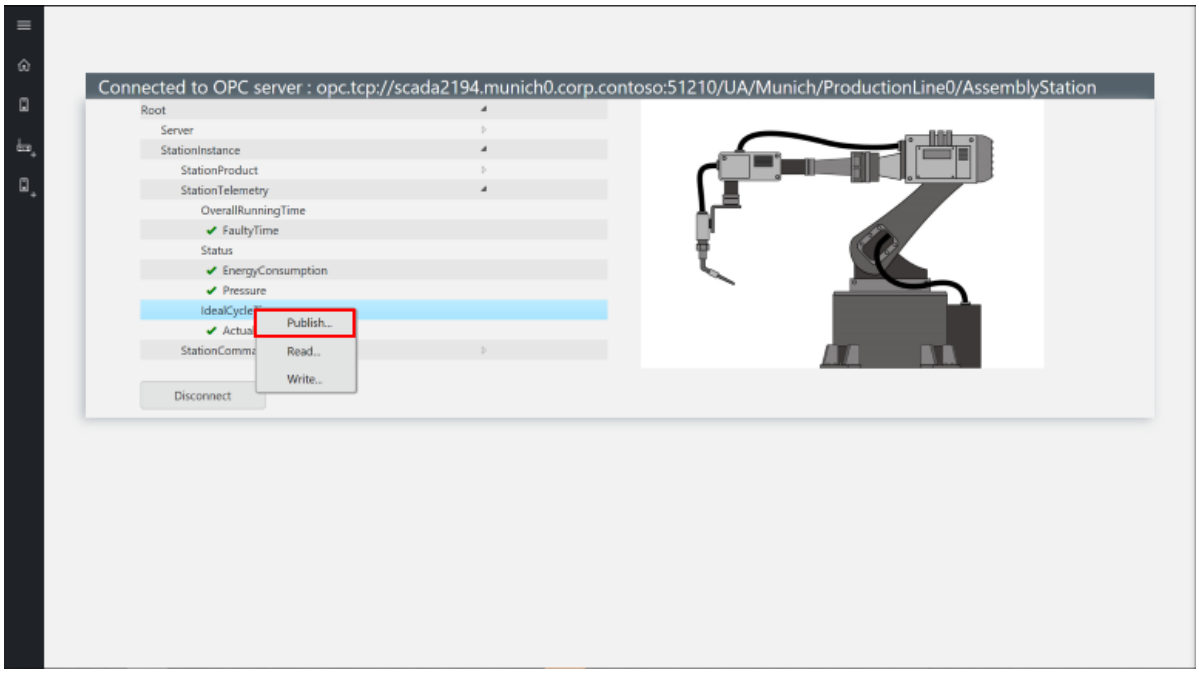



5. Right-click an item to read, write, publish, or call that node. The actions available to you depend on your permissions and the attributes of the node. The read option to displays a context panel showing the value of the specific node. The write option displays a context panel where you can enter a new value. The call option displays a node where you can enter the parameters for the call.

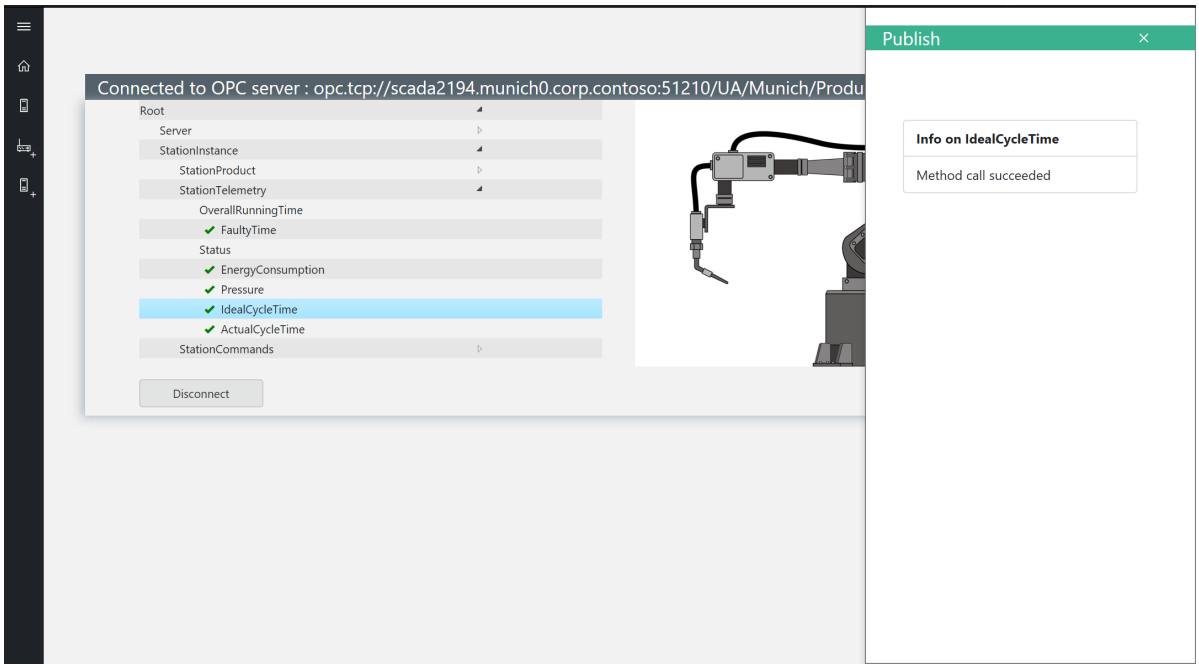
## Publish a node

When you browse a *simulated OPC UA server*, you can also choose to publish new nodes. You can analyze the telemetry from these nodes in the solution. These *simulated OPC UA servers* make it easy to experiment with the preconfigured solution without deploying real, physical devices.

1. Browse to a node in the OPC UA server browser tree that you wish to publish.
2. Right-click the node.
3. Choose **Publish**.



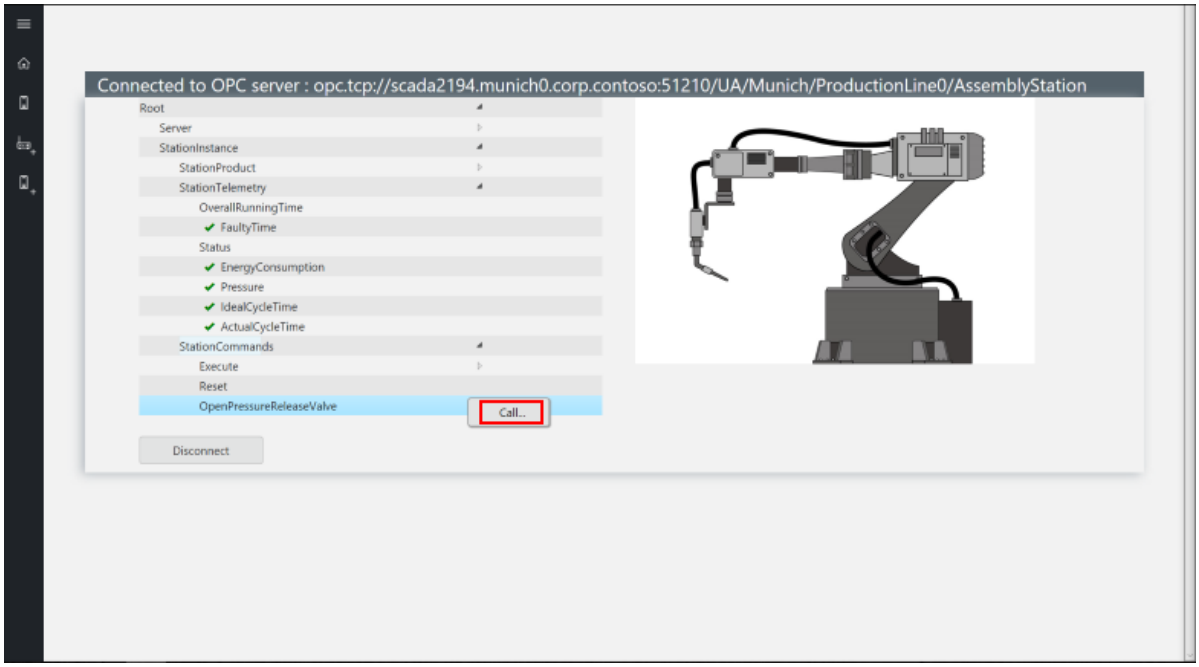

4. A context panel appears which tells you that the publish has succeeded. The node appears in the station level view with a check mark beside it.




## Command and control

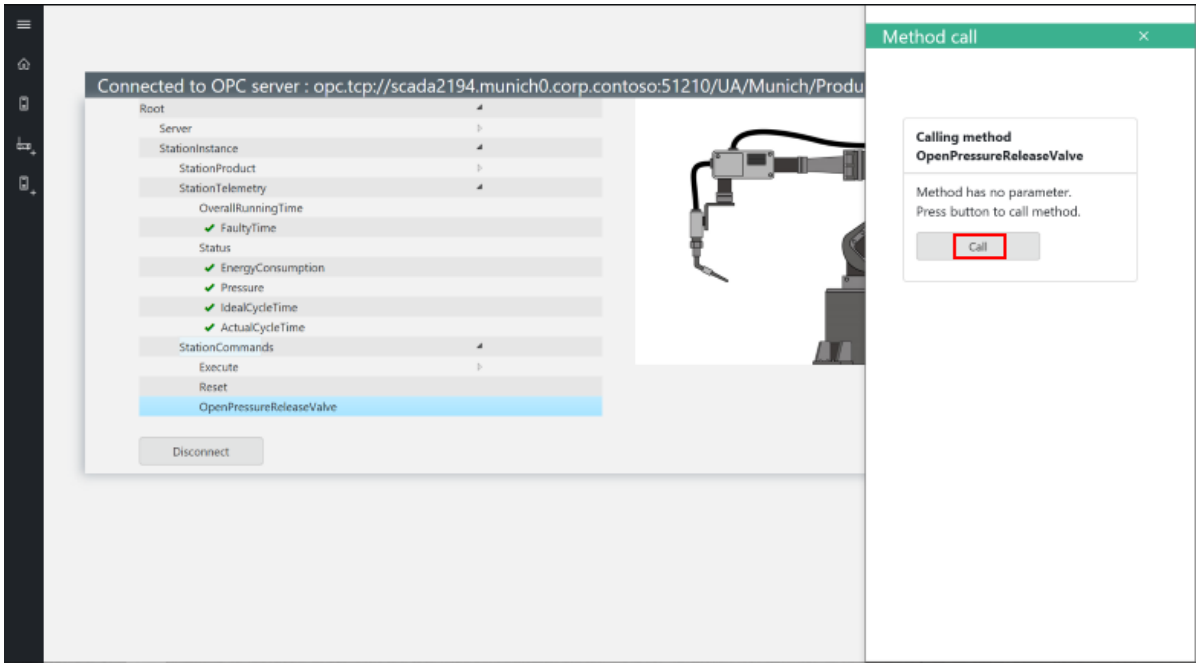
The connected factory allows you command and control your industry devices directly from the cloud. You can use this feature to respond to alerts generated by the device. For example, you could send a command to the device from the cloud. You can find the available commands in the **StationCommands** node in the OPC UA servers browser tree. In this scenario, you open a pressure release valve on the assembly station of a production line in Munich. To use the command and control functionality, you must be in the **Administrator** role for the preconfigured solution deployment.

1. Browse to the **StationCommands** node in the OPC UA server browser tree.
2. Choose the command that you wish use.
3. Right-click the node.
4. Choose **Call**.

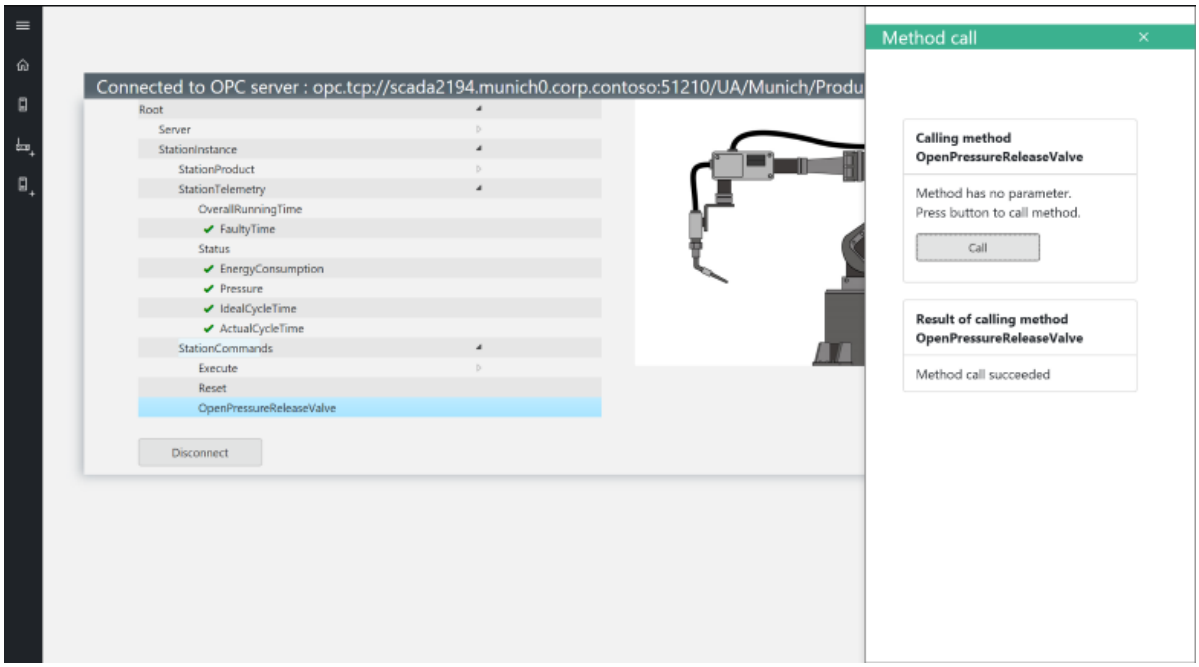



5. A context panel appears informing you which method you are about to call and any parameter details is applicable.

6. Choose **Call**.

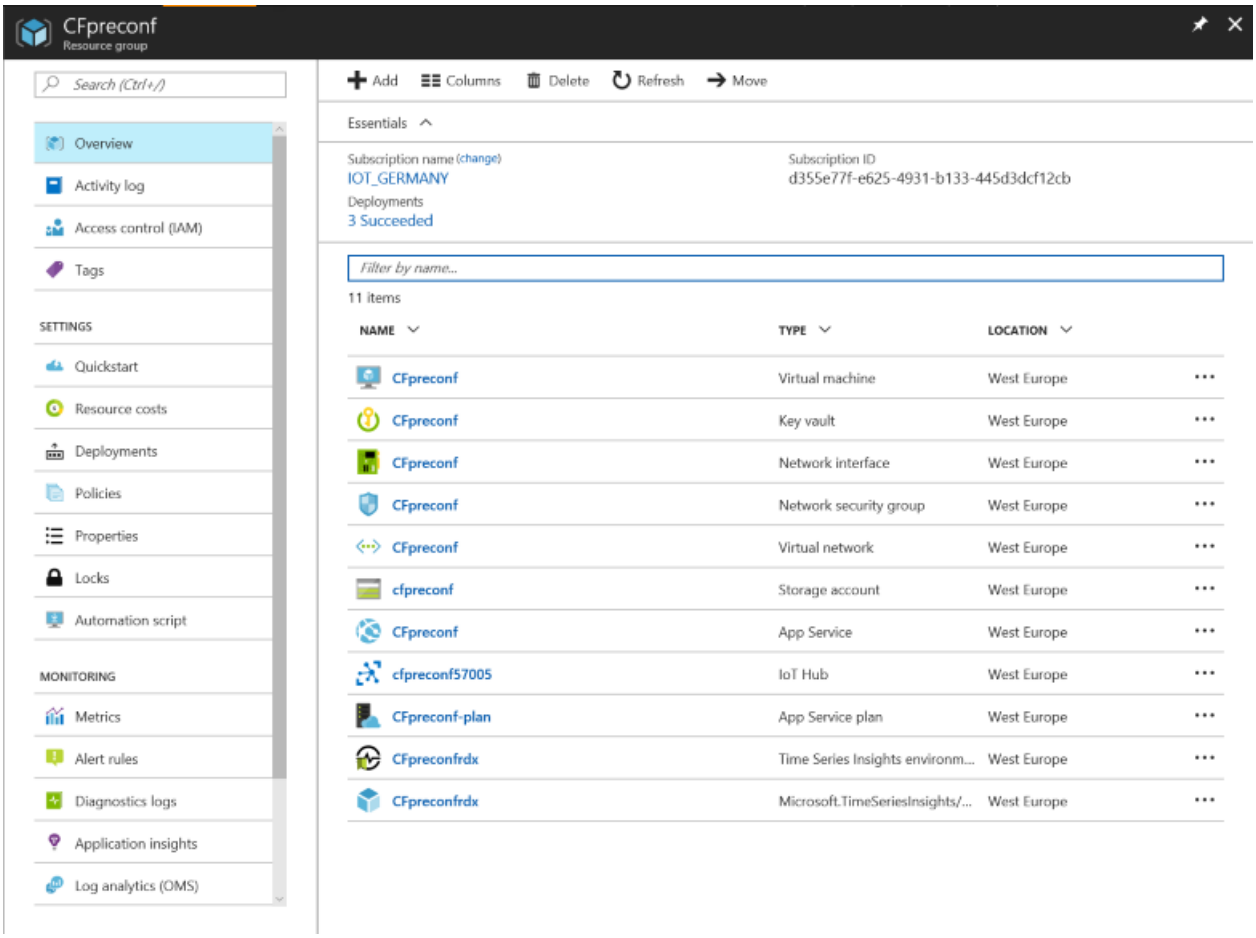



7. The context panel is updated to inform you that the method call succeeded. You can verify the call succeeded by reading the value of the pressure node that updated as a result of the call.




## Behind the scenes

When you deploy a preconfigured solution, the deployment process creates multiple resources in the Azure subscription you selected. You can view these resources in the Azure portal. The deployment process creates a **resource group** with a name based on the name you choose for your preconfigured solution:




You can view the settings of each resource by selecting it in the list of resources in the resource group.

You can also view the source code for the preconfigured solution. The connected factory preconfigured solution source code is in the azure-iot-connected-factory GitHub repository:

When you are done, you can delete the preconfigured solution from your Azure subscription on the azureiotsuite.com site. This site enables you to easily delete all the resources that were provisioned when you created the preconfigured solution.

> **NOTE**
> To ensure that you delete everything related to the preconfigured solution, delete it on the azureiotsuite.com site. Do not delete the resource group in the portal.

## Next Steps

Now that you've deployed a working preconfigured solution, you can continue getting started with IoT Suite by reading the following articles:

- Connected factory preconfigured solution walkthrough
- Connect your device to the Connected factory preconfigured solution
- Permissions on the azureiotsuite.com site

# Remote monitoring preconfigured solution walkthrough

8/24/2017 • 9 min to read • Edit Online

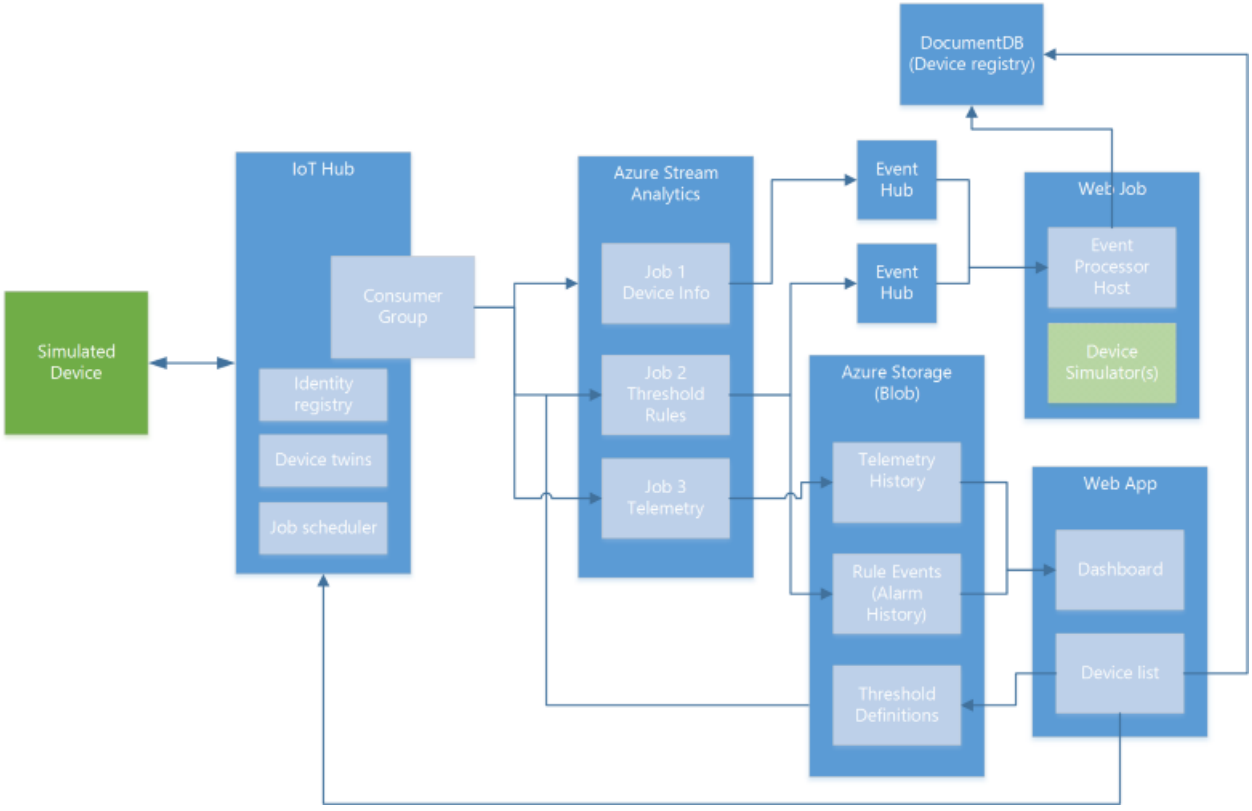
The IoT Suite remote monitoring preconfigured solution is an implementation of an end-to-end monitoring solution for multiple machines running in remote locations. The solution combines key Azure services to provide a generic implementation of the business scenario. You can use the solution as a starting point for your own implementation and customize it to meet your own specific business requirements.

This article walks you through some of the key elements of the remote monitoring solution to enable you to understand how it works. This knowledge helps you to:

- Troubleshoot issues in the solution.
- Plan how to customize to the solution to meet your own specific requirements.
- Design your own IoT solution that uses Azure services.

## Logical architecture

The following diagram outlines the logical components of the preconfigured solution:




## Simulated devices

In the preconfigured solution, the simulated device represents a cooling device (such as a building air conditioner or facility air handling unit). When you deploy the preconfigured solution, you also automatically provision four simulated devices that run in an Azure WebJob. The simulated devices make it easy for you to explore the behavior of the solution without the need to deploy any physical devices. To deploy a real physical device, see the Connect your device to the remote monitoring preconfigured solution tutorial.

### Device-to-cloud messages

Each simulated device can send the following message types to IoT Hub:

| MESSAGE | DESCRIPTION |
|---|---|
| Startup | When the device starts, it sends a **device-info** message containing information about itself to the back end. This data includes the device id and a list of the commands and methods the device supports. |
| Presence | A device periodically sends a **presence** message to report whether the device can sense the presence of a sensor. |
| Telemetry | A device periodically sends a **telemetry** message that reports simulated values for the temperature and humidity collected from the device's simulated sensors. |

> **NOTE**
> The solution stores the list of commands supported by the device in a Cosmos DB database and not in the device twin.

### Properties and device twins

The simulated devices send the following device properties to the twin in the IoT hub as *reported properties*. The device sends reported properties at startup and in response to a **Change Device State** command or method.

| PROPERTY | PURPOSE |
|---|---|
| Config.TelemetryInterval | Frequency (seconds) the device sends telemetry |
| Config.TemperatureMeanValue | Specifies the mean value for the simulated temperature telemetry |
| Device.DeviceID | Id that is either provided or assigned when a device is created in the solution |
| Device.DeviceState | State reported by the device |
| Device.CreatedTime | Time the device was created in the solution |
| Device.StartupTime | Time the device was started |
| Device.LastDesiredPropertyChange | The version number of the last desired property change |
| Device.Location.Latitude | Latitude location of the device |
| Device.Location.Longitude | Longitude location of the device |
| System.Manufacturer | Device manufacturer |
| System.ModelNumber | Model number of the device |
| System.SerialNumber | Serial number of the device |

| PROPERTY | PURPOSE |
| --- | --- |
| System.FirmwareVersion | Current version of firmware on the device |
| System.Platform | Platform architecture of the device |
| System.Processor | Processor running the device |
| System.InstalledRAM | Amount of RAM installed on the device |

The simulator seeds these properties in simulated devices with sample values. Each time the simulator initializes a simulated device, the device reports the pre-defined metadata to IoT Hub as reported properties. Reported properties can only be updated by the device. To change a reported property, you set a desired property in solution portal. It is the responsibility of the device to:

1. Periodically retrieve desired properties from the IoT hub.
2. Update its configuration with the desired property value.
3. Send the new value back to the hub as a reported property.

From the solution dashboard, you can use *desired properties* to set properties on a device by using the device twin. Typically, a device reads a desired property value from the hub to update its internal state and report the change back as a reported property.

> **NOTE**
> The simulated device code only uses the **Desired.Config.TemperatureMeanValue** and **Desired.Config.TelemetryInterval** desired properties to update the reported properties sent back to IoT Hub. All other desired property change requests are ignored in the simulated device.

### Methods

The simulated devices can handle the following methods (direct methods) invoked from the solution portal through the IoT hub:

| METHOD | DESCRIPTION |
| --- | --- |
| InitiateFirmwareUpdate | Instructs the device to perform a firmware update |
| Reboot | Instructs the device to reboot |
| FactoryReset | Instructs the device to perform a factory reset |

Some methods use reported properties to report on progress. For example, the **InitiateFirmwareUpdate** method simulates running the update asynchronously on the device. The method returns immediately on the device, while the asynchronous task continues to send status updates back to the solution dashboard using reported properties.

### Commands

The simulated devices can handle the following commands (cloud-to-device messages) sent from the solution portal through the IoT hub:

| COMMAND | DESCRIPTION |
| --- | --- |
| PingDevice | Sends a *ping* to the device to check it is alive |

| COMMAND | DESCRIPTION |
| --- | --- |
| StartTelemetry | Starts the device sending telemetry |
| StopTelemetry | Stops the device from sending telemetry |
| ChangeSetPointTemp | Changes the set point value around which the random data is generated |
| DiagnosticTelemetry | Triggers the device simulator to send an additional telemetry value (externalTemp) |
| ChangeDeviceState | Changes an extended state property for the device and sends the device info message from the device |

> **NOTE**
> For a comparison of these commands (cloud-to-device messages) and methods (direct methods), see Cloud-to-device communications guidance.

## IoT Hub

The IoT hub ingests data sent from the devices into the cloud and makes it available to the Azure Stream Analytics (ASA) jobs. Each stream ASA job uses a separate IoT Hub consumer group to read the stream of messages from your devices.

The IoT hub in the solution also:

- Maintains an identity registry that stores the ids and authentication keys of all the devices permitted to connect to the portal. You can enable and disable devices through the identity registry.
- Sends commands to your devices on behalf of the solution portal.
- Invokes methods on your devices on behalf of the solution portal.
- Maintains device twins for all registered devices. A device twin stores the property values reported by a device. A device twin also stores desired properties, set in the solution portal, for the device to retrieve when it next connects.
- Schedules jobs to set properties for multiple devices or invoke methods on multiple devices.

## Azure Stream Analytics

In the remote monitoring solution, Azure Stream Analytics (ASA) dispatches device messages received by the IoT hub to other back-end components for processing or storage. Different ASA jobs perform specific functions based on the content of the messages.

**Job 1: Device Info** filters device information messages from the incoming message stream and sends them to an Event Hub endpoint. A device sends device information messages at startup and in response to a **SendDeviceInfo** command. This job uses the following query definition to identify **device-info** messages:

```
SELECT * FROM DeviceDataStream Partition By PartitionId WHERE  ObjectType = 'DeviceInfo'
```

This job sends its output to an Event Hub for further processing.

**Job 2: Rules** evaluates incoming temperature and humidity telemetry values against per-device thresholds. Threshold values are set in the rules editor available in the solution portal. Each device/value pair is stored by

timestamp in a blob which Stream Analytics reads in as **Reference Data**. The job compares any non-empty value against the set threshold for the device. If it exceeds the '>' condition, the job outputs an **alarm** event that indicates that the threshold is exceeded and provides the device, value, and timestamp values. This job uses the following query definition to identify telemetry messages that should trigger an alarm:

```
WITH AlarmsData AS
(
SELECT
     Stream.IoTHub.ConnectionDeviceId AS DeviceId,
     'Temperature' as ReadingType,
     Stream.Temperature as Reading,
     Ref.Temperature as Threshold,
     Ref.TemperatureRuleOutput as RuleOutput,
     Stream.EventEnqueuedUtcTime AS [Time]
FROM IoTTelemetryStream Stream
JOIN DeviceRulesBlob Ref ON Stream.IoTHub.ConnectionDeviceId = Ref.DeviceID
WHERE
     Ref.Temperature IS NOT null AND Stream.Temperature > Ref.Temperature

UNION ALL

SELECT
     Stream.IoTHub.ConnectionDeviceId AS DeviceId,
     'Humidity' as ReadingType,
     Stream.Humidity as Reading,
     Ref.Humidity as Threshold,
     Ref.HumidityRuleOutput as RuleOutput,
     Stream.EventEnqueuedUtcTime AS [Time]
FROM IoTTelemetryStream Stream
JOIN DeviceRulesBlob Ref ON Stream.IoTHub.ConnectionDeviceId = Ref.DeviceID
WHERE
     Ref.Humidity IS NOT null AND Stream.Humidity > Ref.Humidity
)

SELECT *
INTO DeviceRulesMonitoring
FROM AlarmsData

SELECT *
INTO DeviceRulesHub
FROM AlarmsData
```

The job sends its output to an Event Hub for further processing and saves details of each alert to blob storage from where the solution portal can read the alert information.

**Job 3: Telemetry** operates on the incoming device telemetry stream in two ways. The first sends all telemetry messages from the devices to persistent blob storage for long-term storage. The second computes average, minimum, and maximum humidity values over a five-minute sliding window and sends this data to blob storage. The solution portal reads the telemetry data from blob storage to populate the charts. This job uses the following query definition:

```
WITH
    [StreamData]
AS (
    SELECT
        *
    FROM [IoTHubStream]
    WHERE
        [ObjectType] IS NULL -- Filter out device info and command responses
)

SELECT
    IoTHub.ConnectionDeviceId AS DeviceId,
    Temperature,
    Humidity,
    ExternalTemperature,
    EventProcessedUtcTime,
    PartitionId,
    EventEnqueuedUtcTime,
    *
INTO
    [Telemetry]
FROM
    [StreamData]

SELECT
    IoTHub.ConnectionDeviceId AS DeviceId,
    AVG (Humidity) AS [AverageHumidity],
    MIN(Humidity) AS [MinimumHumidity],
    MAX(Humidity) AS [MaxHumidity],
    5.0 AS TimeframeMinutes
INTO
    [TelemetrySummary]
FROM [StreamData]
WHERE
    [Humidity] IS NOT NULL
GROUP BY
    IoTHub.ConnectionDeviceId,
    SlidingWindow (mi, 5)
```

## Event Hubs

The **device info** and **rules** ASA jobs output their data to Event Hubs to reliably forward on to the **Event Processor** running in the WebJob.

## Azure storage

The solution uses Azure blob storage to persist all the raw and summarized telemetry data from the devices in the solution. The portal reads the telemetry data from blob storage to populate the charts. To display alerts, the solution portal reads the data from blob storage that records when telemetry values exceeded the configured threshold values. The solution also uses blob storage to record the threshold values you set in the solution portal.

## WebJobs

In addition to hosting the device simulators, the WebJobs in the solution also host the **Event Processor** running in an Azure WebJob that handles command responses. It uses command response messages to update the device command history (stored in the Cosmos DB database).

## Cosmos DB

The solution uses a Cosmos DB database to store information about the devices connected to the solution. This

information includes the history of commands sent to devices from the solution portal and of methods invoked from the solution portal.

## Solution portal

The solution portal is a web app deployed as part of the preconfigured solution. The key pages in the solution portal are the dashboard and the device list.

**Dashboard**

This page in the web app uses PowerBI javascript controls (See PowerBI-visuals repo) to visualize the telemetry data from the devices. The solution uses the ASA telemetry job to write the telemetry data to blob storage.

**Device list**

From this page in the solution portal you can:

- Provision a new device. This action sets the unique device id and generates the authentication key. It writes information about the device to both the IoT Hub identity registry and the solution-specific Cosmos DB database.
- Manage device properties. This action includes viewing existing properties and updating with new properties.
- Send commands to a device.
- View the command history for a device.
- Enable and disable devices.

## Next steps

The following TechNet blog posts provide more detail about the remote monitoring preconfigured solution:

- IoT Suite - Under The Hood - Remote Monitoring
- IoT Suite - Remote Monitoring - Adding Live and Simulated Devices

You can continue getting started with IoT Suite by reading the following articles:

- Connect your device to the remote monitoring preconfigured solution
- Permissions on the azureiotsuite.com site

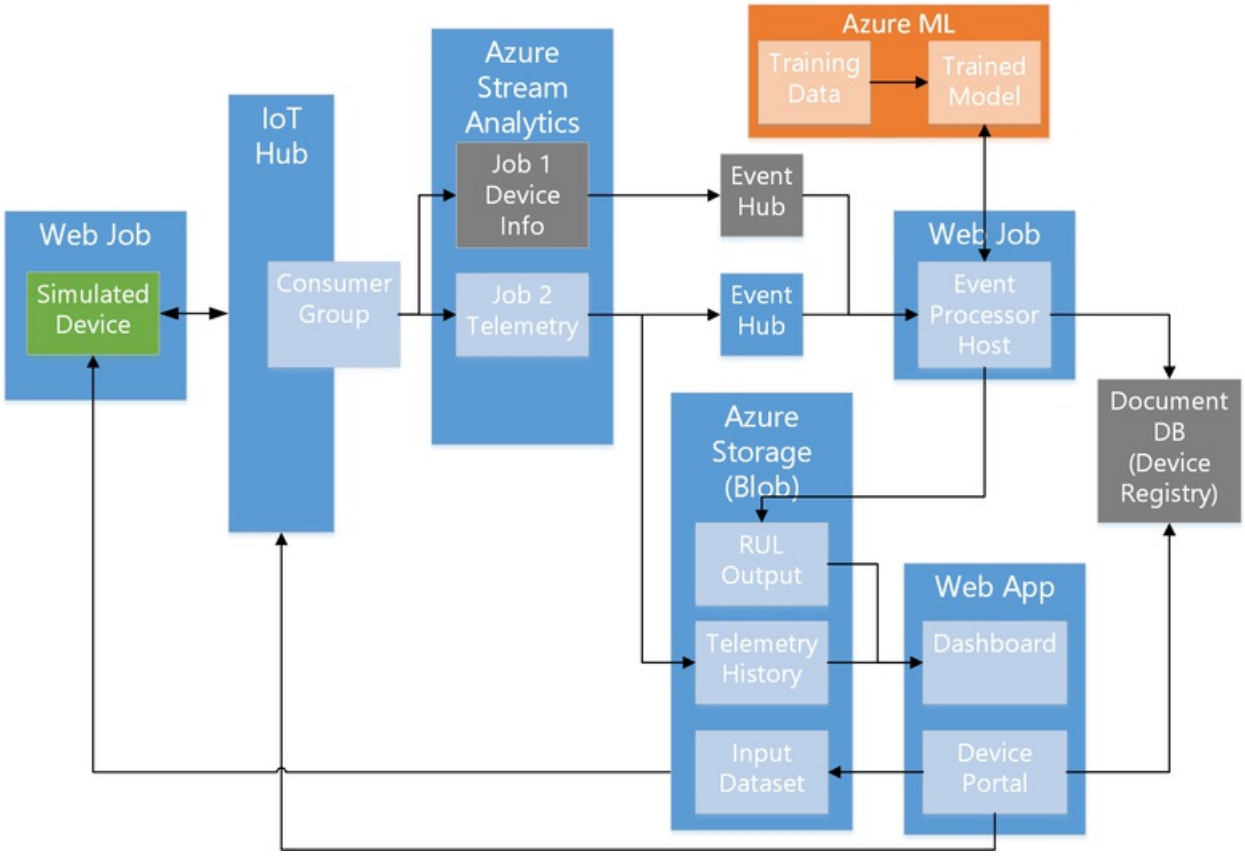
# Predictive maintenance preconfigured solution walkthrough

7/25/2017 • 3 min to read • Edit Online

The predictive maintenance preconfigured solution is an end-to-end solution for a business scenario that predicts the point at which a failure is likely to occur. You can use this preconfigured solution proactively for activities such as optimizing maintenance. The solution combines key Azure IoT Suite services, such as IoT Hub, Stream analytics, and an Azure Machine Learning workspace. This workspace contains a model, based on a public sample data set, to predict the Remaining Useful Life (RUL) of an aircraft engine. The solution fully implements the IoT business scenario as a starting point for you to plan and implement a solution that meets your own specific business requirements.

## Logical architecture

The following diagram outlines the logical components of the preconfigured solution:




The blue items are Azure services provisioned in the region where you deployed the preconfigured solution. The list of regions where you can deploy the preconfigured solution displays on the provisioning page.

The green item is a simulated device that represents an aircraft engine. You can learn more about these simulated devices in the following section.

The gray items represent components that implement *device management* capabilities. The current release of the predictive maintenance preconfigured solution does not provision these resources. To learn more about device management, refer to the remote monitoring pre-configured solution.

## Simulated devices

In the preconfigured solution, a simulated device represents an aircraft engine. The solution is provisioned with two engines that map to a single aircraft. Each engine emits four types of telemetry: Sensor 9, Sensor 11, Sensor 14, and Sensor 15 provide the data necessary for the Machine Learning model to calculate the RUL for the engine. Each simulated device sends the following telemetry messages to IoT Hub:

*Cycle count*. A cycle represents a completed flight with a duration between two and ten hours. During the flight, telemetry data is captured every half hour.

*Telemetry*. There are four sensors that represent engine attributes. The sensors are generically labeled Sensor 9, Sensor 11, Sensor 14, and Sensor 15. These four sensors represent telemetry sufficient to obtain useful results from the RUL model. The model used in the preconfigured solution is created from a public data set that includes real engine sensor data. For more information on how the model was created from the original data set, see the Cortana Intelligence Gallery Predictive Maintenance Template.

The simulated devices can handle the following commands sent from the IoT hub in the solution:

| COMMAND | DESCRIPTION |
| --- | --- |
| StartTelemetry | Controls the state of the simulation.<br>Starts the device sending telemetry |
| StopTelemetry | Controls the state of the simulation.<br>Stops the device sending telemetry |

IoT Hub provides device command acknowledgment.

## Azure Stream Analytics job

**Job: Telemetry** operates on the incoming device telemetry stream using two statements:

- The first selects all telemetry from the devices and sends this data to blob storage. From here, it is visualized in the web app.
- The second computes average sensor values over a two-minute sliding window and sends this data through the Event hub to an **event processor**.

## Event processor

The **event processor host** runs in an Azure Web Job. The **event processor** takes the average sensor values for a completed cycle. It then passes those values to an API that exposes trained model to calculate the RUL for an engine. The API is exposed by a Machine Learning workspace that is provisioned as part of the solution.

## Machine Learning

The Machine Learning component uses a model derived from data collected from real aircraft engines. You can navigate to the Machine Learning workspace from the tile on the azureiotsuite.com page for your provisioned solution. The tile is available when the solution is in the **Ready** state.

## Next steps

Now you've seen the key components of the predictive maintenance preconfigured solution, you may want to customize it. See Guidance on customizing preconfigured solutions.

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Frequently asked questions for IoT Suite
- IoT security from the ground up

# Connected factory preconfigured solution walkthrough

8/24/2017 • 5 min to read • Edit Online

The IoT Suite connected factory preconfigured solution is an implementation of an end-to-end industrial solution that:

- Connects to both simulated industrial devices running OPC UA servers in simulated factory production lines, and real OPC UA server devices. For more information about OPC UA, see the Connected factory FAQ.
- Shows operational KPIs and OEE of those devices and production lines.
- Demonstrates how a cloud-based application could be used to interact with OPC UA server systems.
- Enables you to connect your own OPC UA server devices.
- Enables you to browse and modify the OPC UA server data.
- Integrates with the Azure Time Series Insights (TSI) service to provide customized views of the data from your OPC UA servers.

You can use the solution as a starting point for your own implementation and customize it to meet your own specific business requirements.

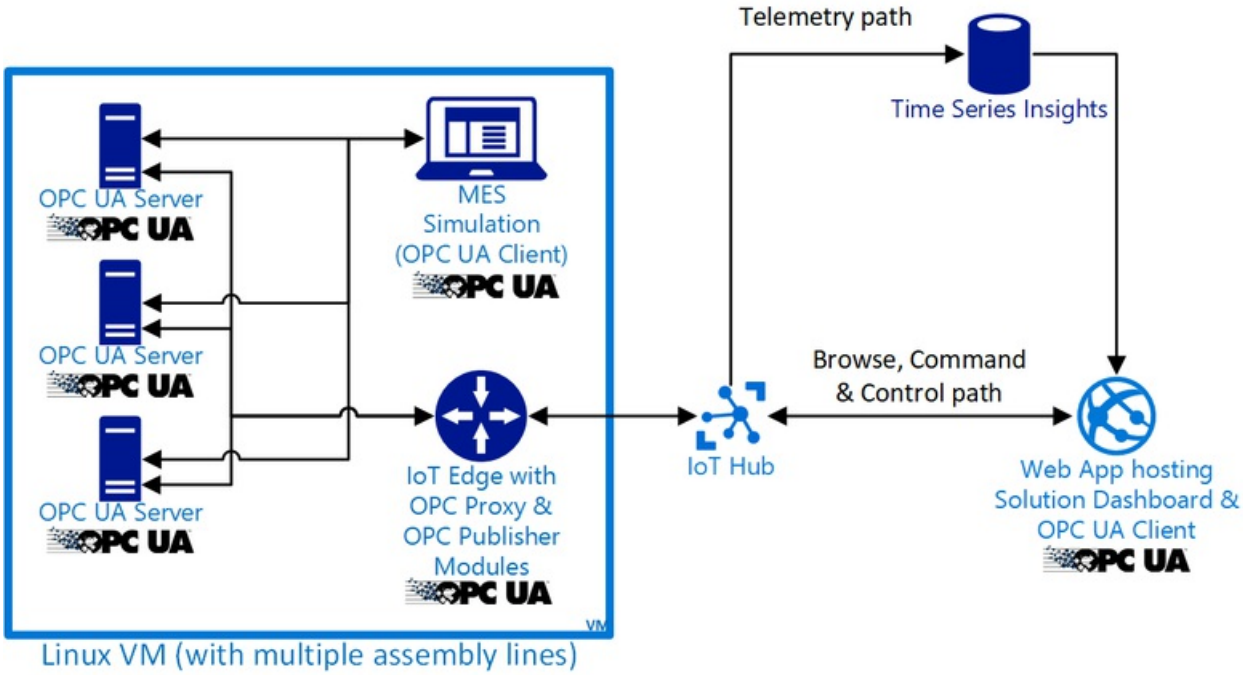
This article walks you through some of the key elements of the connected factory solution to enable you to understand how it works. This knowledge helps you to:

- Troubleshoot issues in the solution.
- Plan how to customize to the solution to meet your own specific requirements.
- Design your own IoT solution that uses Azure services.

For more information, see the Connected factory FAQ.

## Logical architecture

The following diagram outlines the logical components of the preconfigured solution:



## Communication patterns

The solution uses the OPC UA Pub/Sub specification to send OPC UA telemetry data to IoT Hub in JSON format. The solution uses the OPC Publisher IoT Edge module for this purpose.

The solution also has an OPC UA client integrated into a web application that can establish connections with on-premises OPC UA servers. The client uses a reverse-proxy and receives help from IoT Hub to make the connection without requiring open ports in the on-premises firewall. This communication pattern is called service-assisted communication. The solution uses the OPC Proxy IoT Edge module for this purpose.

## Simulation

The simulated stations and the simulated manufacturing execution systems (MES) make up a factory production line. The simulated devices and the OPC Publisher Module are based on the OPC UA .NET Standard published by the OPC Foundation.

The OPC Proxy and OPC Publisher are implemented as modules based on Azure IoT Edge. Each simulated production line has a designated gateway attached.

All simulation components run in Docker containers hosted in an Azure Linux VM. The simulation is configured to run eight simulated production lines by default.

## Simulated production line

A production line manufactures parts. It is composed of different stations: an assembly station, a test station, and a packaging station.

The simulation runs and updates the data that is exposed through the OPC UA nodes. All simulated production line stations are orchestrated by the MES through OPC UA.

## Simulated manufacturing execution system

The MES monitors each station in the production line through OPC UA to detect station status changes. It calls OPC UA methods to control the stations and passes a product from one station to the next until it is complete.

## Gateway OPC publisher module

OPC Publisher Module connects to the station OPC UA servers and subscribes to the OPC nodes to be published. The module converts the node data into JSON format, encrypts it, and sends it to IoT Hub as OPC UA Pub/Sub messages.

The OPC Publisher module only requires an outbound https port (443) and can work with existing enterprise infrastructure.

## Gateway OPC proxy module

The Gateway OPC UA Proxy Module tunnels binary OPC UA command and control messages and only requires an outbound https port (443). It can work with existing enterprise infrastructure, including Web Proxies.

It uses IoT Hub Device methods to transfer packetized TCP/IP data at the application layer and thus ensures endpoint trust, data encryption, and integrity using SSL/TLS.

The OPC UA binary protocol relayed through the proxy itself uses UA authentication and encryption.

## Azure Time Series Insights

The Gateway OPC Publisher Module subscribes to OPC UA server nodes to detect change in the data values. If a data change is detected in one of the nodes, this module then sends messages to Azure IoT Hub.

IoT Hub provides an event source to Azure TSI. TSI stores data for 30 days based on timestamps attached to the messages. This data includes:

- OPC UA ApplicationUri
- OPC UA NodeId
- Value of the node
- Source timestamp
- OPC UA DisplayName

Currently, TSI does not allow customers to customize how long they wish to keep the data for.

TSI queries against node data using a SearchSpan (Time.From, Time.To) and aggregates by OPC UA ApplicationUri or OPC UA NodeId or OPC UA DisplayName.

To retrieve the data for the OEE and KPI gauges, and the time series charts, data is aggregated by count of events, Sum, Avg, Min, and Max.

The time series are built using a different process. OEE and KPIs are calculated from station base data and bubbled up for the topology (production lines, factories, enterprise) in the application.

Additionally, time series for OEE and KPI topology is calculated in the app, whenever a displayed timespan is ready. For example, the day view is updated every full hour.

The time series view of node data comes directly from TSI using an aggregation for timespan.

## IoT Hub

The IoT hub receives data sent from the OPC Publisher Module into the cloud and makes it available to the Azure TSI service.

The IoT Hub in the solution also:

- Maintains an identity registry that stores the IDs for all OPC Publisher Module and all OPC Proxy Modules.
- Is used as transport channel for bidirectional communication of the OPC Proxy Module.

## Azure Storage

The solution uses Azure blob storage as disk storage for the VM and to store deployment data.

## Web app

The web app deployed as part of the preconfigured solution comprises of an integrated OPC UA client, alerts processing and telemetry visualization.

## Next steps

You can continue getting started with IoT Suite by reading the following articles:

- Permissions on the azureiotsuite.com site
- Deploy a gateway on Windows or Linux for the connected factory preconfigured solution

# Connect your Microsoft Azure IoT Raspberry Pi 3 Starter Kit to the remote monitoring solution

7/25/2017 • 1 min to read • Edit Online

The tutorials in this section help you learn how to connect a Raspberry Pi 3 device to the remote monitoring solution. Choose the tutorial appropriate to your preferred programming language and the whether you have the sensor hardware available to use with your Raspberry Pi.

## The tutorials

| TUTORIAL | NOTES | LANGUAGES |
| --- | --- | --- |
| Simulated telemetry (Basic) | Simulates sensor data. Uses a standalone Raspberry Pi. | C, Node.js |
| Real sensor (Intermediate) | Uses data from a BME280 sensor connected to your Raspberry Pi. | C, Node.js |
| Implement firmware update (Advanced) | Uses data from a BME280 sensor connected to your Raspberry Pi. Enables remote firmware updates on your Raspberry Pi. | C, Node.js |

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Raspberry Pi 3 to the remote monitoring solution and send simulated telemetry using C

7/25/2017 • 8 min to read • Edit Online

This tutorial shows you how to use the Raspberry Pi 3 to simulate temperature and humidity data to send to the cloud. The tutorial uses:

- Raspbian OS, the C programming language, and the Microsoft Azure IoT SDK for C to implement a sample device.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your device to communicate with your computer and the remote monitoring solution.
- Update the sample device code to connect to the remote monitoring solution, and send simulated telemetry that you can view on the solution dashboard.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Raspberry Pi.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility. For more information, see SSH Using Linux or Mac OS.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Raspberry Pi.

Microsoft IoT Starter Kit for Raspberry Pi 3 or equivalent components. This tutorial uses the following items from the kit:

- Raspberry Pi 3
- MicroSD Card (with NOOBS)
- A USB Mini cable
- An Ethernet cable

# Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

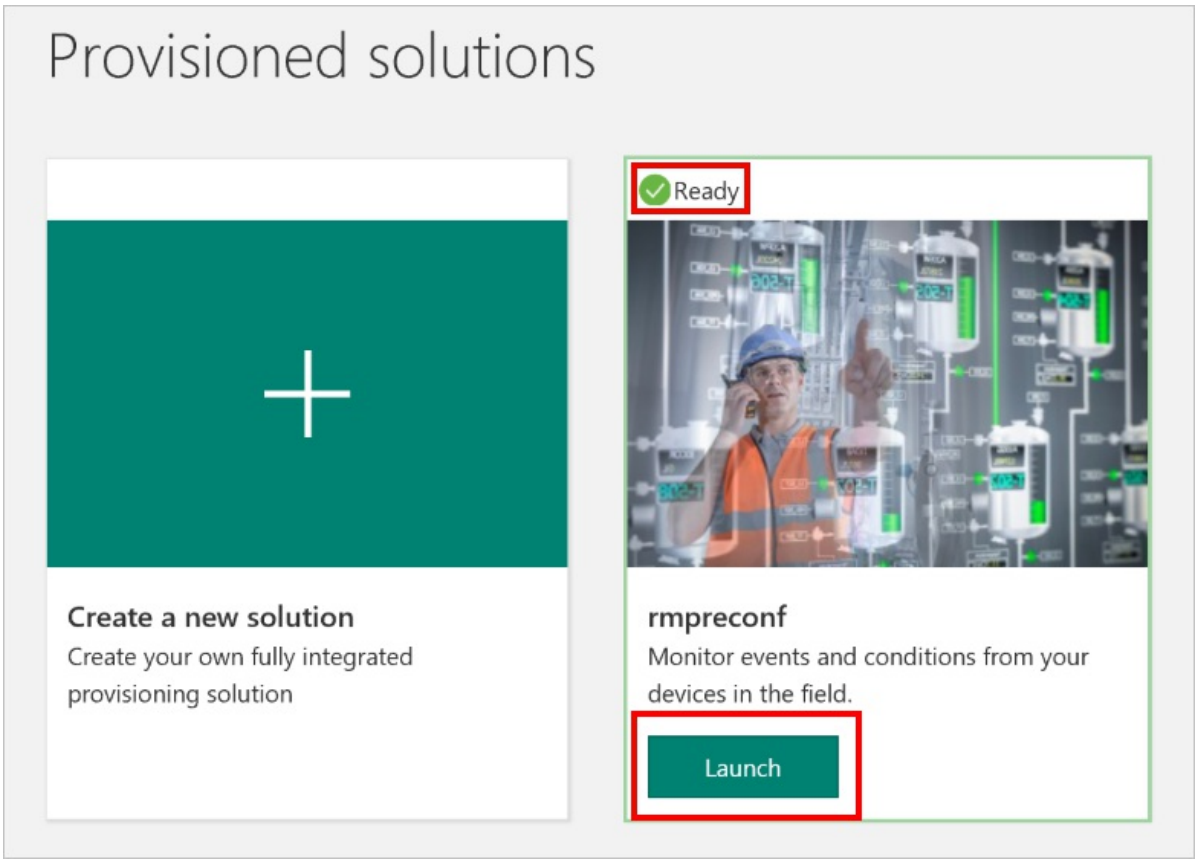
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

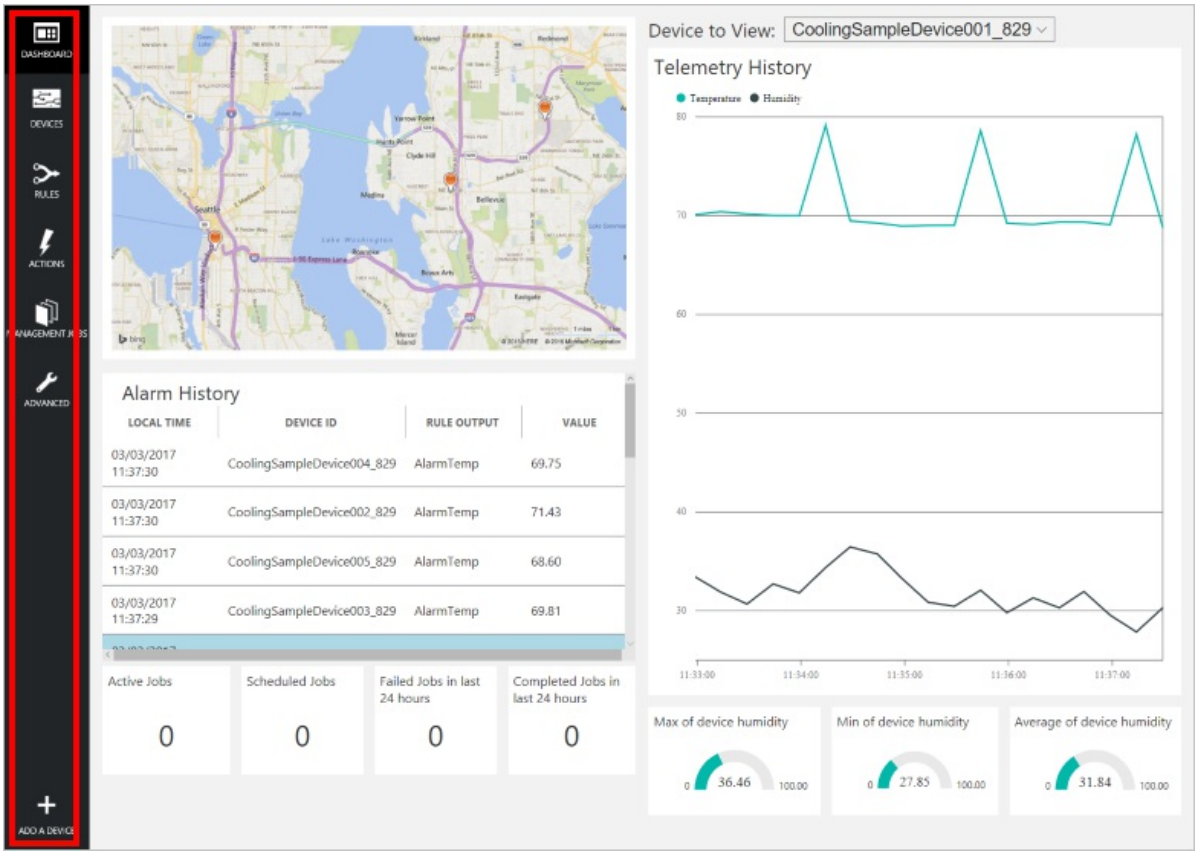
# View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.

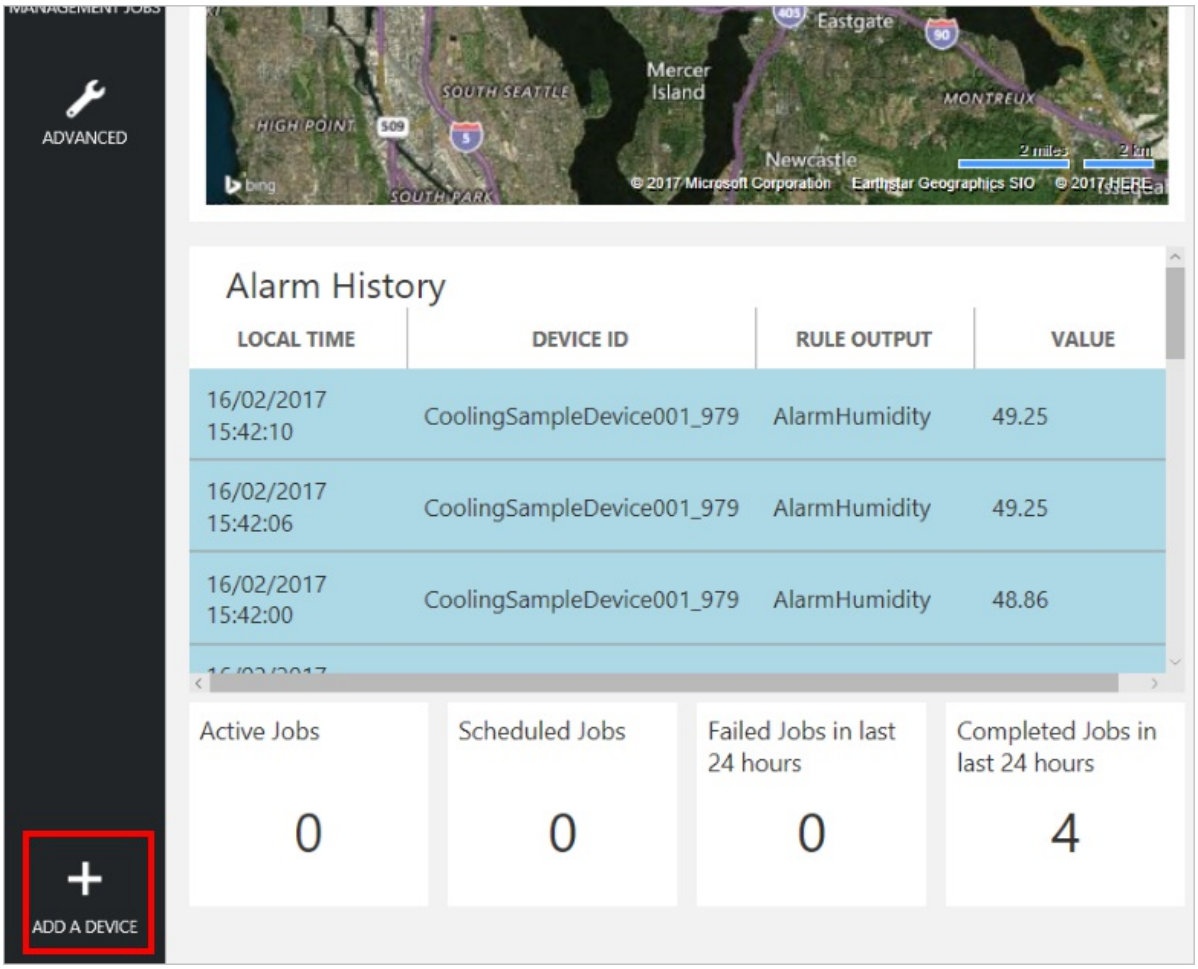



## Add a device

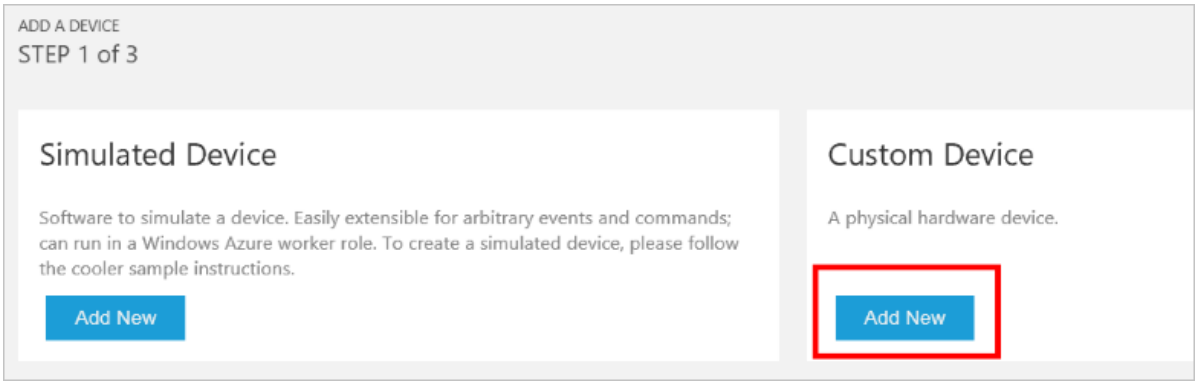
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

If you haven't already done so, add a custom device to your remote monitoring solution. Complete the following steps in the solution dashboard:

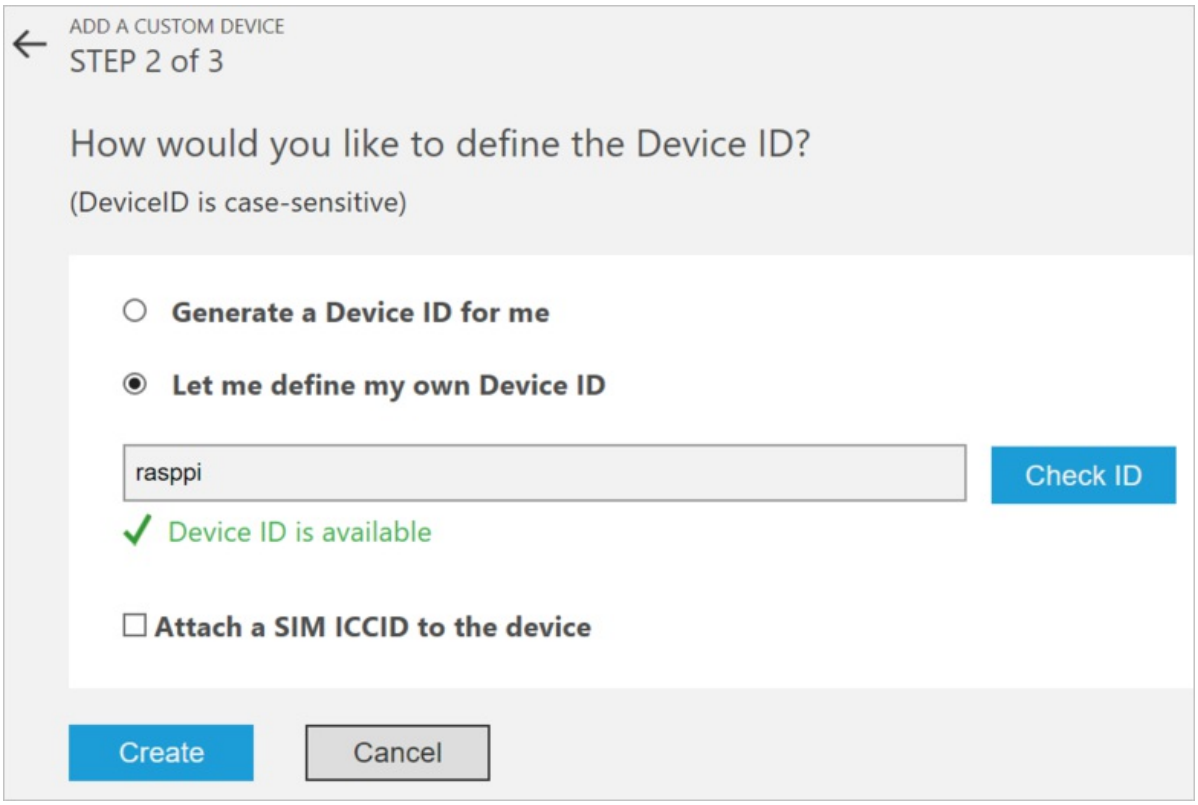
1. In the lower left-hand corner of the dashboard, click **Add a device**.



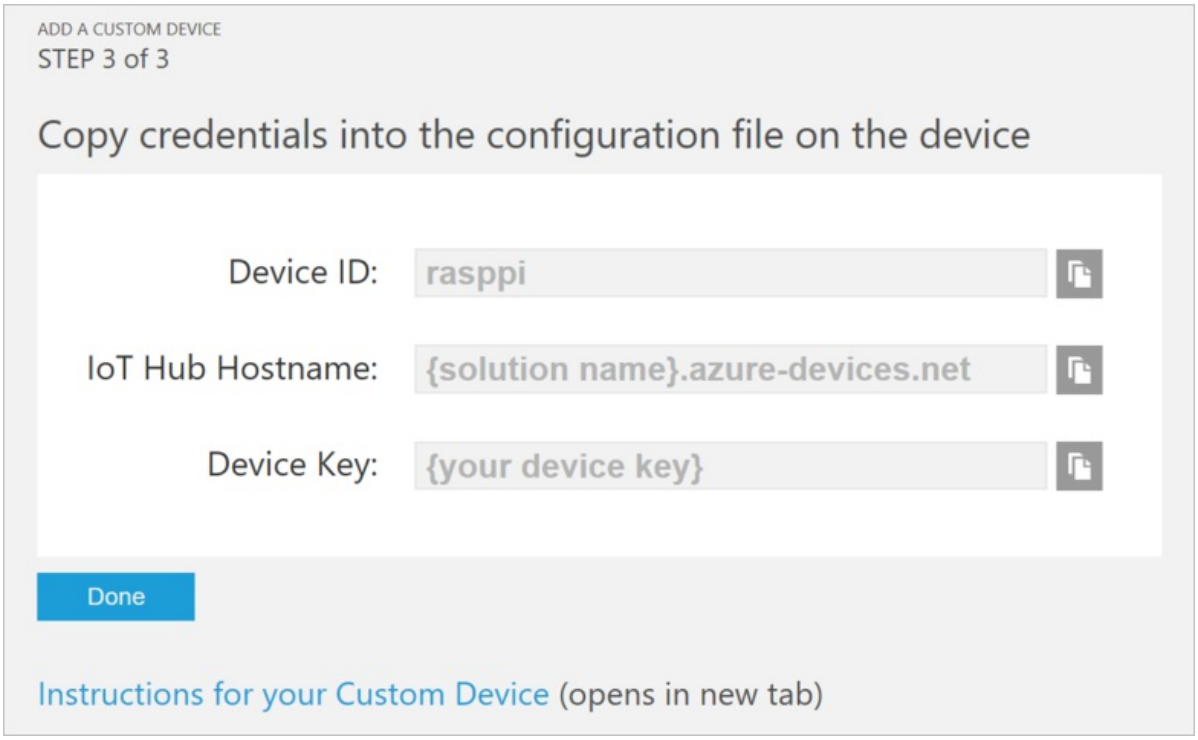

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **rasppi**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Raspberry Pi needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Raspberry Pi

**Install Raspbian**

If this is the first time you are using your Raspberry Pi, you need to install the Raspbian operating system using

NOOBS on the SD card included in the kit. The Raspberry Pi Software Guide describes how to install an operating system on your Raspberry Pi. This tutorial assumes you have installed the Raspbian operating system on your Raspberry Pi.

> **NOTE**
> The SD card included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 already has NOOBS installed. You can boot the Raspberry Pi from this card and choose to install the Raspbian OS.

To complete the hardware setup, you need to:

- Connect your Raspberry Pi to the power supply included in the kit.
- Connect your Raspberry Pi to your network using the Ethernet cable included in your kit. Alternatively, you can set up Wireless Connectivity for your Raspberry Pi.

You have now completed the hardware setup of your Raspberry Pi.

### Sign in and access the terminal

You have two options to access a terminal environment on your Raspberry Pi:

- If you have a keyboard and monitor connected to your Raspberry Pi, you can use the Raspbian GUI to access a terminal window.
- Access the command line on your Raspberry Pi using SSH from your desktop machine.

#### Use a terminal Window in the GUI

The default credentials for Raspbian are username **pi** and password **raspberry**. In the task bar in the GUI, you can launch the **Terminal** utility using the icon that looks like a monitor.

#### Sign in with SSH

You can use SSH for command-line access to your Raspberry Pi. The article SSH (Secure Shell) describes how to configure SSH on your Raspberry Pi, and how to connect from Windows or Linux & Mac OS.

Sign in with username **pi** and password **raspberry**.

#### Optional: Share a folder on your Raspberry Pi

Optionally, you may want to share a folder on your Raspberry Pi with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Raspberry Pi instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Raspberry Pi. Alternatively, use the built-in SFTP server with an SFTP client on your desktop.

## Download and configure the sample

You can now download and configure the remote monitoring client application on your Raspberry Pi.

### Clone the repositories

If you haven't already done so, clone the required repositories by running the following commands in a terminal on your Pi:

```
cd ~
git clone --recursive https://github.com/Azure-Samples/iot-remote-monitoring-c-raspberrypi-getstartedkit.git
```

### Update the device connection string

Open the sample source file in the **nano** editor using the following command:

```
nano ~/iot-remote-monitoring-c-raspberrypi-getstartedkit/simulator/remote_monitoring/remote_monitoring.c
```

Locate the following lines:

```
static const char* deviceId = "[Device Id]";
static const char* connectionString = "HostName=[IoTHub Name].azure-devices.net;DeviceId=[Device
Id];SharedAccessKey=[Device Key]";
```

Replace the placeholder values with the device and IoT Hub information you created and saved at the start of this tutorial. Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).

## Build the sample

Install the prerequisite packages for the Microsoft Azure IoT Device SDK for C by running the following commands in a terminal on the Raspberry Pi:

```
sudo apt-get update
sudo apt-get install g++ make cmake git libcurl4-openssl-dev libssl-dev uuid-dev
```

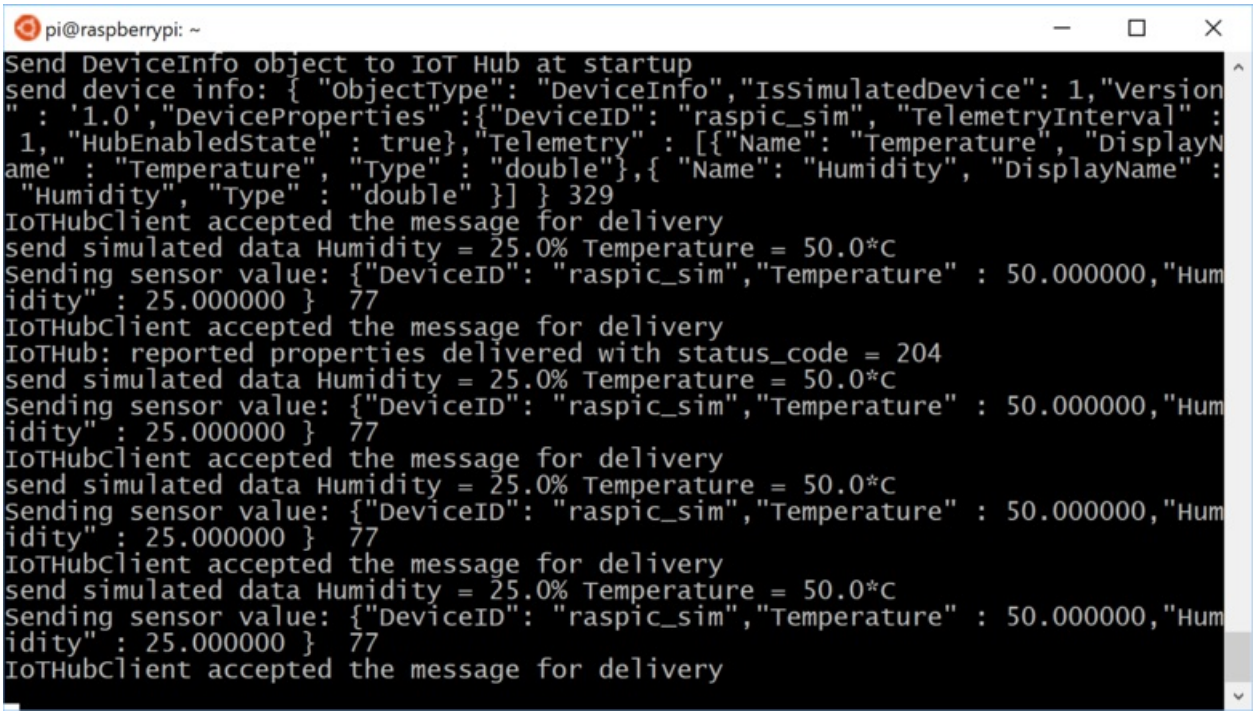
You can now build the updated sample solution on the Raspberry Pi:

```
chmod +x ~/iot-remote-monitoring-c-raspberrypi-getstartedkit/simulator/build.sh
~/iot-remote-monitoring-c-raspberrypi-getstartedkit/simulator/build.sh
```

You can now run the sample program on the Raspberry Pi. Enter the command:

```
sudo ~/cmake/remote_monitoring/remote_monitoring
```

The following sample output is an example of the output you see at the command prompt on the Raspberry Pi:

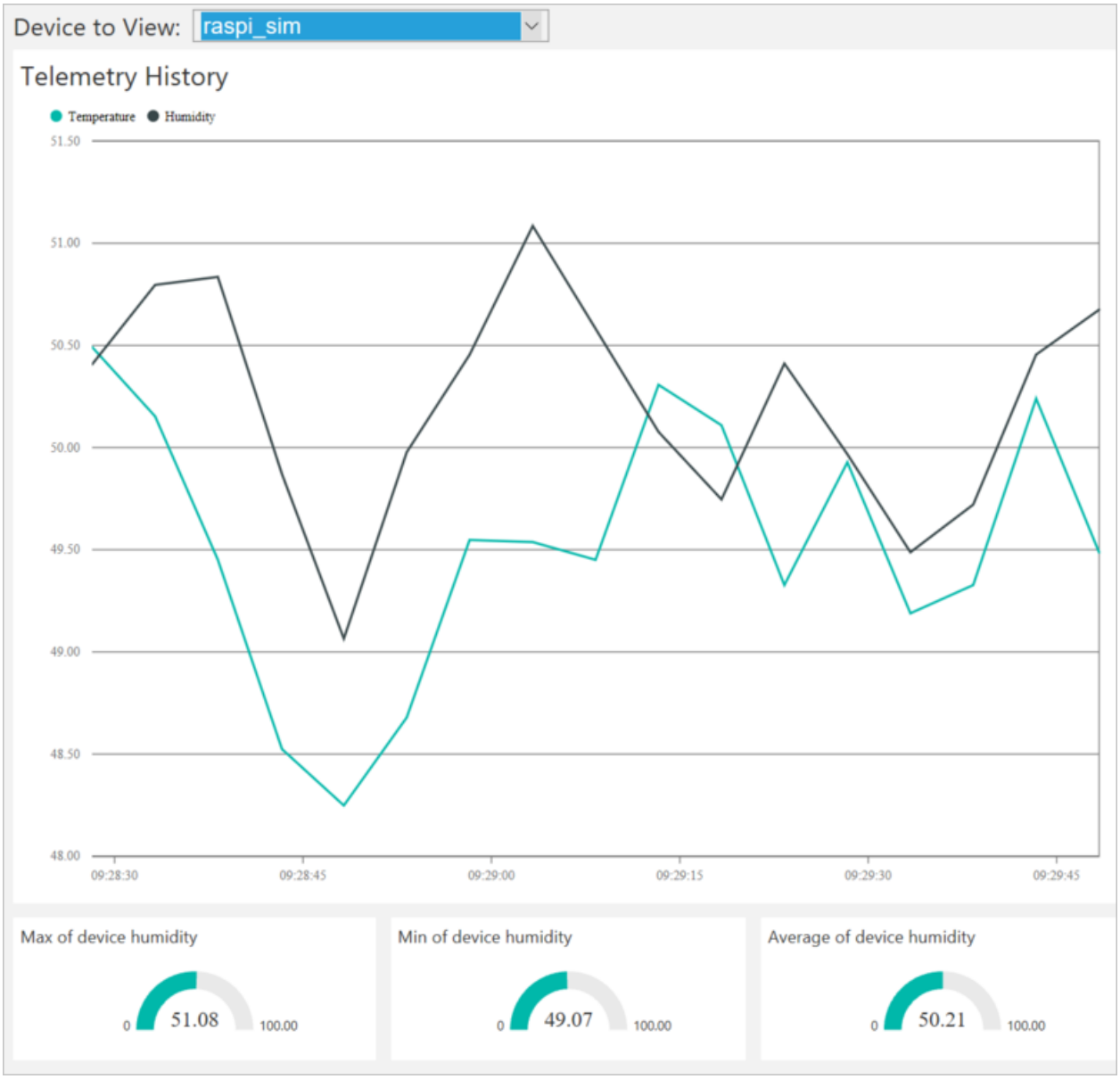



Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The Raspberry Pi is now sending telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send messages to your Raspberry Pi from the solution dashboard.

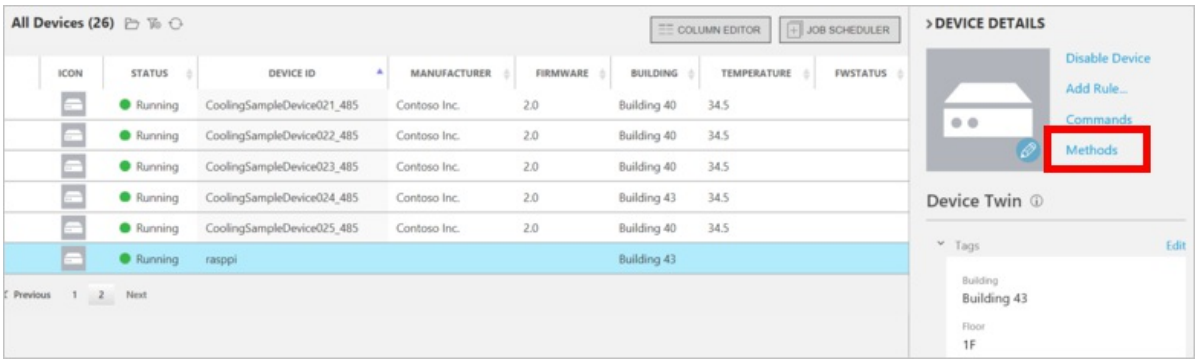
- Navigate to the solution dashboard.
- Select your device in the **Device to View** dropdown.
- The telemetry from the Raspberry Pi displays on the dashboard.



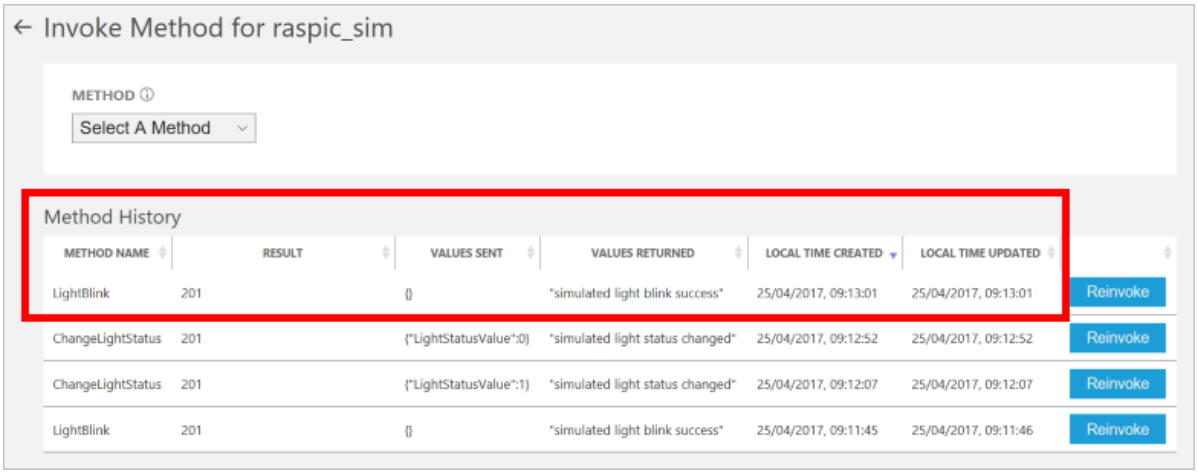

## Act on the device

From the solution dashboard, you can invoke methods on your Raspberry Pi. When the Raspberry Pi connects to the remote monitoring solution, it sends information about the methods it supports.

- In the solution dashboard, click **Devices** to visit the **Devices** page. Select your Raspberry Pi in the **Device List**. Then choose **Methods**:




- On the **Invoke Method** page, choose **LightBlink** in the **Method** dropdown.
- Choose **InvokeMethod**. The simulator prints a message in the console on the Raspberry Pi. The app on the Raspberry Pi sends an acknowledgment back to the solution dashboard:




- You can switch the LED on and off using the **ChangeLightStatus** method with a **LightStatusValue** set to **1** for on or **0** for off.

**WARNING**

If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Raspberry Pi 3 to the remote monitoring solution and send telemetry from a real sensor using C

7/25/2017 • 9 min to read • Edit Online

This tutorial shows you how to use the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to develop a temperature and humidity reader that can communicate with the cloud. The tutorial uses:

- Raspbian OS, the C programming language, and the Microsoft Azure IoT SDK for C to implement a sample device.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your device and sensors to communicate with your computer and the remote monitoring solution.
- Update the sample device code to connect to the remote monitoring solution, and send telemetry that you can view on the solution dashboard.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Raspberry Pi.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility. For more information, see SSH Using Linux or Mac OS.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Raspberry Pi.

Microsoft IoT Starter Kit for Raspberry Pi 3 or equivalent components. This tutorial uses the following items from the kit:

- Raspberry Pi 3
- MicroSD Card (with NOOBS)
- A USB Mini cable
- An Ethernet cable

- BME280 sensor
- Breadboard
- Jumper wires
- Resistors
- LEDs

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

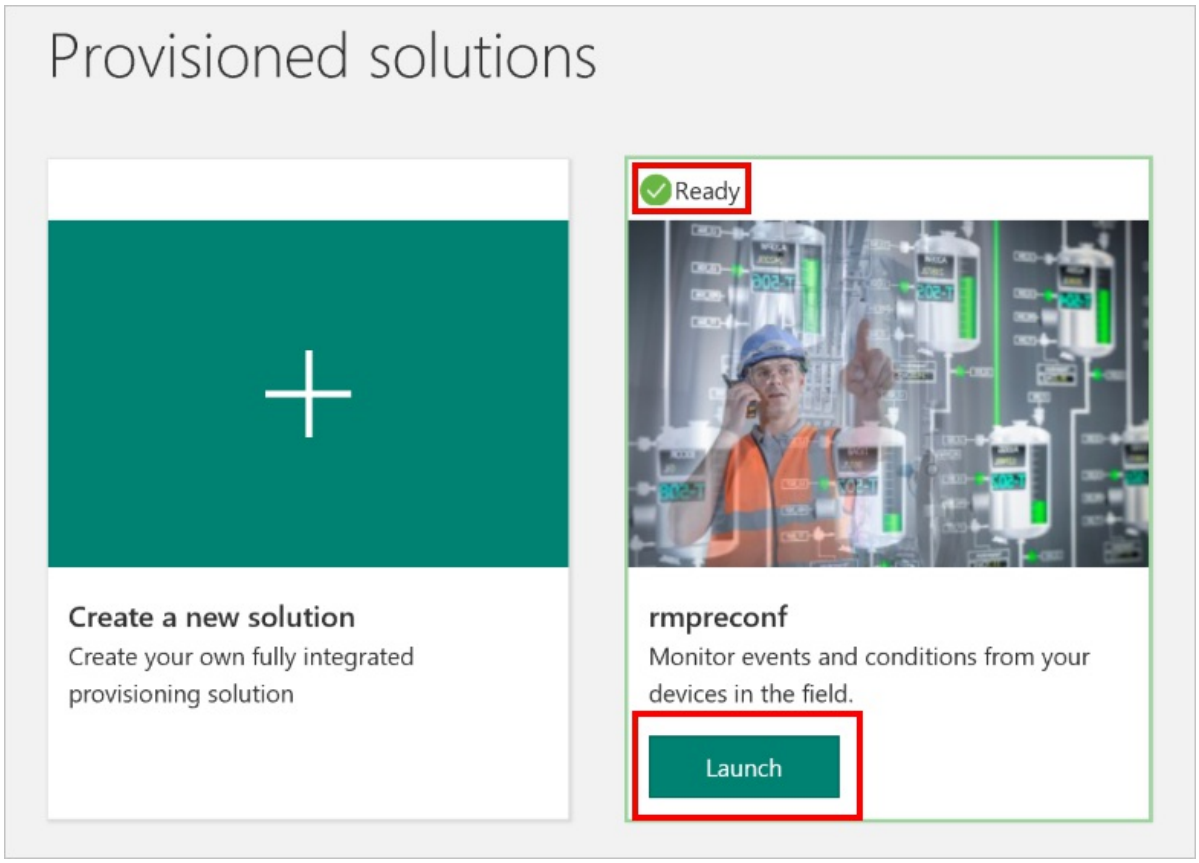
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

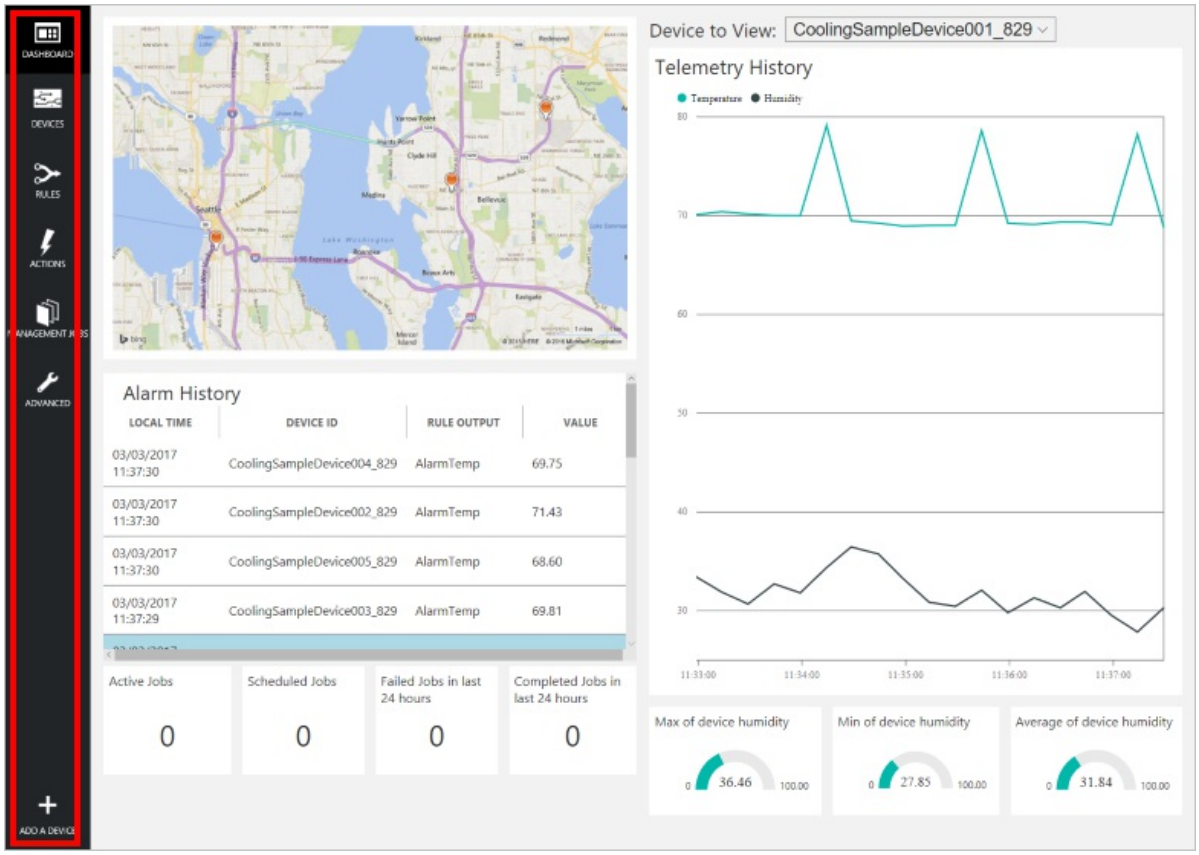
## View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.

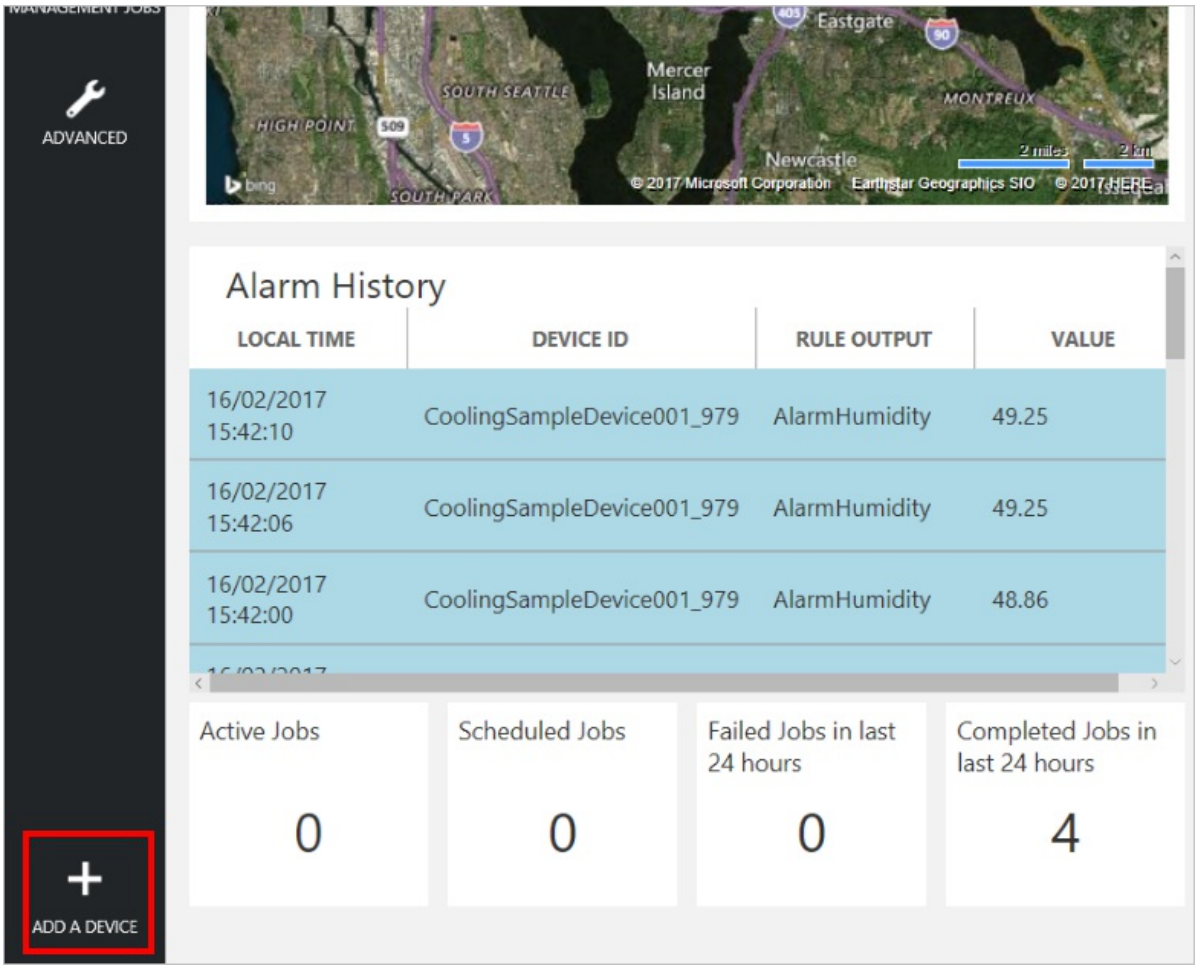



## Add a device

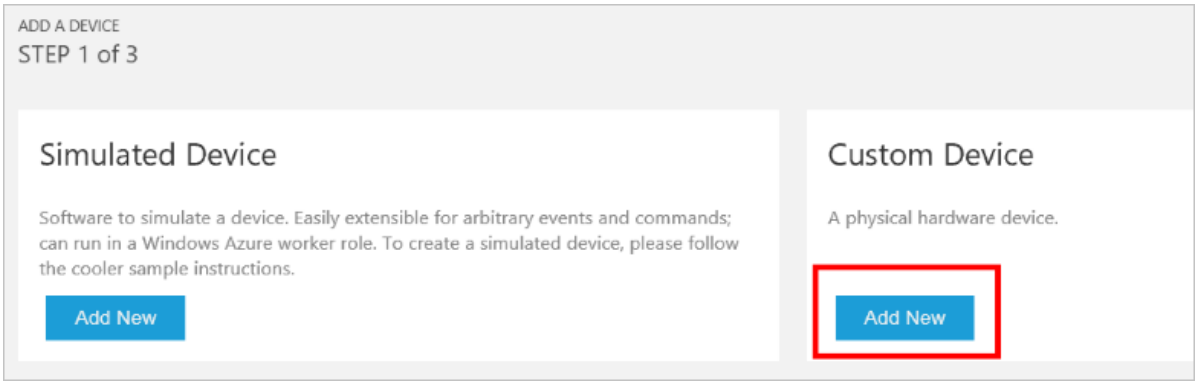
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

If you haven't already done so, add a custom device to your remote monitoring solution. Complete the following steps in the solution dashboard:

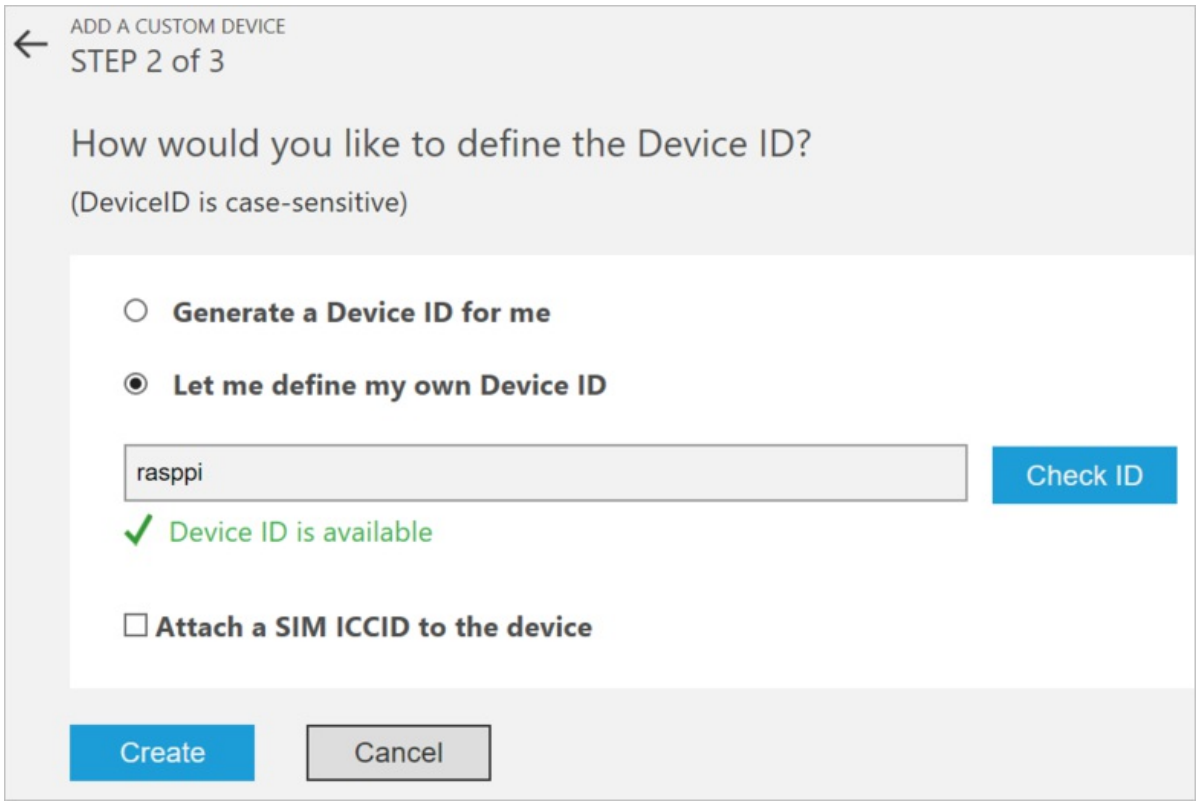
1. In the lower left-hand corner of the dashboard, click **Add a device**.



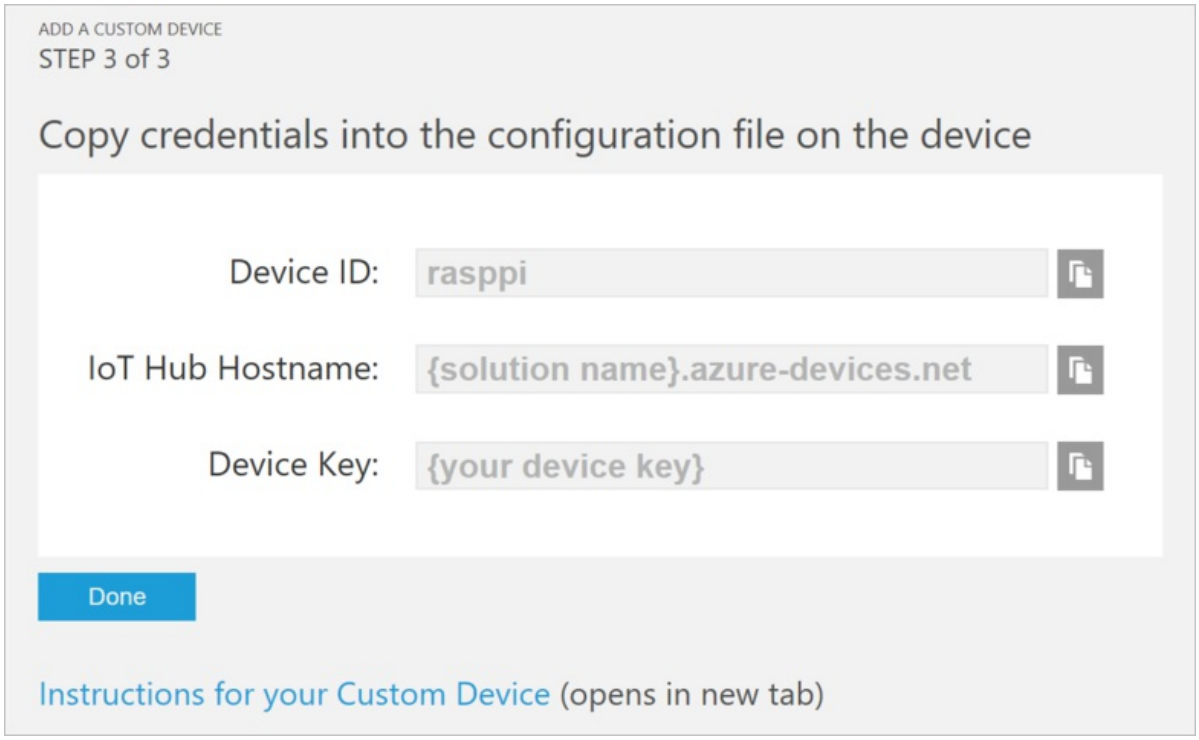

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **rasppi**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Raspberry Pi needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Raspberry Pi

**Install Raspbian**

If this is the first time you are using your Raspberry Pi, you need to install the Raspbian operating system using

NOOBS on the SD card included in the kit. The Raspberry Pi Software Guide describes how to install an operating system on your Raspberry Pi. This tutorial assumes you have installed the Raspbian operating system on your Raspberry Pi.

> **NOTE**
>
> The SD card included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 already has NOOBS installed. You can boot the Raspberry Pi from this card and choose to install the Raspbian OS.

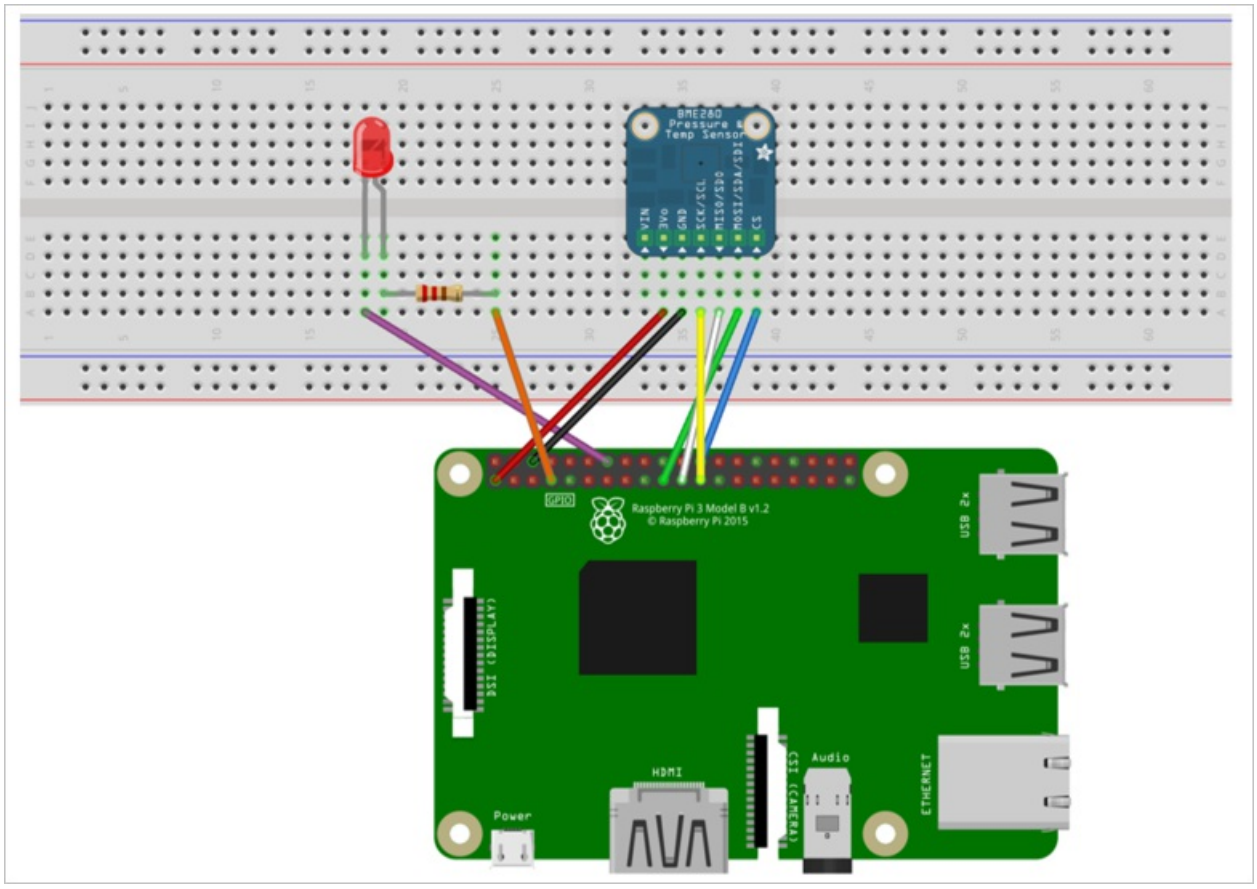
### Set up the hardware

This tutorial uses the BME280 sensor included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to generate telemetry data. It uses an LED to indicate when the Raspberry Pi processes a method invocation from the solution dashboard.

The components on the bread board are:

- Red LED
- 220-Ohm resistor (red, red, brown)
- BME280 sensor

The following diagram shows how to connect your hardware:




The following table summarizes the connections from the Raspberry Pi to the components on the breadboard:

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| GND (Pin 14) | LED -ve pin (18A) | Purple |
| GPCLK0 (Pin 7) | Resistor (25A) | Orange |

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| SPI_CE0 (Pin 24) | CS (39A) | Blue |
| SPI_SCLK (Pin 23) | SCK (36A) | Yellow |
| SPI_MISO (Pin 21) | SDO (37A) | White |
| SPI_MOSI (Pin 19) | SDI (38A) | Green |
| GND (Pin 6) | GND (35A) | Black |
| 3.3 V (Pin 1) | 3Vo (34A) | Red |

To complete the hardware setup, you need to:

- Connect your Raspberry Pi to the power supply included in the kit.
- Connect your Raspberry Pi to your network using the Ethernet cable included in your kit. Alternatively, you can set up Wireless Connectivity for your Raspberry Pi.

You have now completed the hardware setup of your Raspberry Pi.

### Sign in and access the terminal

You have two options to access a terminal environment on your Raspberry Pi:

- If you have a keyboard and monitor connected to your Raspberry Pi, you can use the Raspbian GUI to access a terminal window.
- Access the command line on your Raspberry Pi using SSH from your desktop machine.

#### Use a terminal Window in the GUI

The default credentials for Raspbian are username **pi** and password **raspberry**. In the task bar in the GUI, you can launch the **Terminal** utility using the icon that looks like a monitor.

#### Sign in with SSH

You can use SSH for command-line access to your Raspberry Pi. The article SSH (Secure Shell) describes how to configure SSH on your Raspberry Pi, and how to connect from Windows or Linux & Mac OS.

Sign in with username **pi** and password **raspberry**.

#### Optional: Share a folder on your Raspberry Pi

Optionally, you may want to share a folder on your Raspberry Pi with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Raspberry Pi instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Raspberry Pi. Alternatively, use the built-in SFTP server with an SFTP client on your desktop.

### Enable SPI

Before you can run the sample application, you must enable the Serial Peripheral Interface (SPI) bus on the Raspberry Pi. The Raspberry Pi communicates with the BME280 sensor device over the SPI bus. Use the following command to edit the configuration file:

```
sudo nano /boot/config.txt
```

Find the line:

`#dtparam=spi=on`

- To uncomment the line, delete the `#` at the start.
- Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).
- To enable SPI, reboot the Raspberry Pi. Rebooting disconnects the terminal, you need to sign in again when the Raspberry Pi restarts:

```
sudo reboot
```

# Download and configure the sample

You can now download and configure the remote monitoring client application on your Raspberry Pi.

### Clone the repositories

If you haven't already done so, clone the required repositories by running the following commands in a terminal on your Pi:

```
cd ~
git clone --recursive https://github.com/Azure-Samples/iot-remote-monitoring-c-raspberrypi-getstartedkit.git
git clone --recursive https://github.com/WiringPi/WiringPi.git
```

### Update the device connection string

Open the sample source file in the **nano** editor using the following command:

```
nano ~/iot-remote-monitoring-c-raspberrypi-getstartedkit/basic/remote_monitoring/remote_monitoring.c
```

Locate the following lines:

```
static const char* deviceId = "[Device Id]";
static const char* connectionString = "HostName=[IoTHub Name].azure-devices.net;DeviceId=[Device
Id];SharedAccessKey=[Device Key]";
```

Replace the placeholder values with the device and IoT Hub information you created and saved at the start of this tutorial. Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).

# Build the sample

Install the prerequisite packages for the Microsoft Azure IoT Device SDK for C by running the following commands in a terminal on the Raspberry Pi:

```
sudo apt-get update
sudo apt-get install g++ make cmake git libcurl4-openssl-dev libssl-dev uuid-dev
```

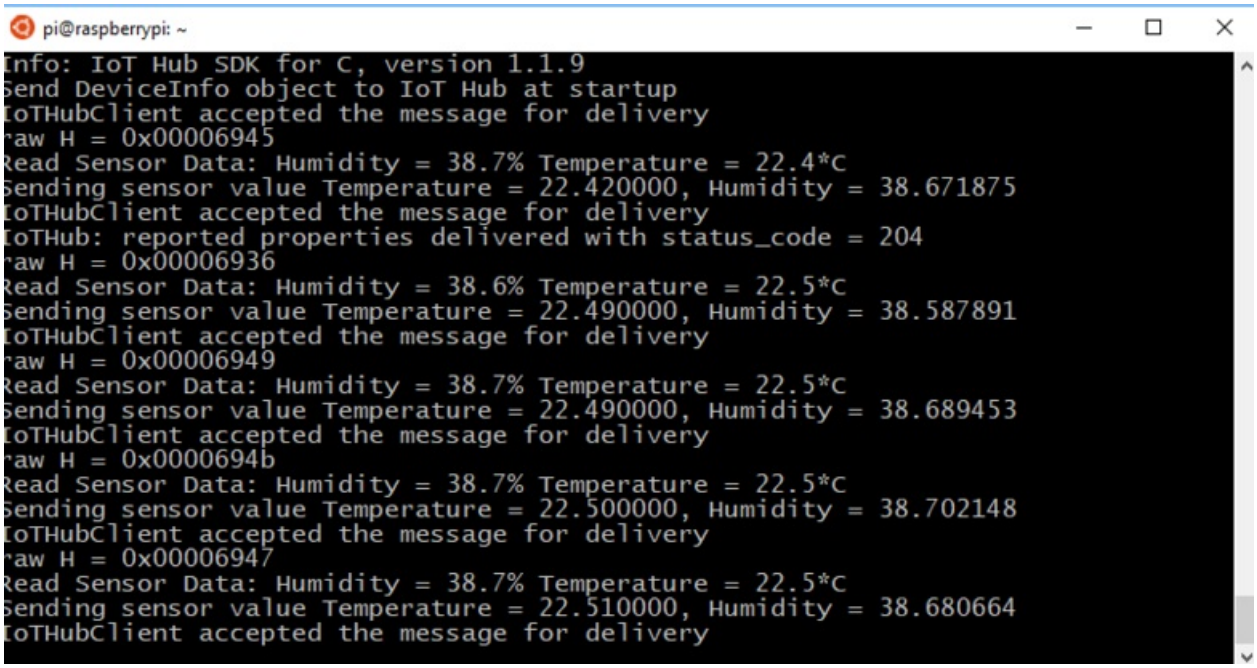
You can now build the updated sample solution on the Raspberry Pi:

```
chmod +x ~/iot-remote-monitoring-c-raspberrypi-getstartedkit/basic/build.sh
~/iot-remote-monitoring-c-raspberrypi-getstartedkit/basic/build.sh
```

You can now run the sample program on the Raspberry Pi. Enter the command:

```
sudo ~/cmake/remote_monitoring/remote_monitoring
```

The following sample output is an example of the output you see at the command prompt on the Raspberry Pi:

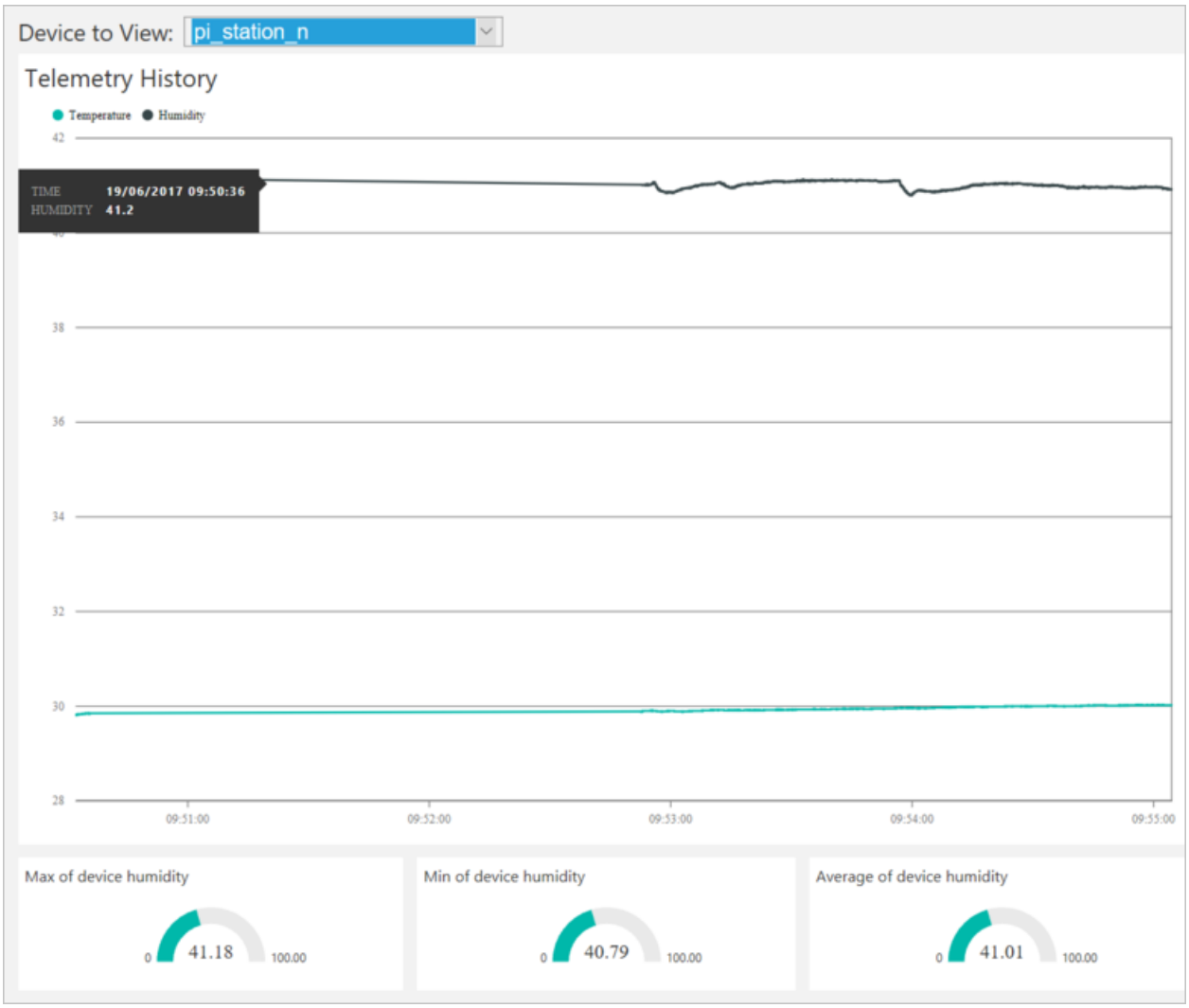



Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The Raspberry Pi is now sending telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send messages to your Raspberry Pi from the solution dashboard.

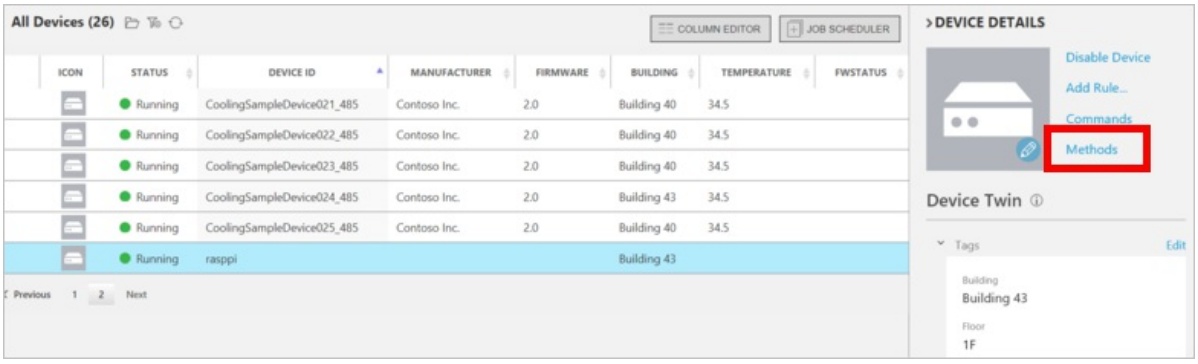
- Navigate to the solution dashboard.
- Select your device in the **Device to View** dropdown.
- The telemetry from the Raspberry Pi displays on the dashboard.



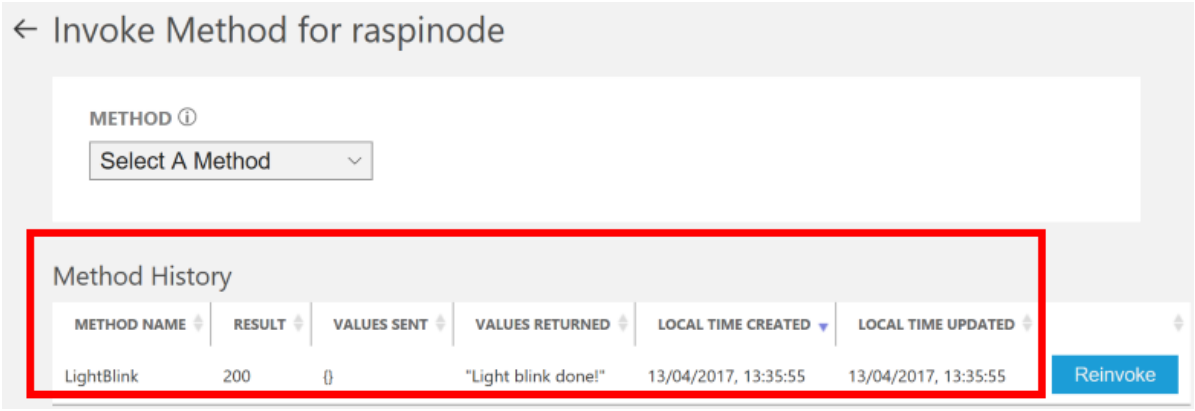

## Act on the device

From the solution dashboard, you can invoke methods on your Raspberry Pi. When the Raspberry Pi connects to the remote monitoring solution, it sends information about the methods it supports.

- In the solution dashboard, click **Devices** to visit the **Devices** page. Select your Raspberry Pi in the **Device List**. Then choose **Methods**:




- On the **Invoke Method** page, choose **LightBlink** in the **Method** dropdown.
- Choose **InvokeMethod**. The LED connected to the Raspberry Pi flashes several times. The app on the Raspberry Pi sends an acknowledgment back to the solution dashboard:




- You can switch the LED on and off using the **ChangeLightStatus** method with a **LightStatusValue** set to **1** for on or **0** for off.

> **WARNING**
> If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Raspberry Pi 3 to the remote monitoring solution and enable remote firmware updates using C

7/25/2017 • 10 min to read • Edit Online

This tutorial shows you how to use the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to:

- Develop a temperature and humidity reader that can communicate with the cloud.
- Enable and perform a remote firmware update to update the client application on the Raspberry Pi.

The tutorial uses:

- Raspbian OS, the C programming language, and the Microsoft Azure IoT SDK for C to implement a sample device.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your device and sensors to communicate with your computer and the remote monitoring solution.
- Update the sample device code to connect to the remote monitoring solution, and send telemetry that you can view on the solution dashboard.
- Use the sample device code to update the client application.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Raspberry Pi.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility. For more information, see SSH Using Linux or Mac OS.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Raspberry Pi.

Microsoft IoT Starter Kit for Raspberry Pi 3 or equivalent components. This tutorial uses the following items from the kit:

- Raspberry Pi 3
- MicroSD Card (with NOOBS)
- A USB Mini cable
- An Ethernet cable
- BME280 sensor
- Breadboard
- Jumper wires
- Resistors
- LEDs

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

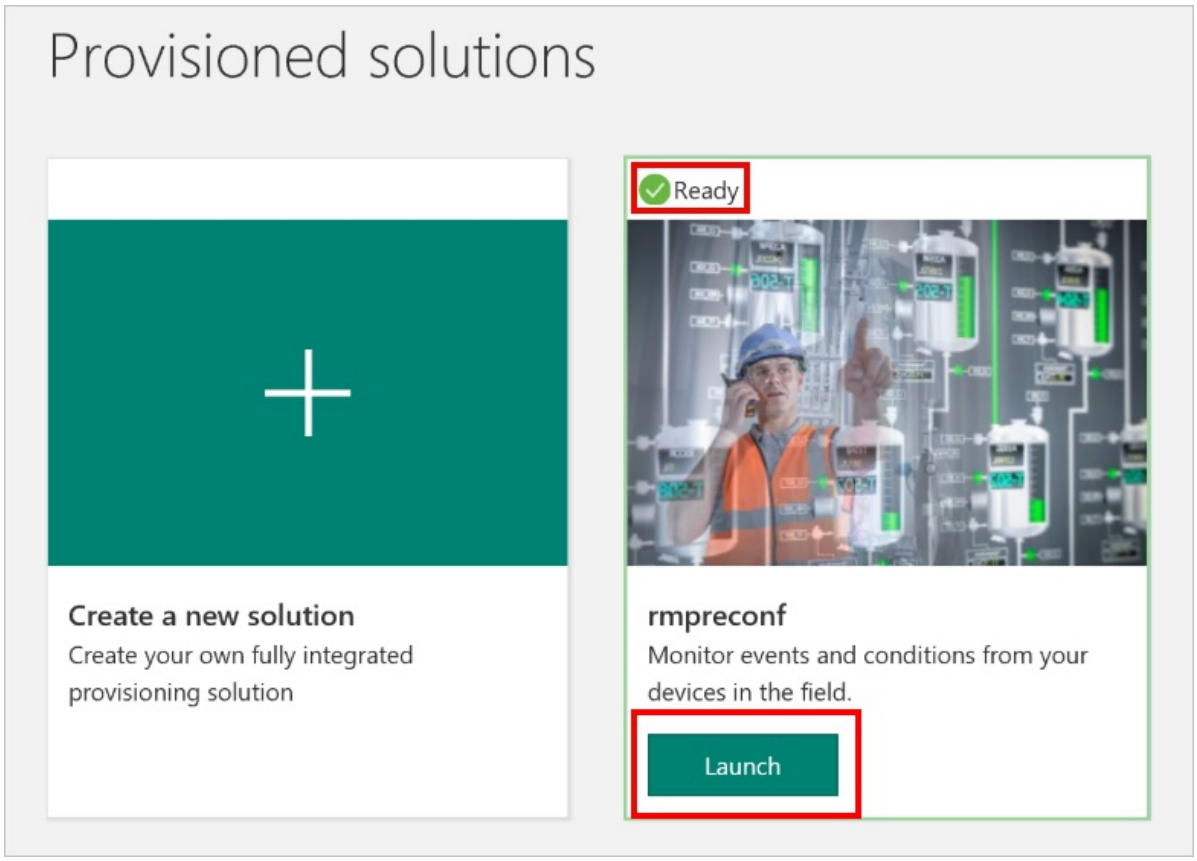
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

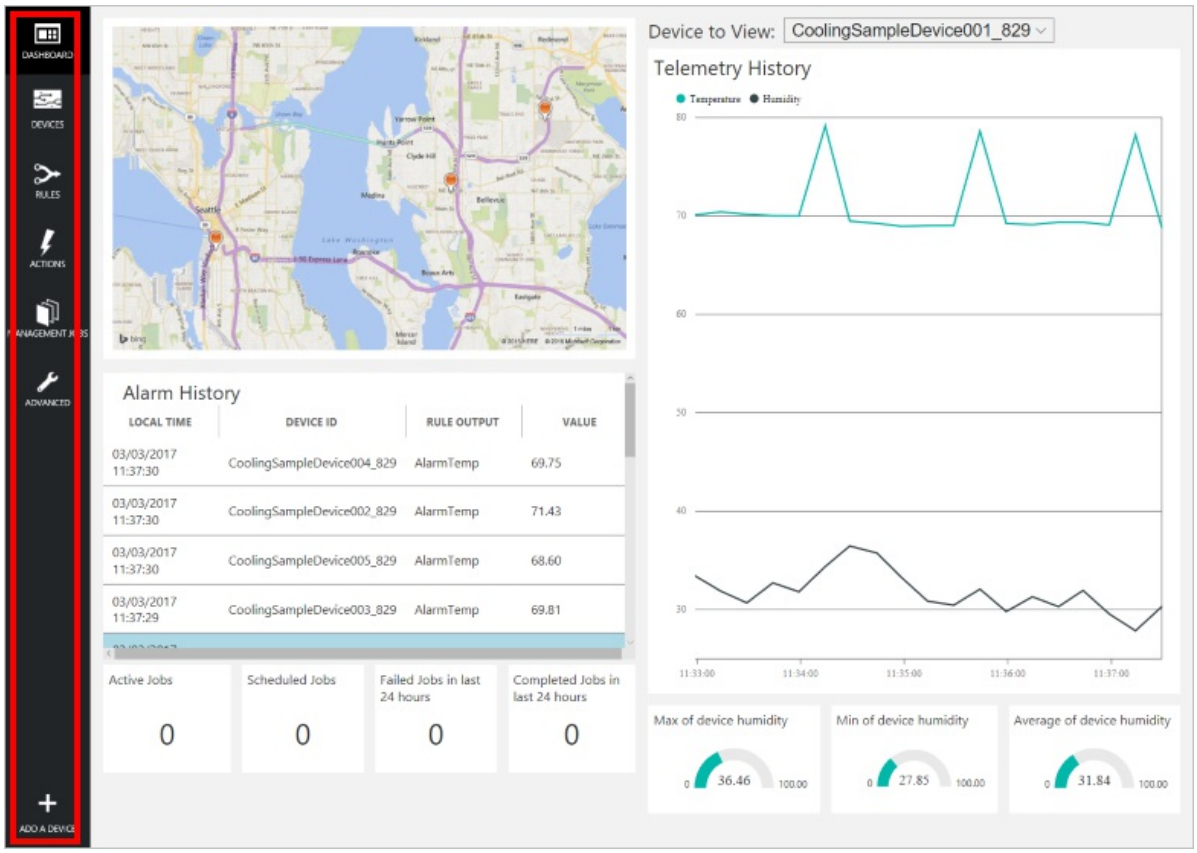
## View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.

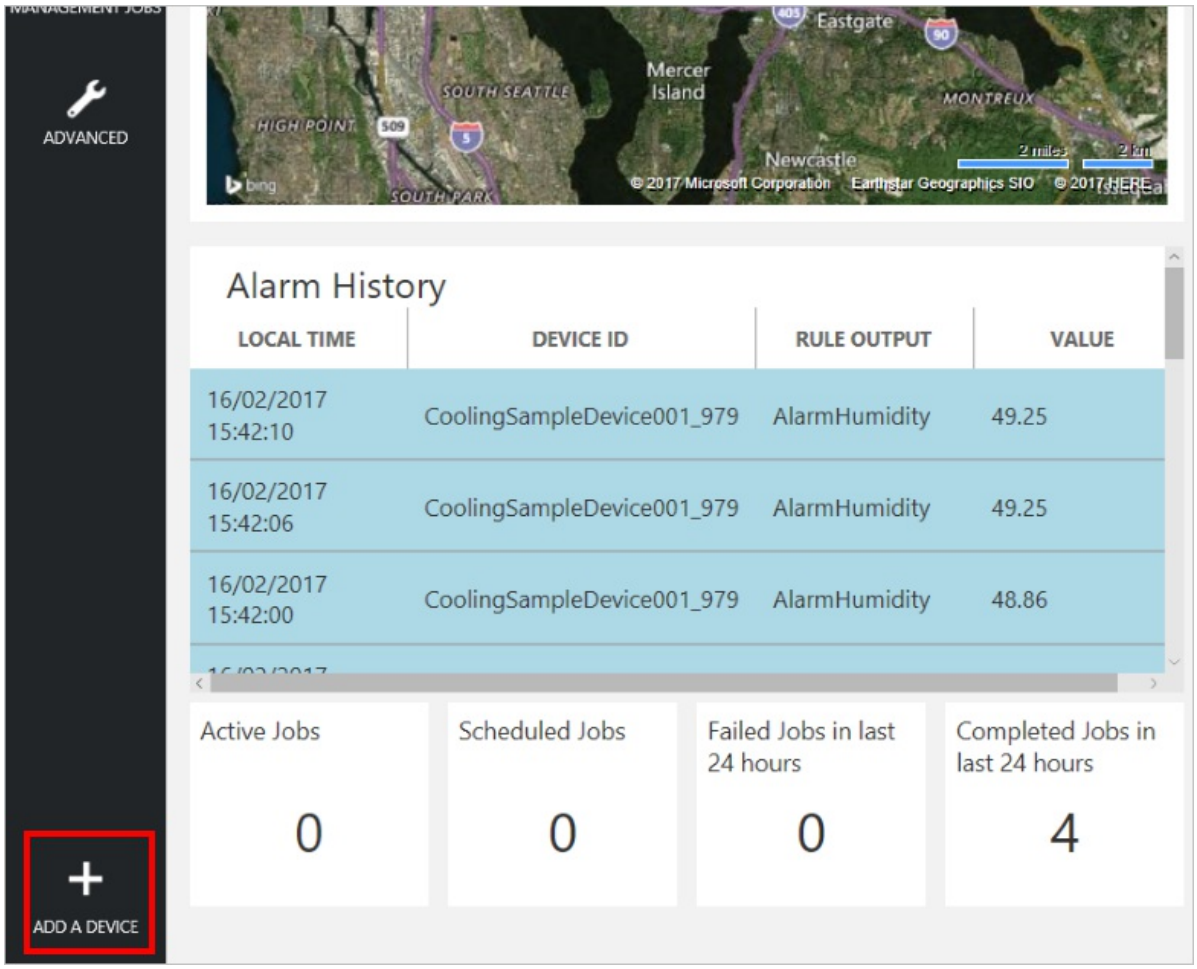



## Add a device

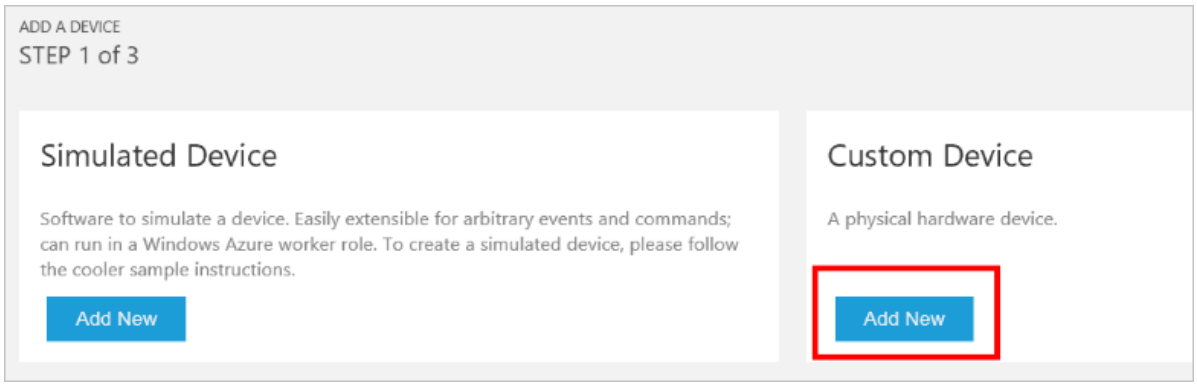
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

If you haven't already done so, add a custom device to your remote monitoring solution. Complete the following steps in the solution dashboard:

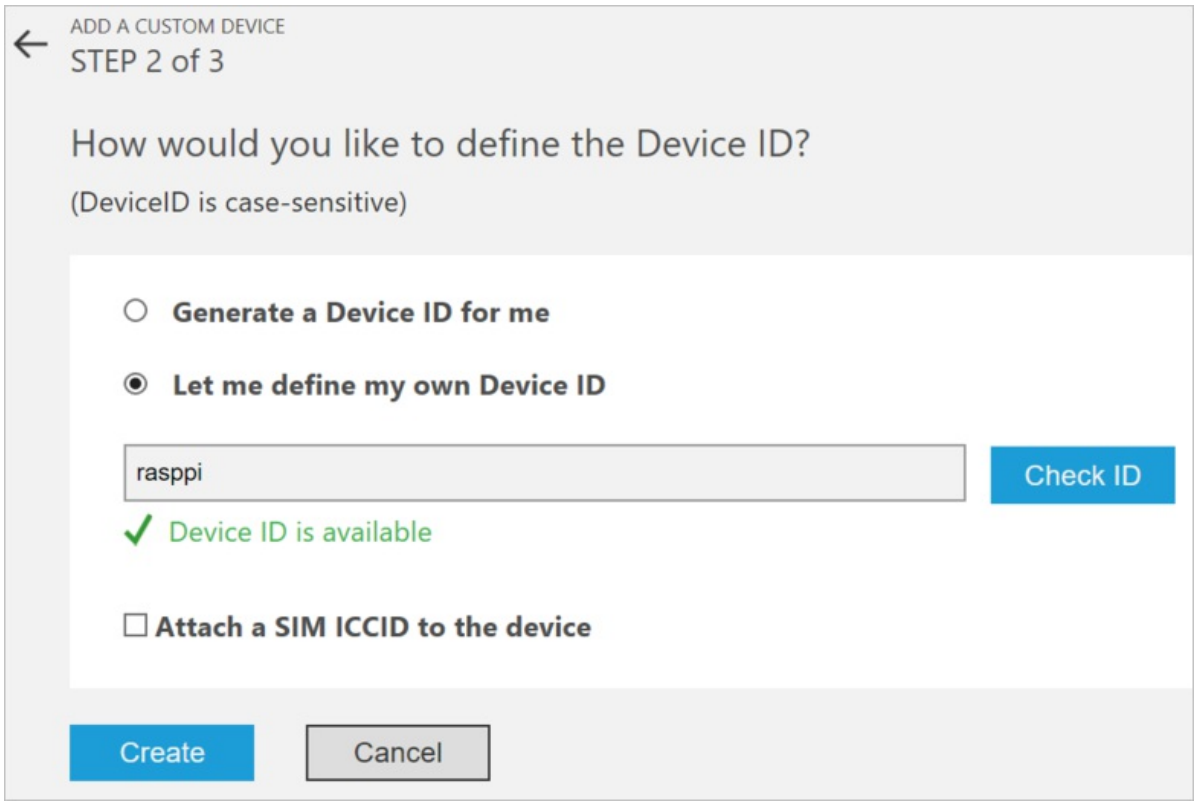
1. In the lower left-hand corner of the dashboard, click **Add a device**.



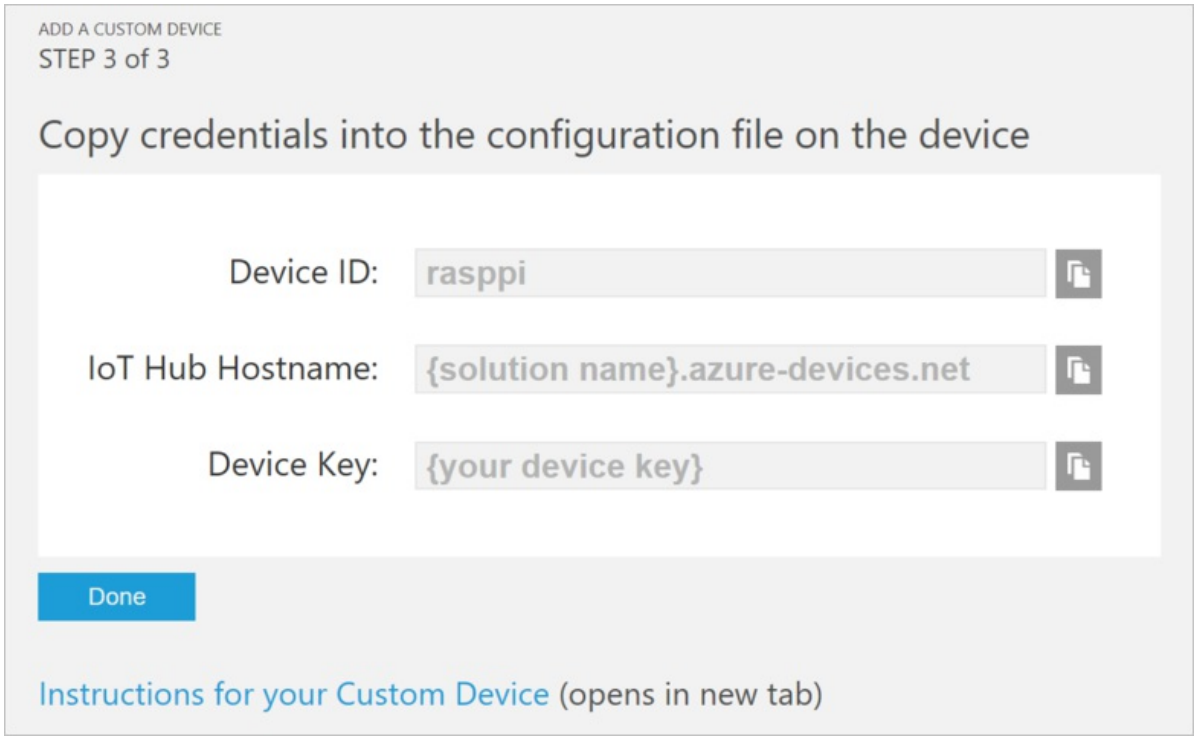

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **rasppi**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Raspberry Pi needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Raspberry Pi

**Install Raspbian**

If this is the first time you are using your Raspberry Pi, you need to install the Raspbian operating system using

NOOBS on the SD card included in the kit. The Raspberry Pi Software Guide describes how to install an operating system on your Raspberry Pi. This tutorial assumes you have installed the Raspbian operating system on your Raspberry Pi.

> **NOTE**
>
> The SD card included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 already has NOOBS installed. You can boot the Raspberry Pi from this card and choose to install the Raspbian OS.

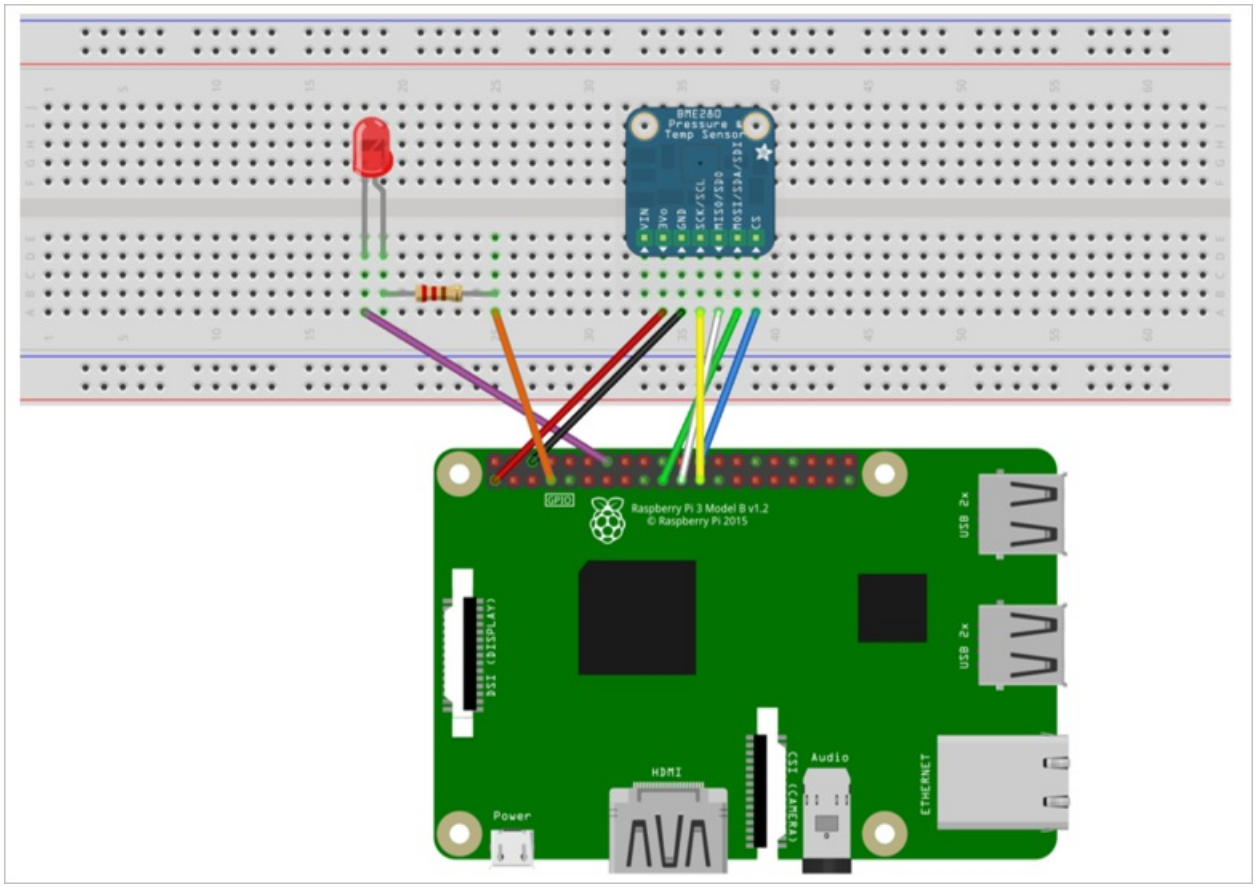
### Set up the hardware

This tutorial uses the BME280 sensor included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to generate telemetry data. It uses an LED to indicate when the Raspberry Pi processes a method invocation from the solution dashboard.

The components on the bread board are:

- Red LED
- 220-Ohm resistor (red, red, brown)
- BME280 sensor

The following diagram shows how to connect your hardware:




The following table summarizes the connections from the Raspberry Pi to the components on the breadboard:

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| GND (Pin 14) | LED -ve pin (18A) | Purple |
| GPCLK0 (Pin 7) | Resistor (25A) | Orange |

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| SPI_CE0 (Pin 24) | CS (39A) | Blue |
| SPI_SCLK (Pin 23) | SCK (36A) | Yellow |
| SPI_MISO (Pin 21) | SDO (37A) | White |
| SPI_MOSI (Pin 19) | SDI (38A) | Green |
| GND (Pin 6) | GND (35A) | Black |
| 3.3 V (Pin 1) | 3Vo (34A) | Red |

To complete the hardware setup, you need to:

- Connect your Raspberry Pi to the power supply included in the kit.
- Connect your Raspberry Pi to your network using the Ethernet cable included in your kit. Alternatively, you can set up Wireless Connectivity for your Raspberry Pi.

You have now completed the hardware setup of your Raspberry Pi.

### Sign in and access the terminal

You have two options to access a terminal environment on your Raspberry Pi:

- If you have a keyboard and monitor connected to your Raspberry Pi, you can use the Raspbian GUI to access a terminal window.
- Access the command line on your Raspberry Pi using SSH from your desktop machine.

#### Use a terminal Window in the GUI

The default credentials for Raspbian are username **pi** and password **raspberry**. In the task bar in the GUI, you can launch the **Terminal** utility using the icon that looks like a monitor.

#### Sign in with SSH

You can use SSH for command-line access to your Raspberry Pi. The article SSH (Secure Shell) describes how to configure SSH on your Raspberry Pi, and how to connect from Windows or Linux & Mac OS.

Sign in with username **pi** and password **raspberry**.

#### Optional: Share a folder on your Raspberry Pi

Optionally, you may want to share a folder on your Raspberry Pi with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Raspberry Pi instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Raspberry Pi. Alternatively, use the built-in SFTP server with an SFTP client on your desktop.

### Enable SPI

Before you can run the sample application, you must enable the Serial Peripheral Interface (SPI) bus on the Raspberry Pi. The Raspberry Pi communicates with the BME280 sensor device over the SPI bus. Use the following command to edit the configuration file:

```
sudo nano /boot/config.txt
```

Find the line:

`#dtparam=spi=on`

- To uncomment the line, delete the `#` at the start.
- Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).
- To enable SPI, reboot the Raspberry Pi. Rebooting disconnects the terminal, you need to sign in again when the Raspberry Pi restarts:

```
sudo reboot
```

# Download and configure the sample

You can now download and configure the remote monitoring client application on your Raspberry Pi.

### Clone the repositories

If you haven't done so already, clone the required repositories by running the following commands on your Pi:

```
cd ~
git clone --recursive https://github.com/Azure-Samples/iot-remote-monitoring-c-raspberrypi-getstartedkit.git
```

### Update the device connection string

Open the sample configuration file in the **nano** editor using the following command:

```
nano ~/iot-remote-monitoring-c-raspberrypi-getstartedkit/advanced/config/deviceinfo
```

Replace the placeholder values with the device ID and IoT Hub information you created and saved at the start of this tutorial.

When you are done, the contents of the deviceinfo file should look like the following example:

```
yourdeviceid
HostName=youriothubname.azure-devices.net;DeviceId=yourdeviceid;SharedAccessKey=yourdevicekey
```

Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).

# Build the sample

If you have not already done so, install the prerequisite packages for the Microsoft Azure IoT Device SDK for C by running the following commands in a terminal on the Raspberry Pi:

```
sudo apt-get update
sudo apt-get install g++ make cmake git libcurl4-openssl-dev libssl-dev uuid-dev
```

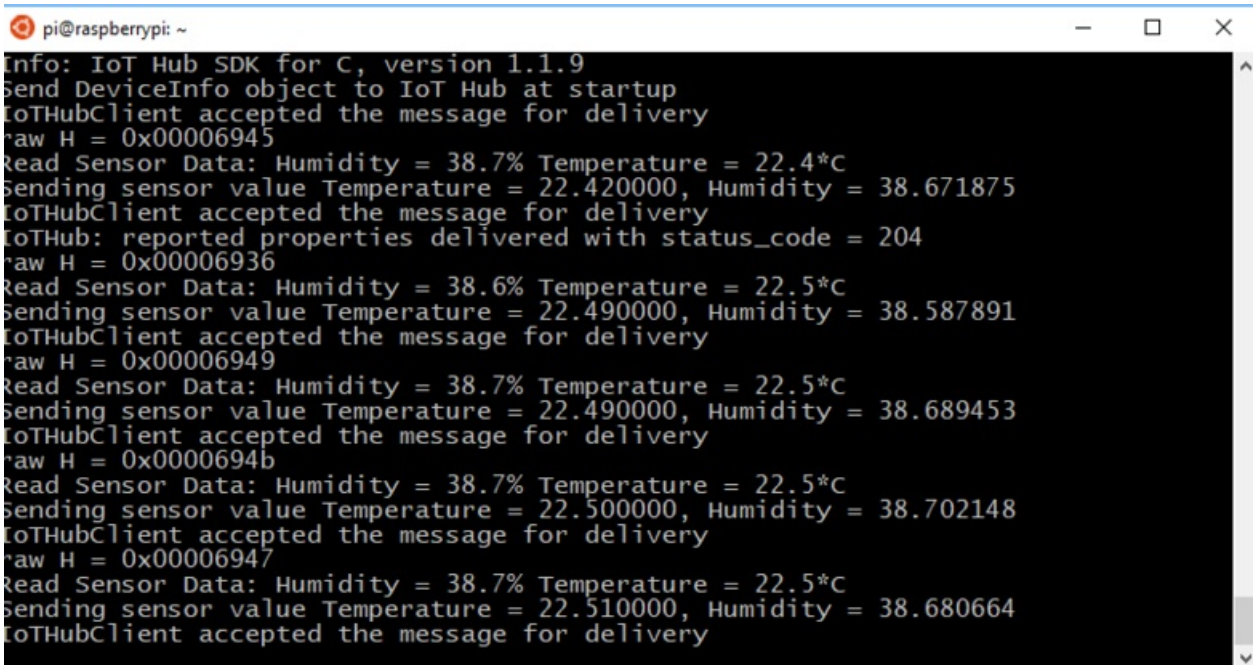
You can now build the sample solution on the Raspberry Pi:

```
chmod +x ~/iot-remote-monitoring-c-raspberrypi-getstartedkit/advanced/1.0/build.sh
~/iot-remote-monitoring-c-raspberrypi-getstartedkit/advanced/1.0/build.sh
```

You can now run the sample program on the Raspberry Pi. Enter the command:

```
sudo ~/cmake/remote_monitoring/remote_monitoring
```

The following sample output is an example of the output you see at the command prompt on the Raspberry Pi:

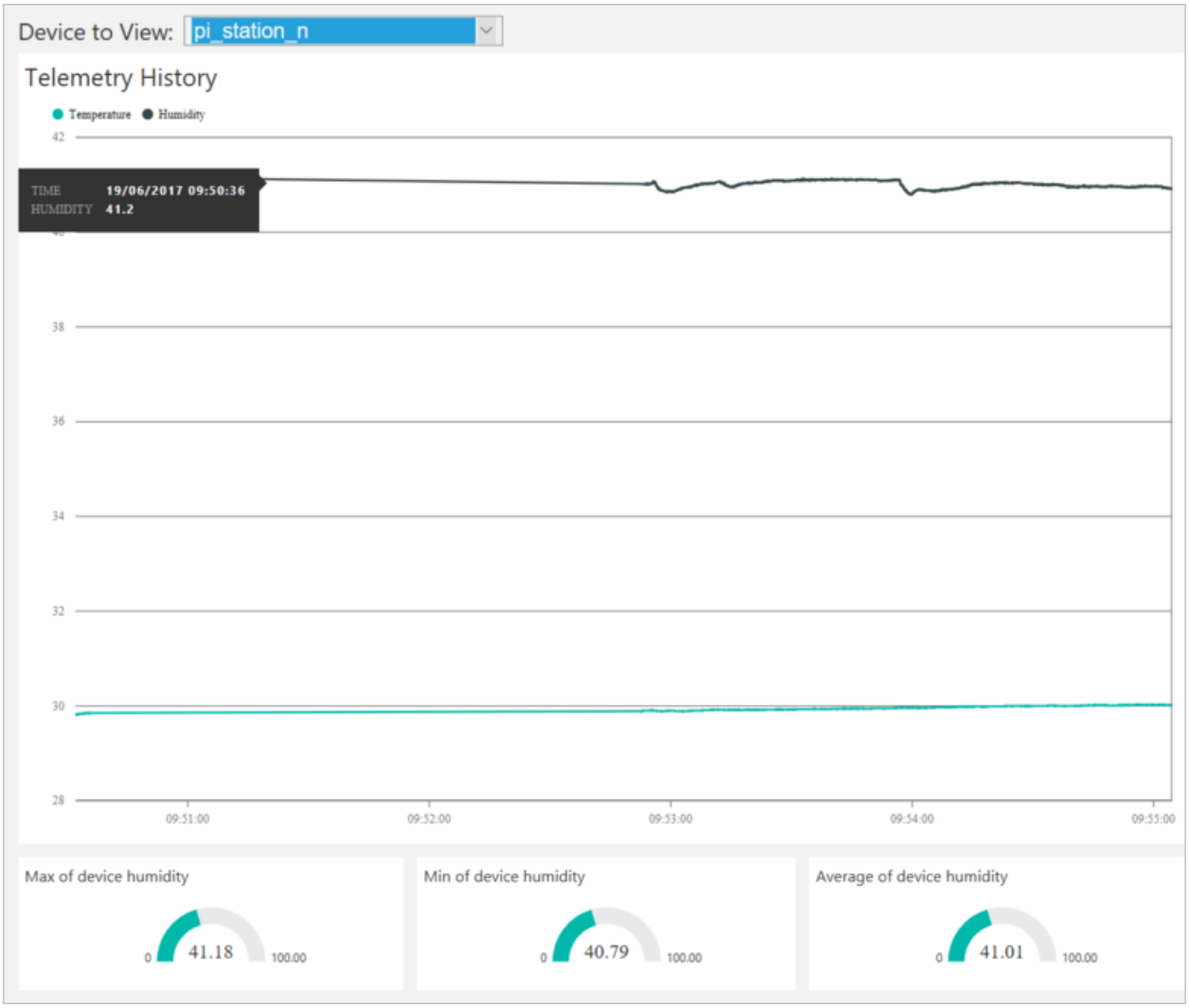



Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The Raspberry Pi is now sending telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send messages to your Raspberry Pi from the solution dashboard.

- Navigate to the solution dashboard.
- Select your device in the **Device to View** dropdown.
- The telemetry from the Raspberry Pi displays on the dashboard.



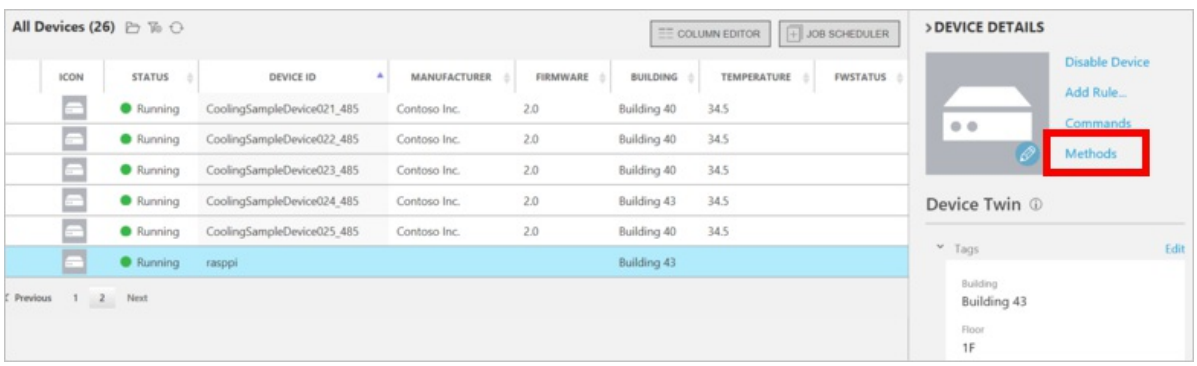

## Initiate the firmware update

The firmware update process downloads and installs an updated version of the device client application on the Raspberry Pi. For more information about the firmware update process, see the description of the firmware update pattern in Overview of device management with IoT Hub.

You initiate the firmware update process by invoking a method on the device. This method is asynchronous, and returns as soon as the update process begins. The device uses reported properties to notify the solution about the progress of the update.

You invoke methods on your Raspberry Pi from the solution dashboard. When the Raspberry Pi first connects to the remote monitoring solution, it sends information about the methods it supports.

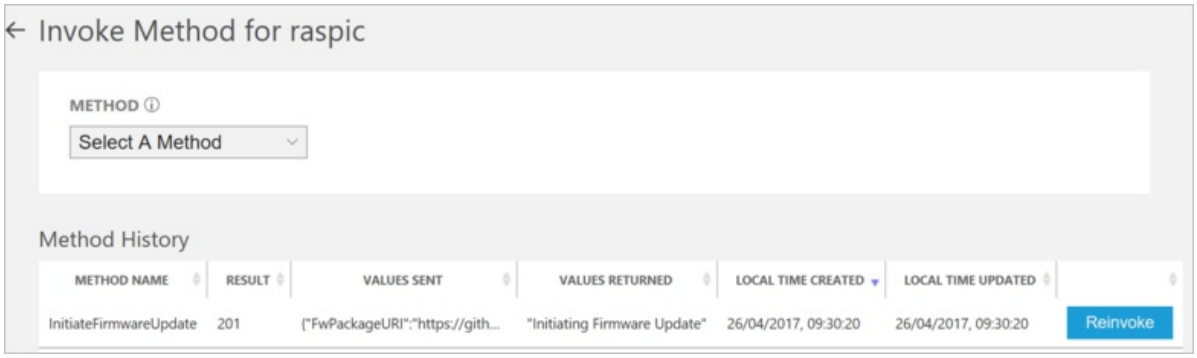
1. In the solution dashboard, click **Devices** to visit the **Devices** page. Select your Raspberry Pi in the **Device List**. Then choose **Methods**:




2. On the **Invoke Method** page, choose **InitiateFirmwareUpdate** in the **Method** dropdown.

3. In the **FWPackageURI** field, enter **https://github.com/Azure-Samples/iot-remote-monitoring-c-raspberrypi-getstartedkit/raw/master/advanced/2.0/package/remote_monitoring.zip**. This archive file contains the implementation of version 2.0 of the firmware.

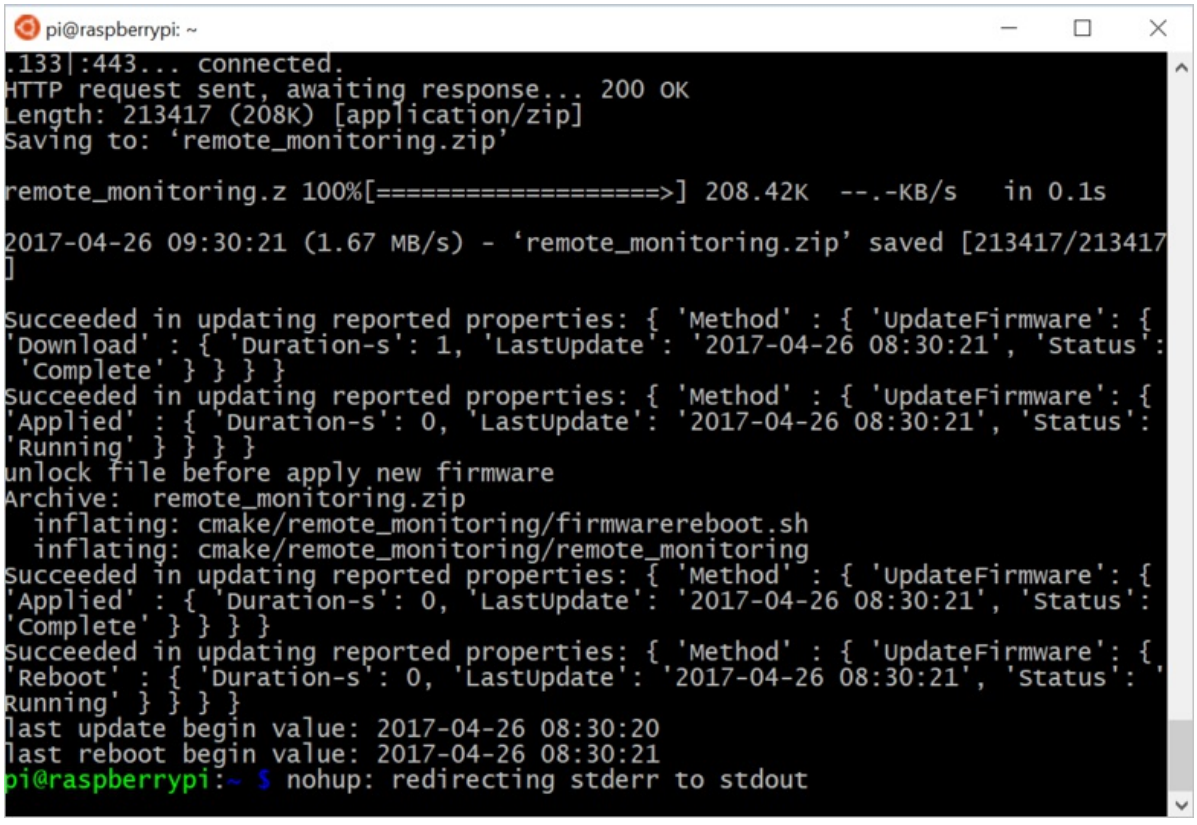
4. Choose **InvokeMethod**. The app on the Raspberry Pi sends an acknowledgment back to the solution dashboard. It then starts the firmware update process by downloading the new version of the firmware:




## Observe the firmware update process

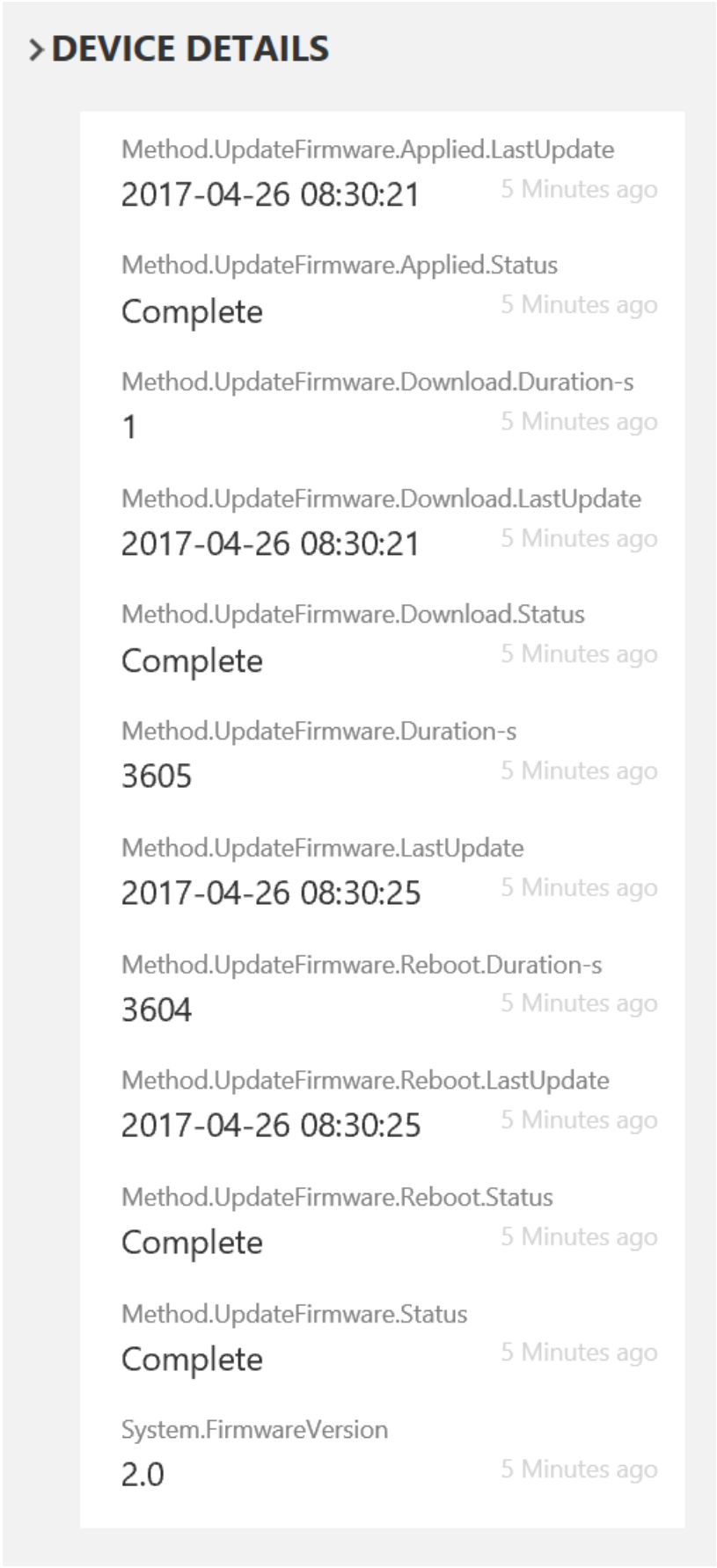
You can observe the firmware update process as it runs on the device and by viewing the reported properties in the solution dashboard:

1. You can view the progress in of the update process on the Raspberry Pi:




> **NOTE**
> The remote monitoring app restarts silently when the update completes. Use the command `ps -ef` to verify it is running. If you want to terminate the process, use the `kill` command with the process id.

2. You can view the status of the firmware update, as reported by the device, in the solution portal. The following screenshot shows the status and duration of each stage of the update process, and the new firmware version:




If you navigate back to the dashboard, you can verify the device is still sending telemetry following the firmware update.

**WARNING**

If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Raspberry Pi 3 to the remote monitoring solution and send simulated telemetry using Node.js

7/25/2017 • 8 min to read • Edit Online

This tutorial shows you how to use the Raspberry Pi 3 to simulate temperature and humidity data to send to the cloud. The tutorial uses:

- Raspbian OS, the Node.js programming language, and the Microsoft Azure IoT SDK for Node.js to implement a sample device.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your device to communicate with your computer and the remote monitoring solution.
- Update the sample device code to connect to the remote monitoring solution, and send simulated telemetry that you can view on the solution dashboard.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Raspberry Pi.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility. For more information, see SSH Using Linux or Mac OS.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Raspberry Pi.

Microsoft IoT Starter Kit for Raspberry Pi 3 or equivalent components. This tutorial uses the following items from the kit:

- Raspberry Pi 3
- MicroSD Card (with NOOBS)
- A USB Mini cable
- An Ethernet cable

# Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click + to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

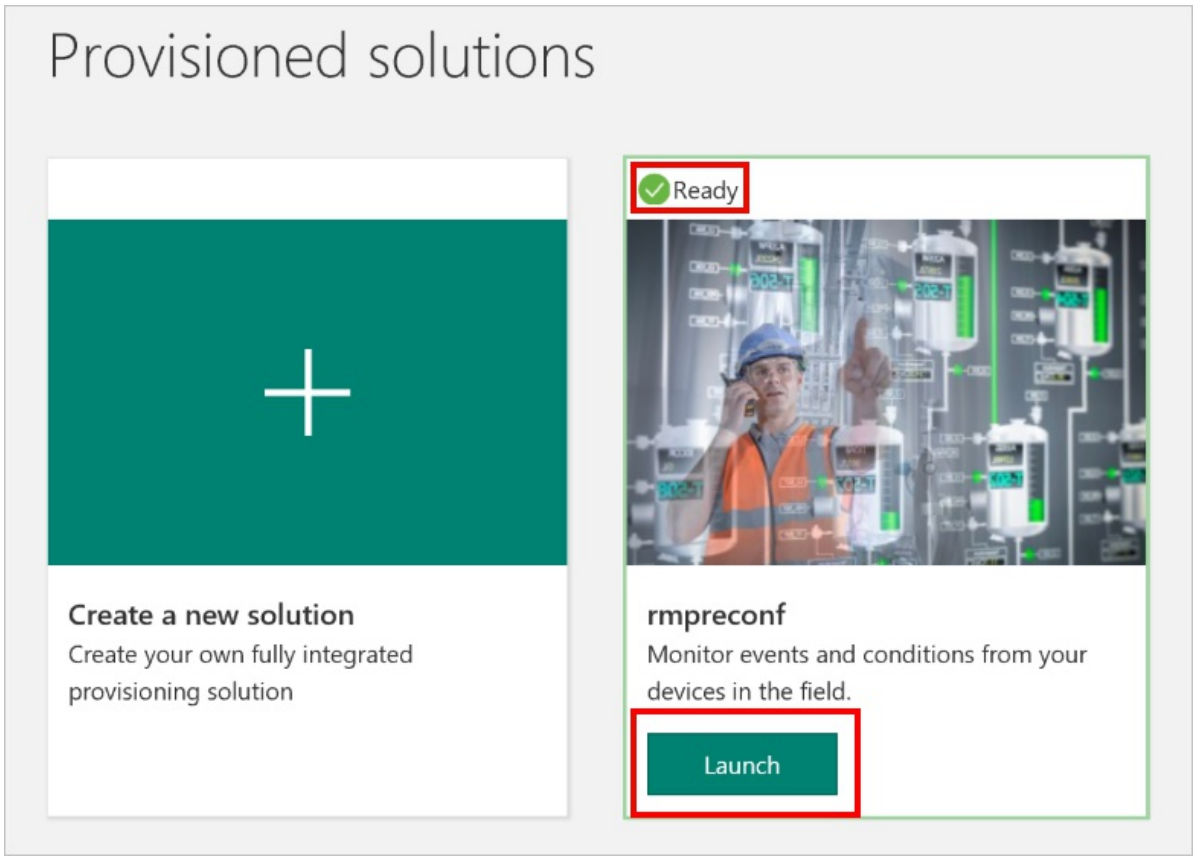
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

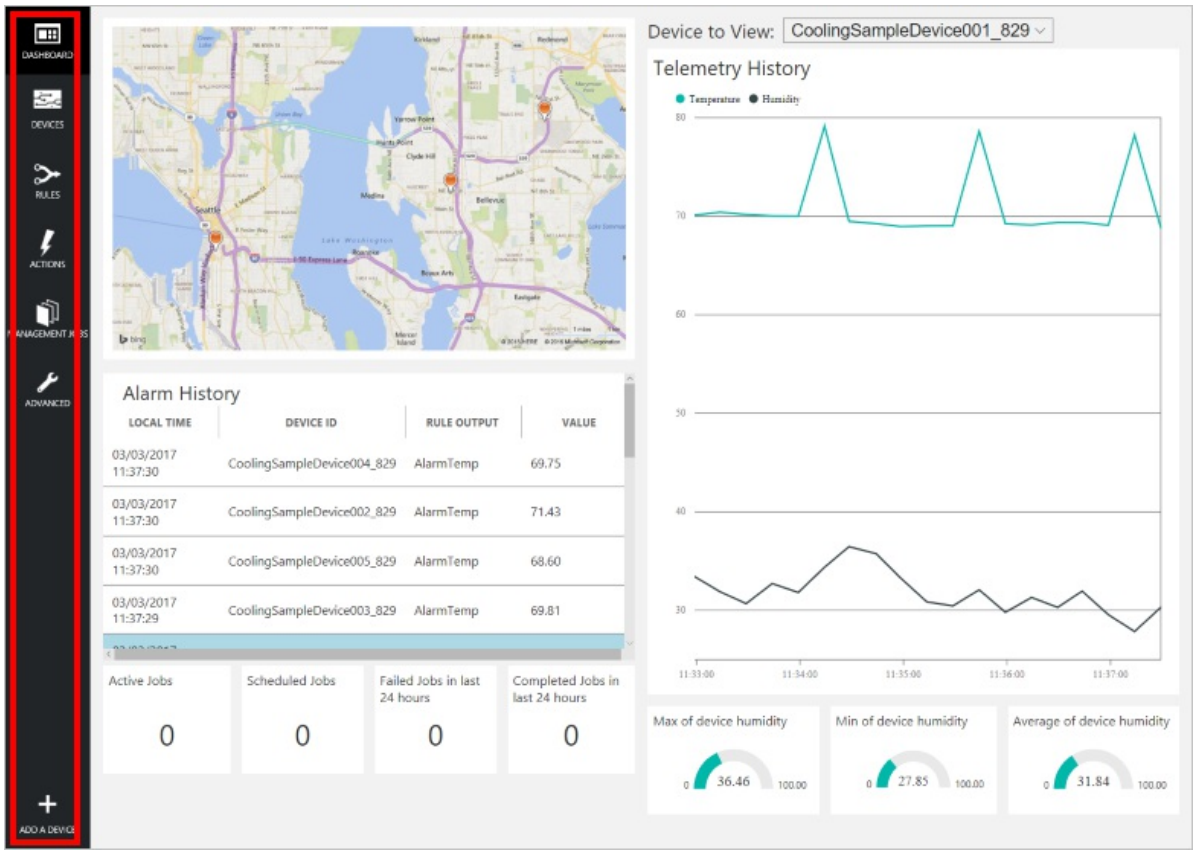
# View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.

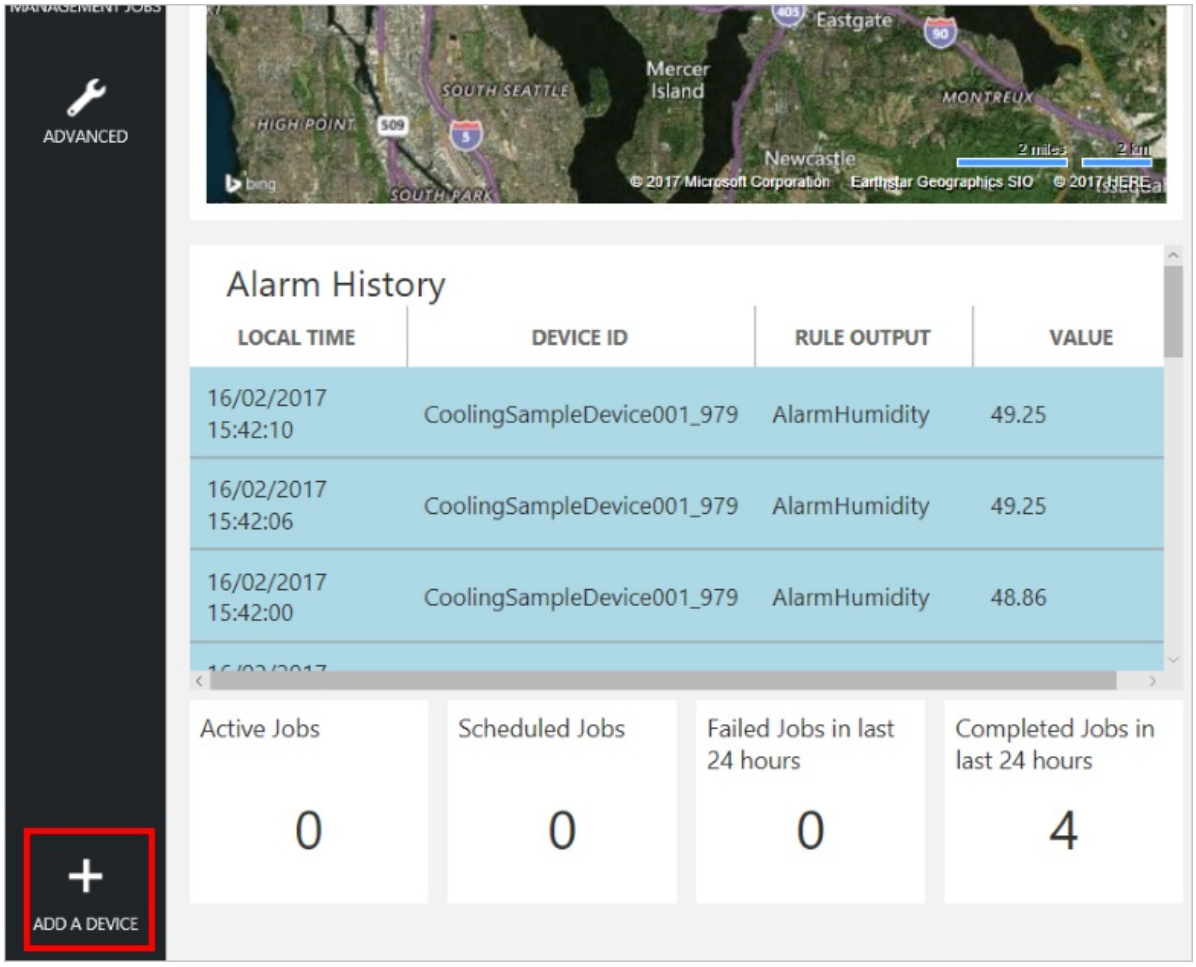



## Add a device

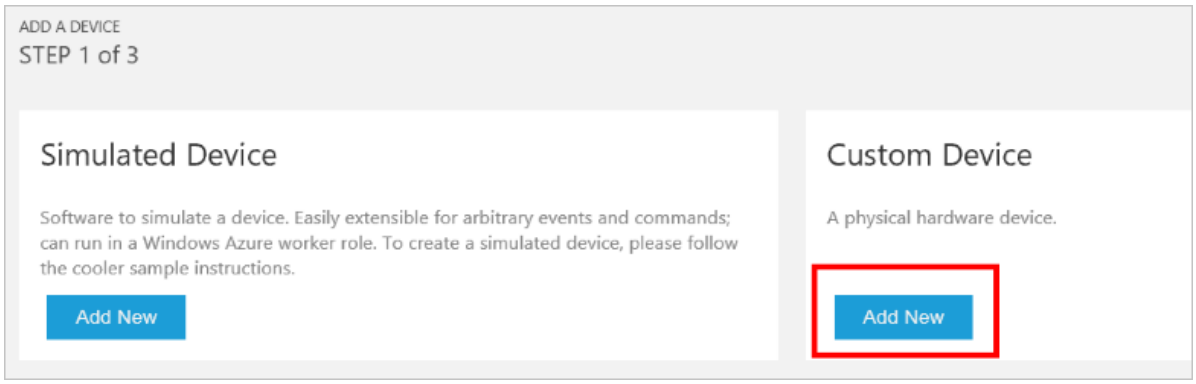
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

If you haven't already done so, add a custom device to your remote monitoring solution. Complete the following steps in the solution dashboard:

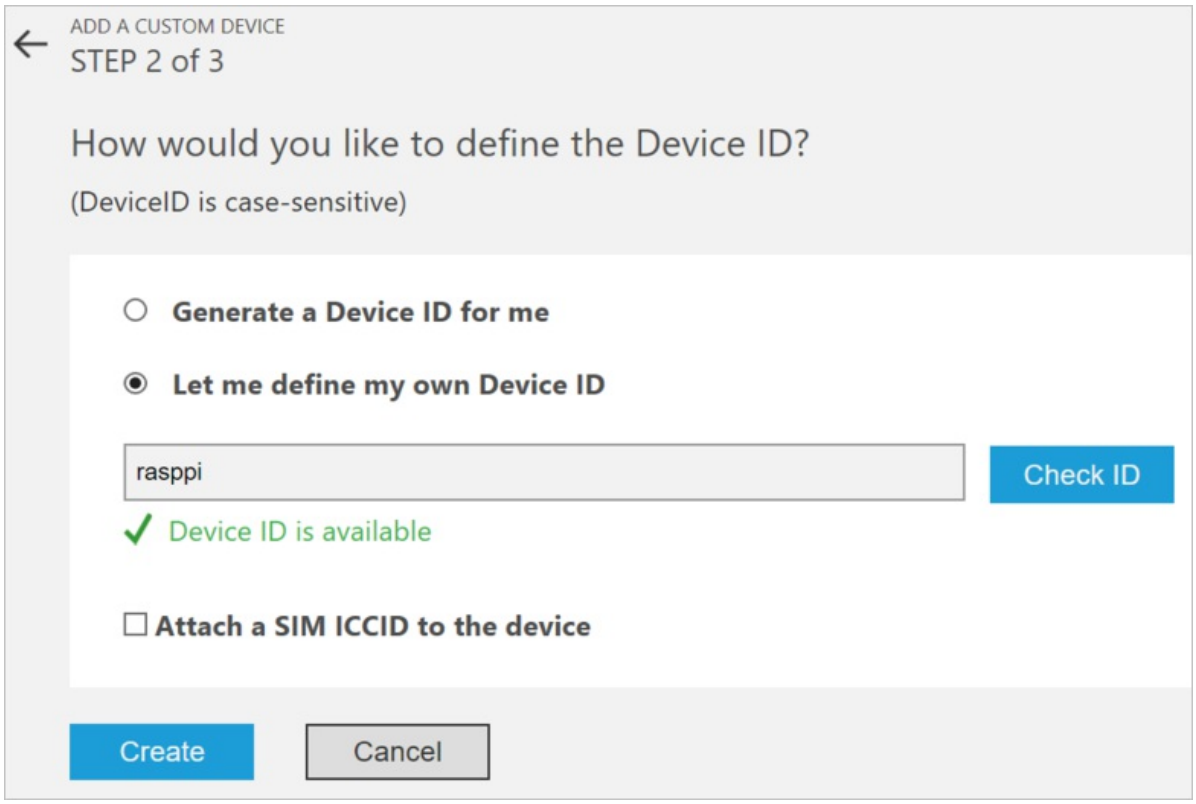
1. In the lower left-hand corner of the dashboard, click **Add a device**.



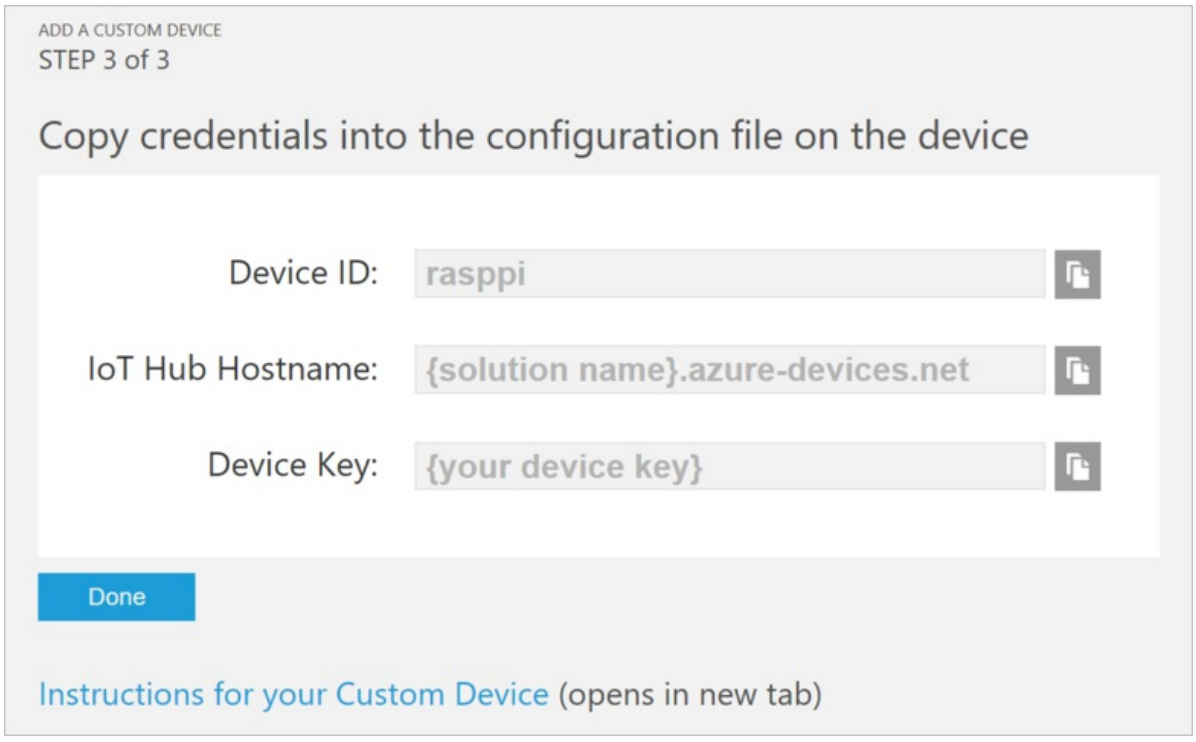

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **raspi**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Raspberry Pi needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Raspberry Pi

**Install Raspbian**

If this is the first time you are using your Raspberry Pi, you need to install the Raspbian operating system using NOOBS on the SD card included in the kit. The Raspberry Pi Software Guide describes how to install an operating

system on your Raspberry Pi. This tutorial assumes you have installed the Raspbian operating system on your Raspberry Pi.

> **NOTE**
> The SD card included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 already has NOOBS installed. You can boot the Raspberry Pi from this card and choose to install the Raspbian OS.

To complete the hardware setup, you need to:

- Connect your Raspberry Pi to the power supply included in the kit.
- Connect your Raspberry Pi to your network using the Ethernet cable included in your kit. Alternatively, you can set up Wireless Connectivity for your Raspberry Pi.

You have now completed the hardware setup of your Raspberry Pi.

### Sign in and access the terminal

You have two options to access a terminal environment on your Raspberry Pi:

- If you have a keyboard and monitor connected to your Raspberry Pi, you can use the Raspbian GUI to access a terminal window.
- Access the command line on your Raspberry Pi using SSH from your desktop machine.

#### Use a terminal Window in the GUI

The default credentials for Raspbian are username **pi** and password **raspberry**. In the task bar in the GUI, you can launch the **Terminal** utility using the icon that looks like a monitor.

#### Sign in with SSH

You can use SSH for command-line access to your Raspberry Pi. The article SSH (Secure Shell) describes how to configure SSH on your Raspberry Pi, and how to connect from Windows or Linux & Mac OS.

Sign in with username **pi** and password **raspberry**.

#### Optional: Share a folder on your Raspberry Pi

Optionally, you may want to share a folder on your Raspberry Pi with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Raspberry Pi instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Raspberry Pi. Alternatively, use the built-in SFTP server with an SFTP client on your desktop.

## Download and configure the sample

You can now download and configure the remote monitoring client application on your Raspberry Pi.

### Install Node.js

If you haven't done so already, install Node.js on your Raspberry Pi. The IoT SDK for Node.js requires version 0.11.5 of Node.js or later. The following steps show you how to install Node.js v6.10.2 on your Raspberry Pi:

1. Use the following command to update your Raspberry Pi:

   ```
   sudo apt-get update
   ```

2. Use the following command to download the Node.js binaries to your Raspberry Pi:

```
wget https://nodejs.org/dist/v6.10.2/node-v6.10.2-linux-armv7l.tar.gz
```

3. Use the following command to install the binaries:

```
sudo tar -C /usr/local --strip-components 1 -xzf node-v6.10.2-linux-armv7l.tar.gz
```

4. Use the following command to verify you have installed Node.js v6.10.2 successfully:

```
node --version
```

**Clone the repositories**

If you haven't already done so, clone the required repositories by running the following commands in a terminal on your Pi:

```
cd ~
git clone --recursive https://github.com/Azure-Samples/iot-remote-monitoring-node-raspberrypi-getstartedkit.git
```

**Update the device connection string**

Open the sample source file in the **nano** editor using the following command:

```
nano ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/simulator/remote_monitoring.js
```

Find the line:

```
var connectionString = 'HostName=[Your IoT hub name].azure-devices.net;DeviceId=[Your device id];SharedAccessKey=[Your device key]';
```

Replace the placeholder values with the device and IoT Hub information you created and saved at the start of this tutorial. Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).

# Run the sample

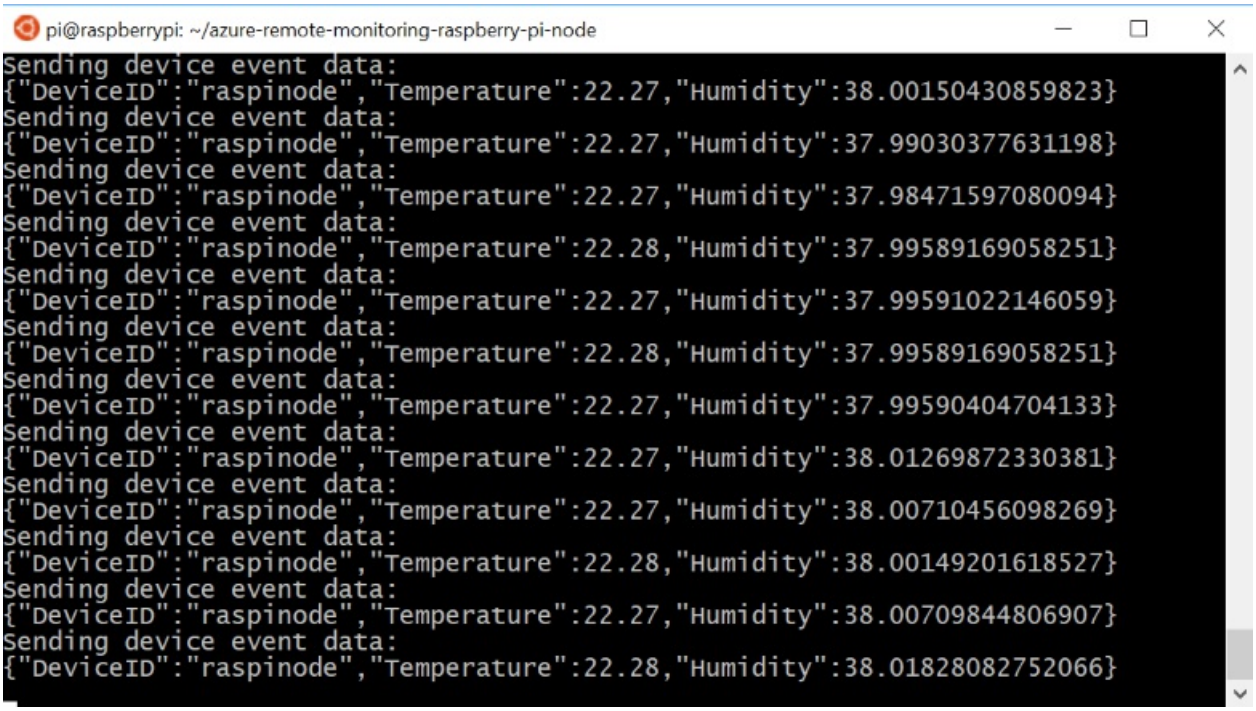
Run the following commands to install the prerequisite packages for the sample:

```
cd ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/simulator
npm install
```

You can now run the sample program on the Raspberry Pi. Enter the command:

```
sudo node ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/simulator/remote_monitoring.js
```

The following sample output is an example of the output you see at the command prompt on the Raspberry Pi:

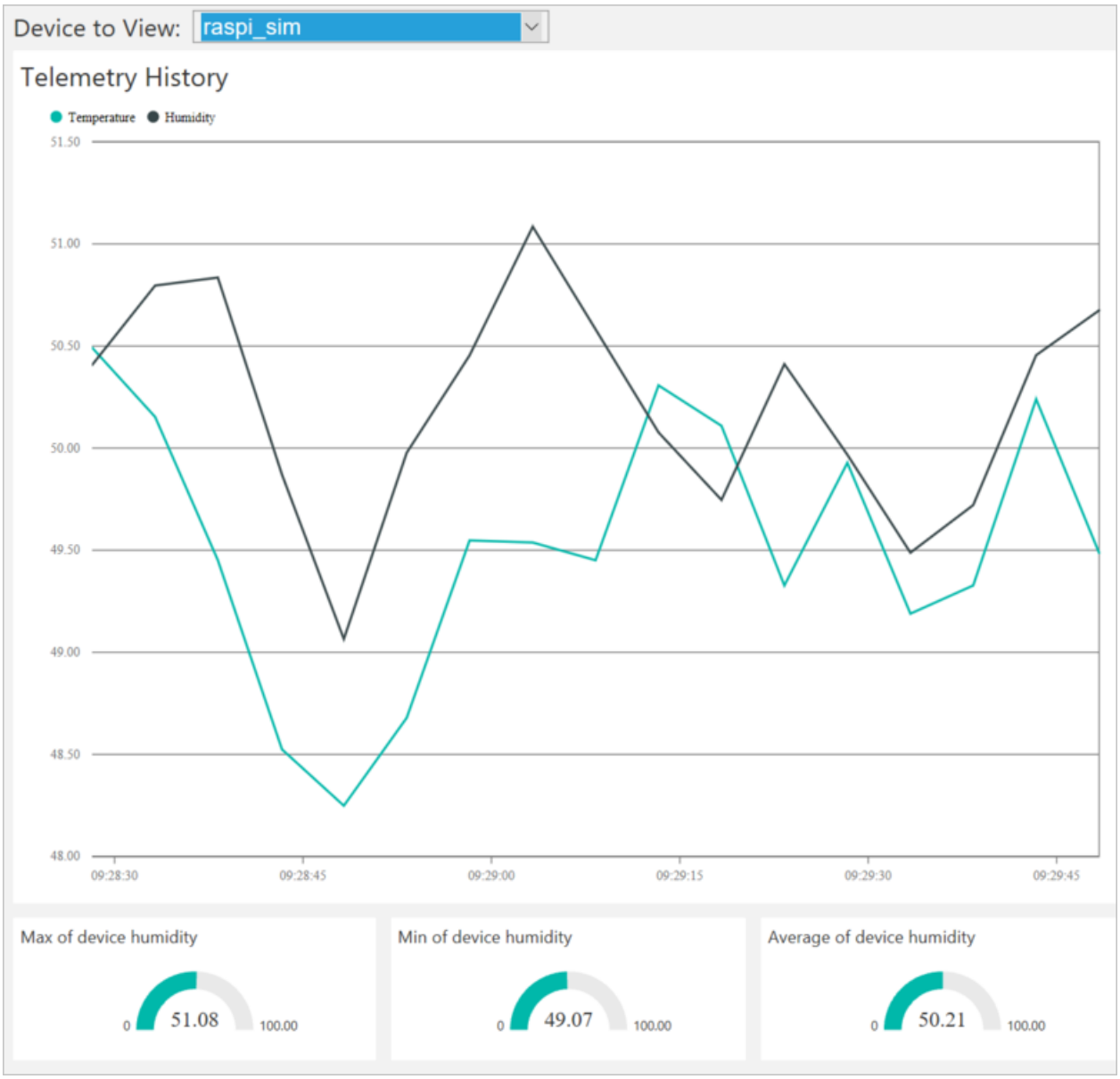



Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The Raspberry Pi is now sending telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send messages to your Raspberry Pi from the solution dashboard.

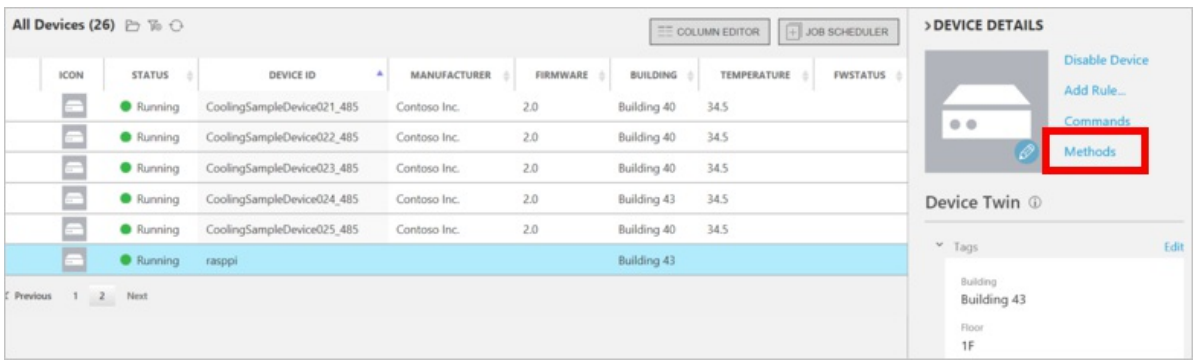
- Navigate to the solution dashboard.
- Select your device in the **Device to View** dropdown.
- The telemetry from the Raspberry Pi displays on the dashboard.



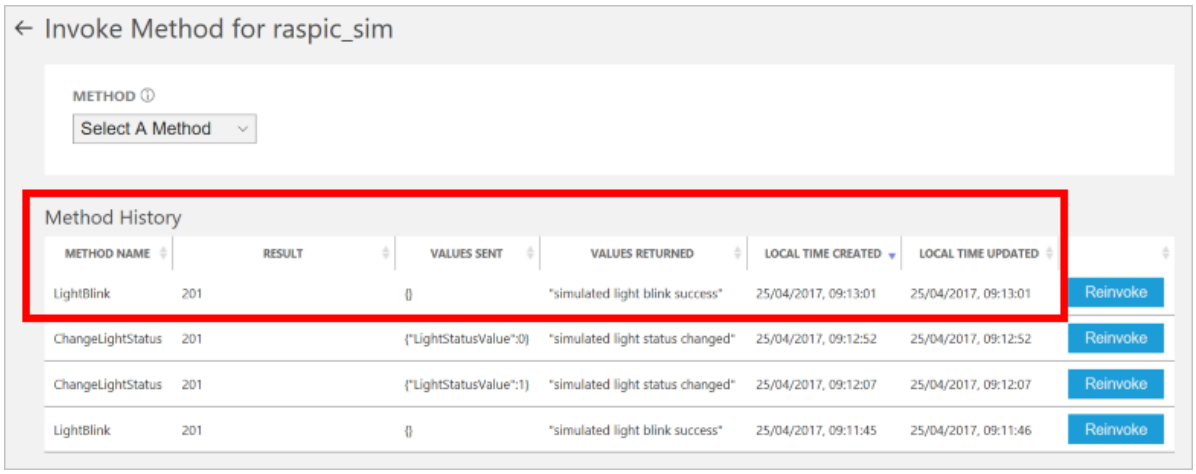

## Act on the device

From the solution dashboard, you can invoke methods on your Raspberry Pi. When the Raspberry Pi connects to the remote monitoring solution, it sends information about the methods it supports.

- In the solution dashboard, click **Devices** to visit the **Devices** page. Select your Raspberry Pi in the **Device List**. Then choose **Methods**:




- On the **Invoke Method** page, choose **LightBlink** in the **Method** dropdown.
- Choose **InvokeMethod**. The simulator prints a message in the console on the Raspberry Pi. The app on the Raspberry Pi sends an acknowledgment back to the solution dashboard:




- You can switch the LED on and off using the **ChangeLightStatus** method with a **LightStatusValue** set to **1** for on or **0** for off.

> **WARNING**
>
> If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Raspberry Pi 3 to the remote monitoring solution and send telemetry from a real sensor using Node.js

7/25/2017 • 9 min to read • Edit Online

This tutorial shows you how to use the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to develop a temperature and humidity reader that can communicate with the cloud. The tutorial uses:

- Raspbian OS, the Node.js programming language, and the Microsoft Azure IoT SDK for Node.js to implement a sample device.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your device and sensors to communicate with your computer and the remote monitoring solution.
- Update the sample device code to connect to the remote monitoring solution, and send telemetry that you can view on the solution dashboard.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Raspberry Pi.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility. For more information, see SSH Using Linux or Mac OS.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Raspberry Pi.

Microsoft IoT Starter Kit for Raspberry Pi 3 or equivalent components. This tutorial uses the following items from the kit:

- Raspberry Pi 3
- MicroSD Card (with NOOBS)
- A USB Mini cable
- An Ethernet cable

- BME280 sensor
- Breadboard
- Jumper wires
- Resistors
- LEDs

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

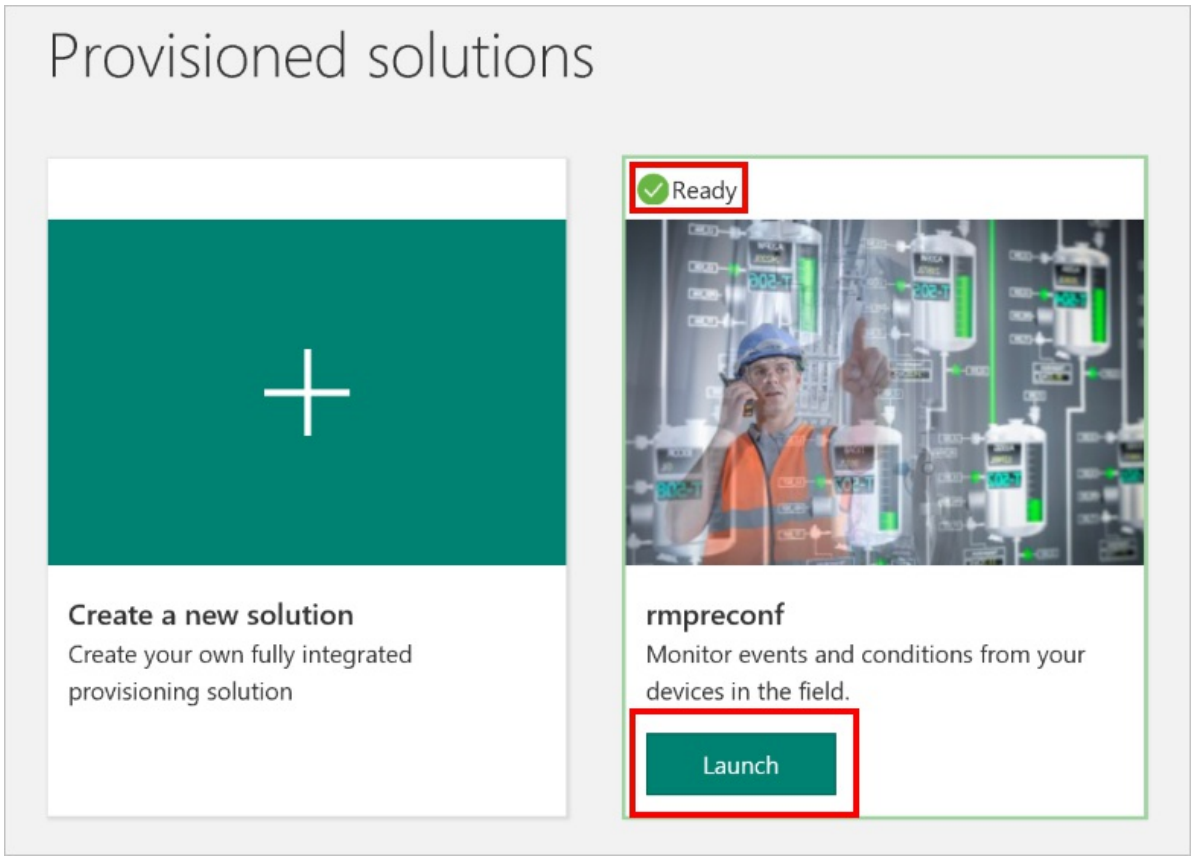
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

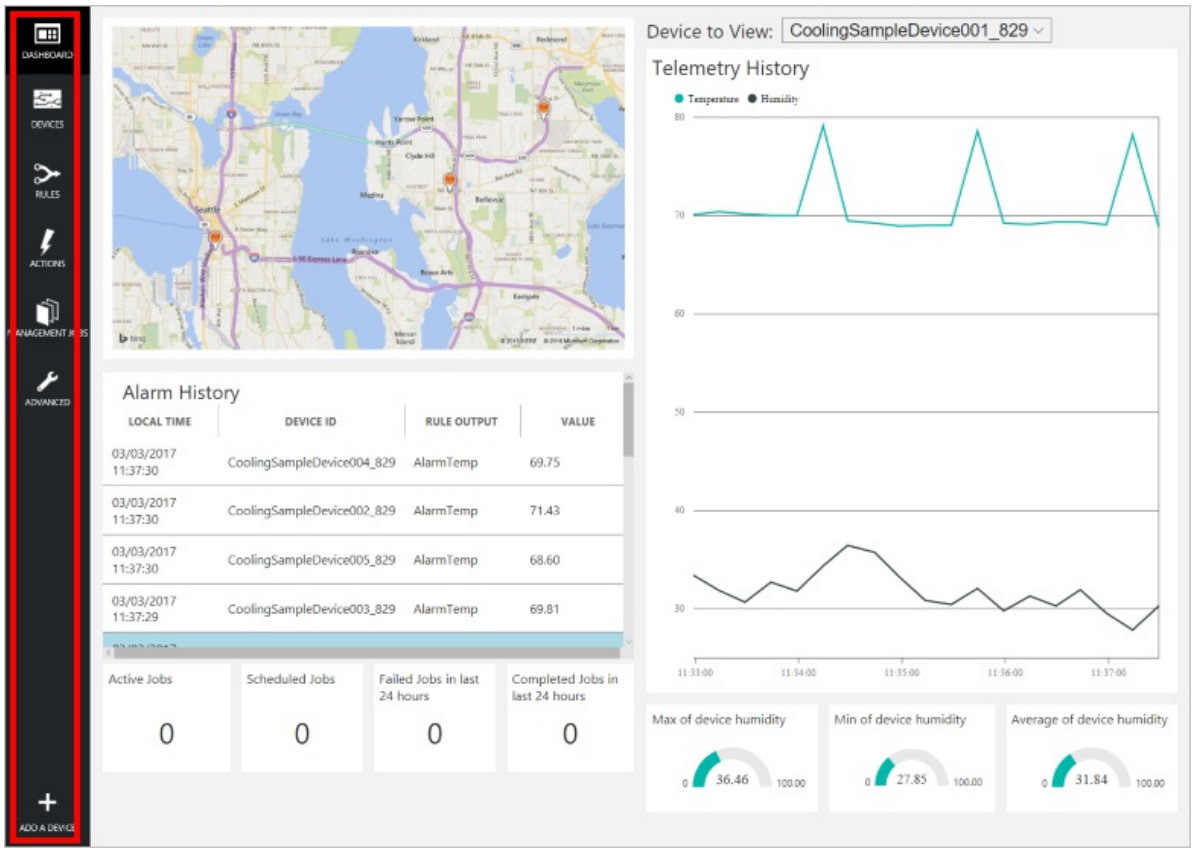
## View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.

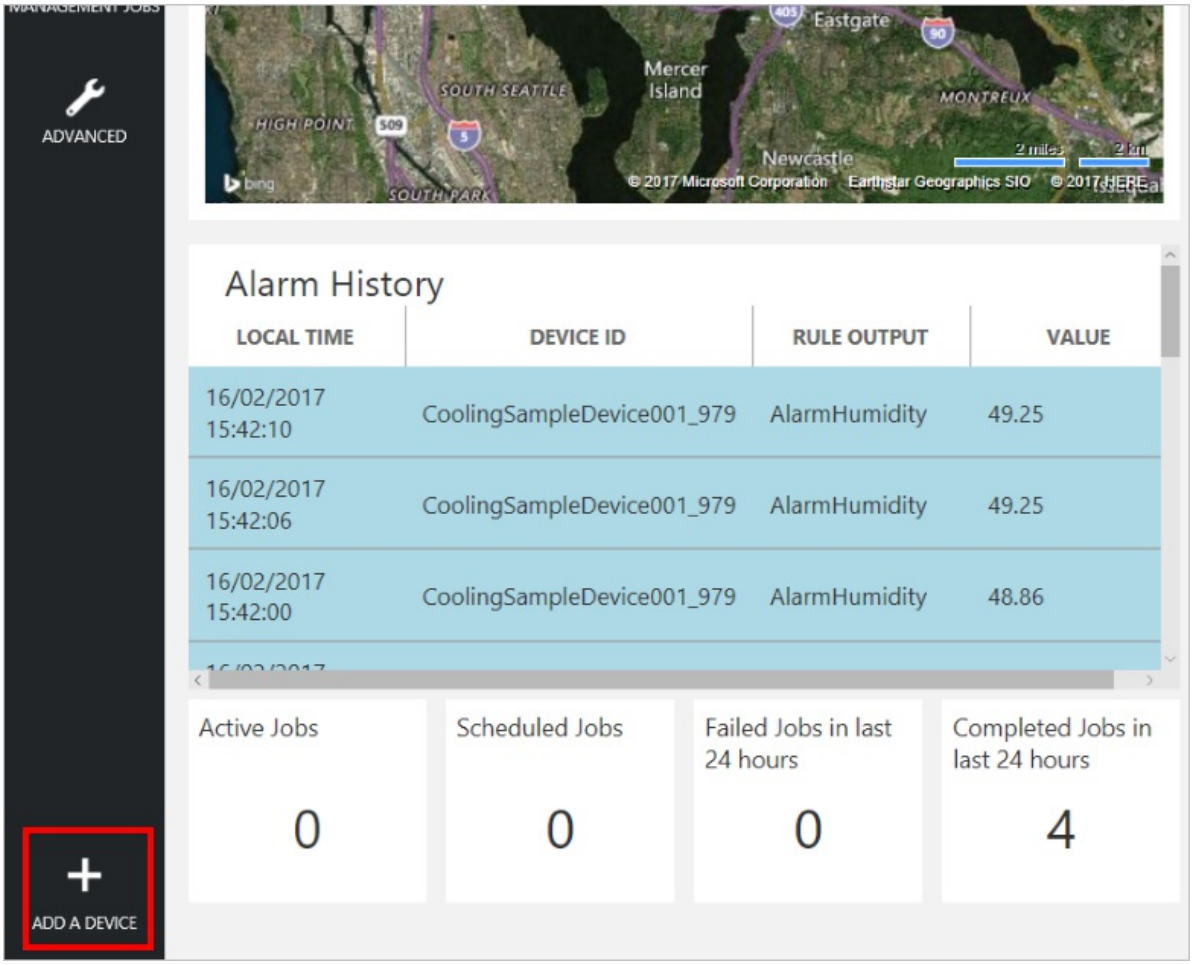



## Add a device

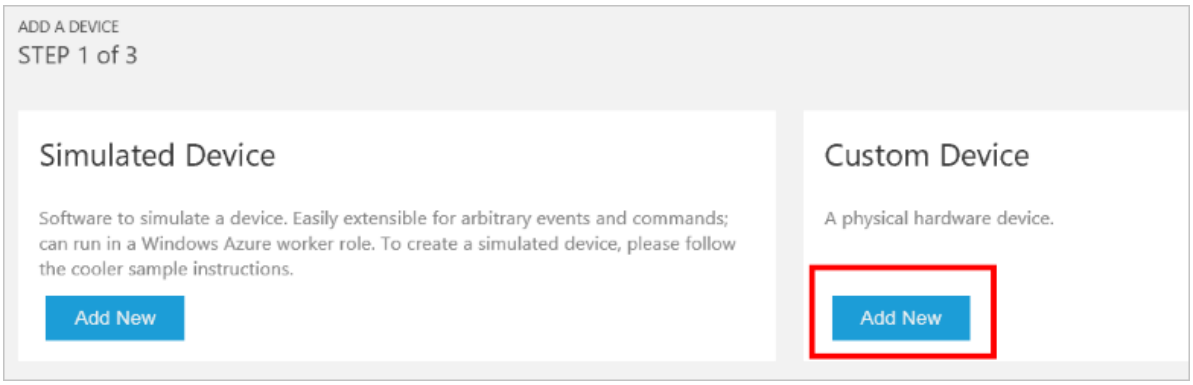
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

If you haven't already done so, add a custom device to your remote monitoring solution. Complete the following steps in the solution dashboard:

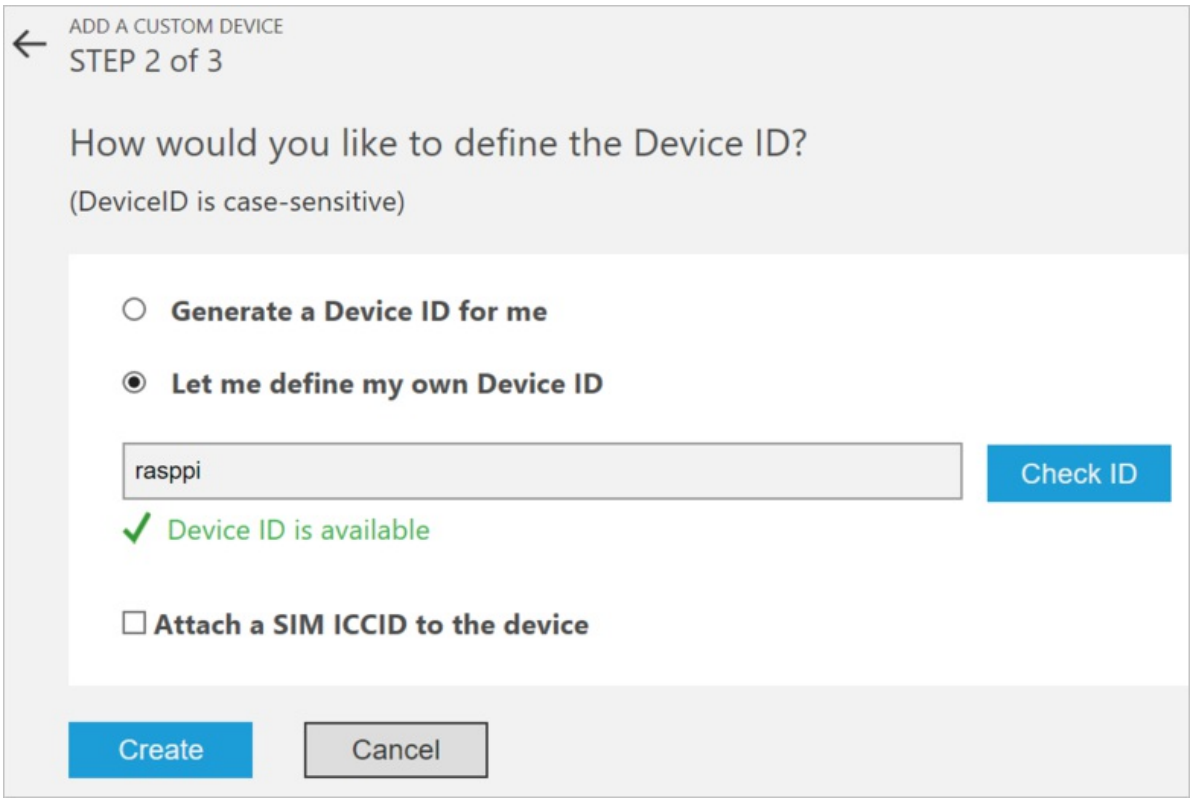
1. In the lower left-hand corner of the dashboard, click **Add a device**.



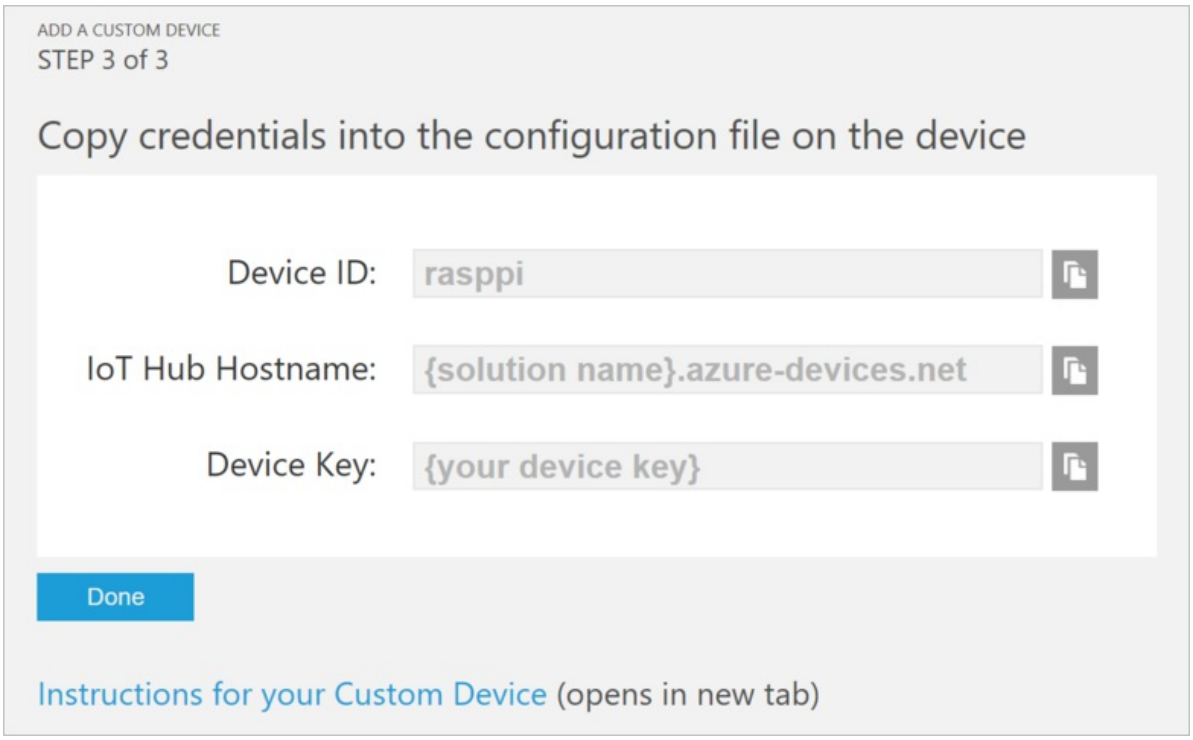

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **raspi**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Raspberry Pi needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Raspberry Pi

**Install Raspbian**

If this is the first time you are using your Raspberry Pi, you need to install the Raspbian operating system using NOOBS on the SD card included in the kit. The Raspberry Pi Software Guide describes how to install an operating

system on your Raspberry Pi. This tutorial assumes you have installed the Raspbian operating system on your Raspberry Pi.

> **NOTE**
> The SD card included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 already has NOOBS installed. You can boot the Raspberry Pi from this card and choose to install the Raspbian OS.

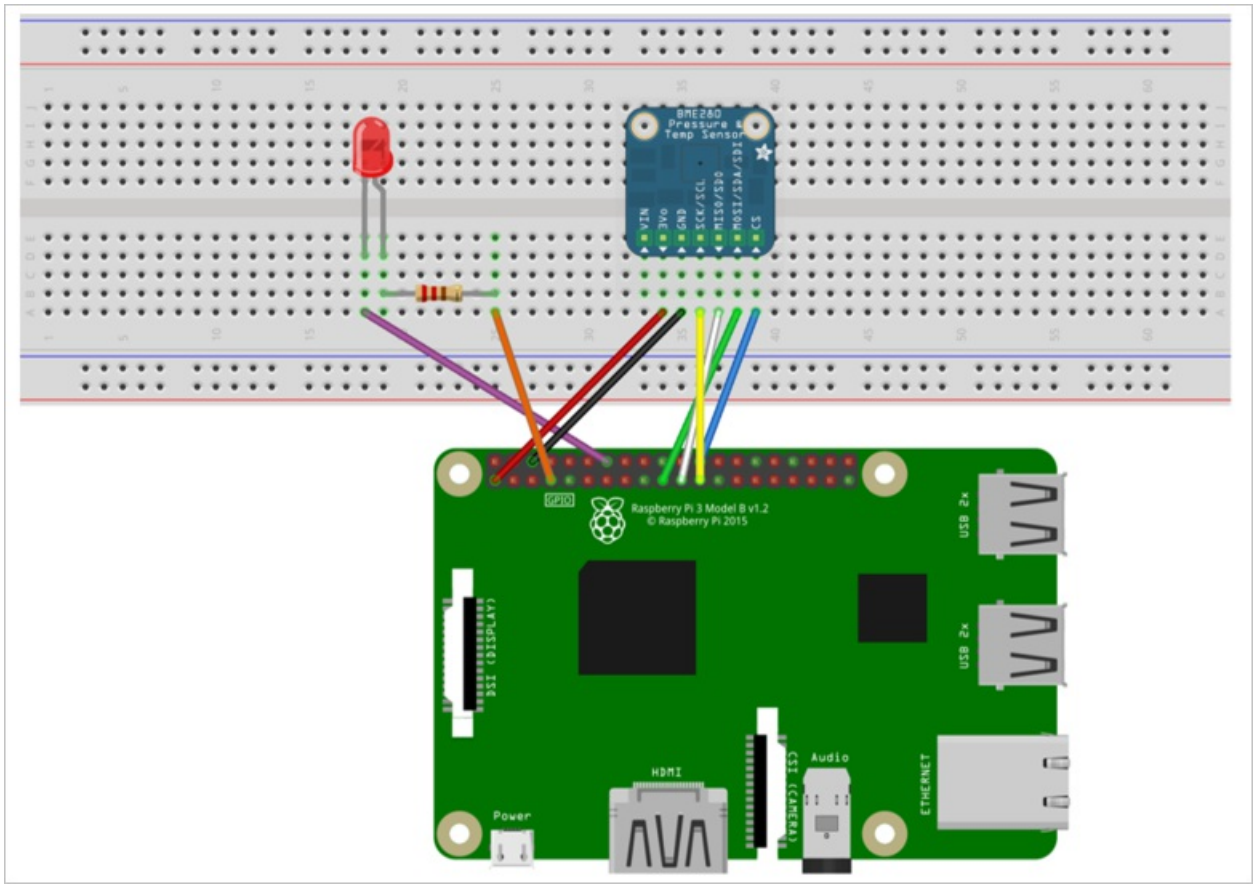
### Set up the hardware

This tutorial uses the BME280 sensor included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to generate telemetry data. It uses an LED to indicate when the Raspberry Pi processes a method invocation from the solution dashboard.

The components on the bread board are:

- Red LED
- 220-Ohm resistor (red, red, brown)
- BME280 sensor

The following diagram shows how to connect your hardware:




The following table summarizes the connections from the Raspberry Pi to the components on the breadboard:

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| GND (Pin 14) | LED -ve pin (18A) | Purple |
| GPCLK0 (Pin 7) | Resistor (25A) | Orange |
| SPI_CE0 (Pin 24) | CS (39A) | Blue |

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| SPI_SCLK (Pin 23) | SCK (36A) | Yellow |
| SPI_MISO (Pin 21) | SDO (37A) | White |
| SPI_MOSI (Pin 19) | SDI (38A) | Green |
| GND (Pin 6) | GND (35A) | Black |
| 3.3 V (Pin 1) | 3Vo (34A) | Red |

To complete the hardware setup, you need to:

- Connect your Raspberry Pi to the power supply included in the kit.
- Connect your Raspberry Pi to your network using the Ethernet cable included in your kit. Alternatively, you can set up Wireless Connectivity for your Raspberry Pi.

You have now completed the hardware setup of your Raspberry Pi.

### Sign in and access the terminal

You have two options to access a terminal environment on your Raspberry Pi:

- If you have a keyboard and monitor connected to your Raspberry Pi, you can use the Raspbian GUI to access a terminal window.
- Access the command line on your Raspberry Pi using SSH from your desktop machine.

#### Use a terminal Window in the GUI

The default credentials for Raspbian are username **pi** and password **raspberry**. In the task bar in the GUI, you can launch the **Terminal** utility using the icon that looks like a monitor.

#### Sign in with SSH

You can use SSH for command-line access to your Raspberry Pi. The article SSH (Secure Shell) describes how to configure SSH on your Raspberry Pi, and how to connect from Windows or Linux & Mac OS.

Sign in with username **pi** and password **raspberry**.

#### Optional: Share a folder on your Raspberry Pi

Optionally, you may want to share a folder on your Raspberry Pi with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Raspberry Pi instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Raspberry Pi. Alternatively, use the built-in SFTP server with an SFTP client on your desktop.

### Enable SPI

Before you can run the sample application, you must enable the Serial Peripheral Interface (SPI) bus on the Raspberry Pi. The Raspberry Pi communicates with the BME280 sensor device over the SPI bus. Use the following command to edit the configuration file:

```
sudo nano /boot/config.txt
```

Find the line:

```
#dtparam=spi=on
```

- To uncomment the line, delete the `#` at the start.
- Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).
- To enable SPI, reboot the Raspberry Pi. Rebooting disconnects the terminal, you need to sign in again when the Raspberry Pi restarts:

```
sudo reboot
```

# Download and configure the sample

You can now download and configure the remote monitoring client application on your Raspberry Pi.

### Install Node.js

Install Node.js on your Raspberry Pi. The IoT SDK for Node.js requires version 0.11.5 of Node.js or later. The following steps show you how to install Node.js v6.10.2 on your Raspberry Pi:

1. Use the following command to update your Raspberry Pi:

```
sudo apt-get update
```

2. Use the following command to download the Node.js binaries to your Raspberry Pi:

```
wget https://nodejs.org/dist/v6.10.2/node-v6.10.2-linux-armv7l.tar.gz
```

3. Use the following command to install the binaries:

```
sudo tar -C /usr/local --strip-components 1 -xzf node-v6.10.2-linux-armv7l.tar.gz
```

4. Use the following command to verify you have installed Node.js v6.10.2 successfully:

```
node --version
```

### Clone the repositories

If you haven't already done so, clone the required repositories by running the following commands on your Pi:

```
cd ~
git clone --recursive https://github.com/Azure-Samples/iot-remote-monitoring-node-raspberrypi-
getstartedkit.git`
```

### Update the device connection string

Open the sample source file in the **nano** editor using the following command:

```
nano ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/basic/remote_monitoring.js
```

Find the line:

```
var connectionString = 'HostName=[Your IoT hub name].azure-devices.net;DeviceId=[Your device
id];SharedAccessKey=[Your device key]';
```

Replace the placeholder values with the device and IoT Hub information you created and saved at the start of this

tutorial. Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).

## Run the sample

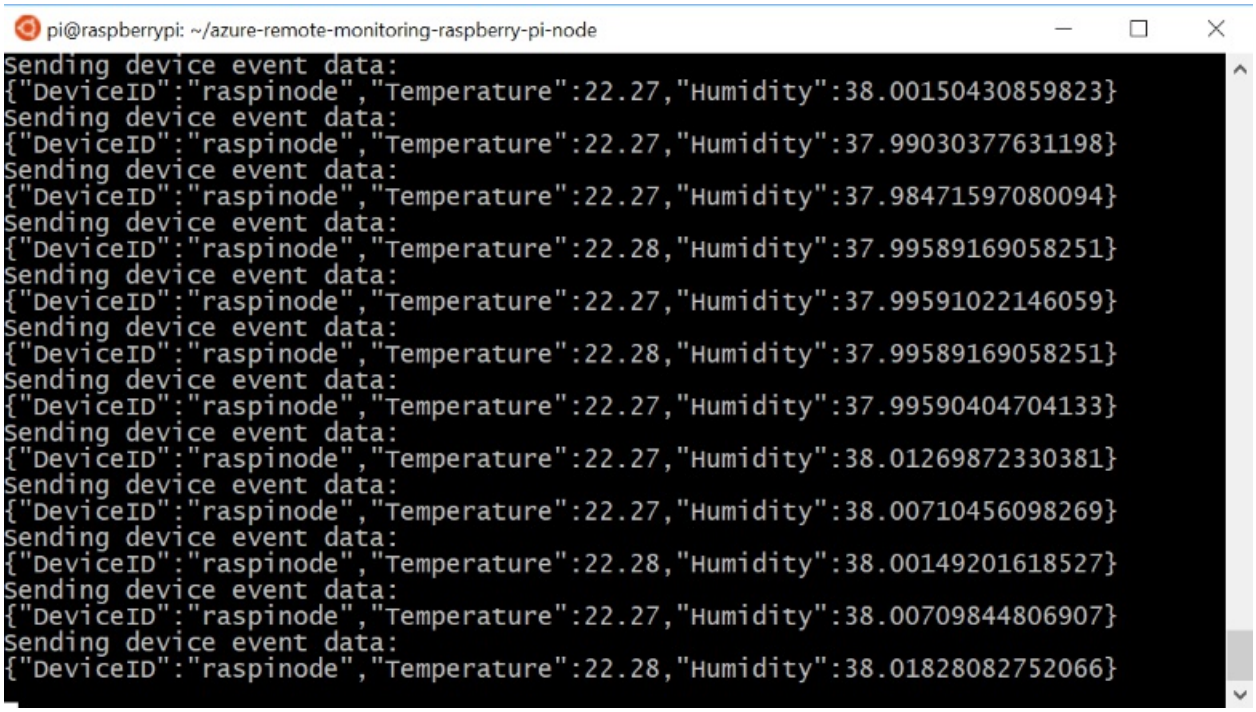
Run the following commands to install the prerequisite packages for the sample:

```
cd ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/basic
npm install
```

You can now run the sample program on the Raspberry Pi. Enter the command:

```
sudo node ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/basic/remote_monitoring.js
```

The following sample output is an example of the output you see at the command prompt on the Raspberry Pi:

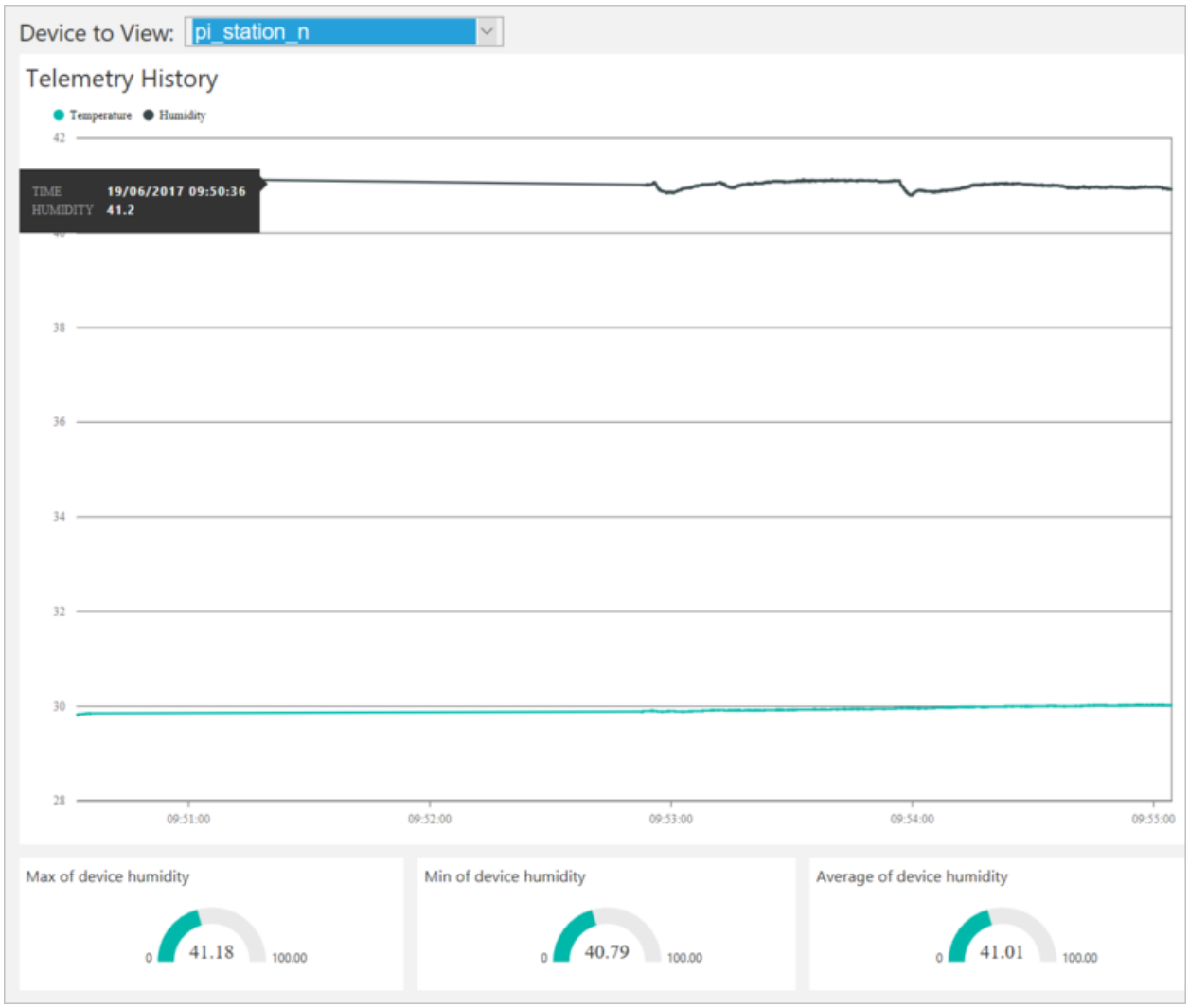



Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The Raspberry Pi is now sending telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send messages to your Raspberry Pi from the solution dashboard.

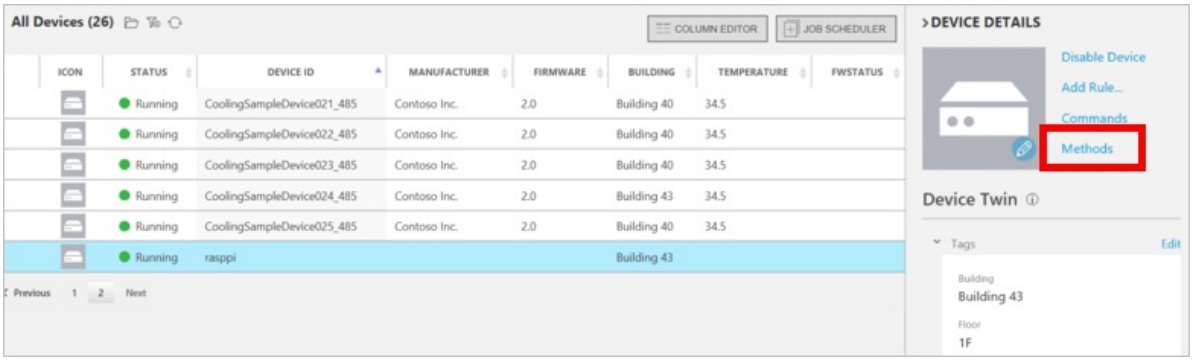
- Navigate to the solution dashboard.
- Select your device in the **Device to View** dropdown.
- The telemetry from the Raspberry Pi displays on the dashboard.



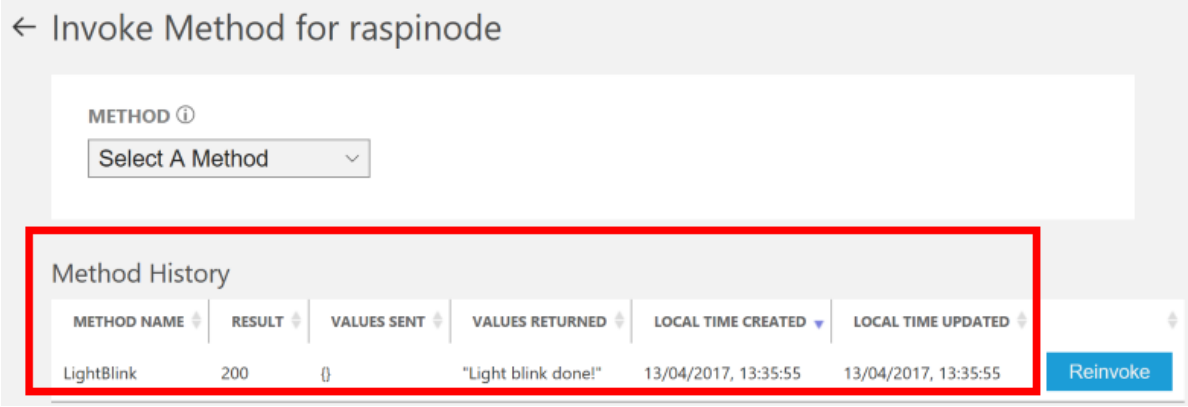

## Act on the device

From the solution dashboard, you can invoke methods on your Raspberry Pi. When the Raspberry Pi connects to the remote monitoring solution, it sends information about the methods it supports.

- In the solution dashboard, click **Devices** to visit the **Devices** page. Select your Raspberry Pi in the **Device List**. Then choose **Methods**:




- On the **Invoke Method** page, choose **LightBlink** in the **Method** dropdown.
- Choose **InvokeMethod**. The LED connected to the Raspberry Pi flashes several times. The app on the Raspberry Pi sends an acknowledgment back to the solution dashboard:




- You can switch the LED on and off using the **ChangeLightStatus** method with a **LightStatusValue** set to **1** for on or **0** for off.

> **WARNING**
> If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Raspberry Pi 3 to the remote monitoring solution and enable remote firmware updates using Node.js

7/25/2017 • 11 min to read • Edit Online

This tutorial shows you how to use the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to:

- Develop a temperature and humidity reader that can communicate with the cloud.
- Enable and perform a remote firmware update to update the client application on the Raspberry Pi.

The tutorial uses:

- Raspbian OS, the Node.js programming language, and the Microsoft Azure IoT SDK for Node.js to implement a sample device.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your device and sensors to communicate with your computer and the remote monitoring solution.
- Update the sample device code to connect to the remote monitoring solution, and send telemetry that you can view on the solution dashboard.
- Use the sample device code to update the client application.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Raspberry Pi.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility. For more information, see SSH Using Linux or Mac OS.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Raspberry Pi.

Microsoft IoT Starter Kit for Raspberry Pi 3 or equivalent components. This tutorial uses the following items from the kit:

- Raspberry Pi 3
- MicroSD Card (with NOOBS)
- A USB Mini cable
- An Ethernet cable
- BME280 sensor
- Breadboard
- Jumper wires
- Resistors
- LEDs

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

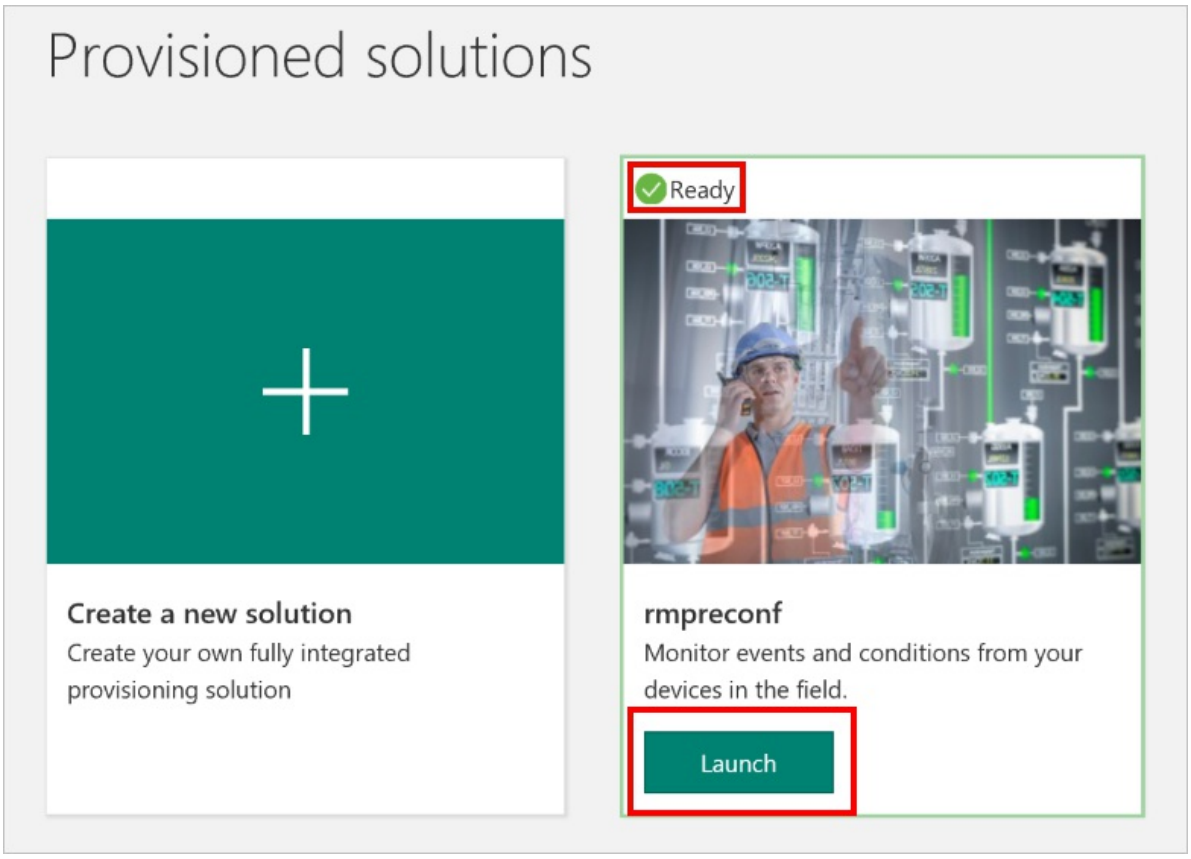
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

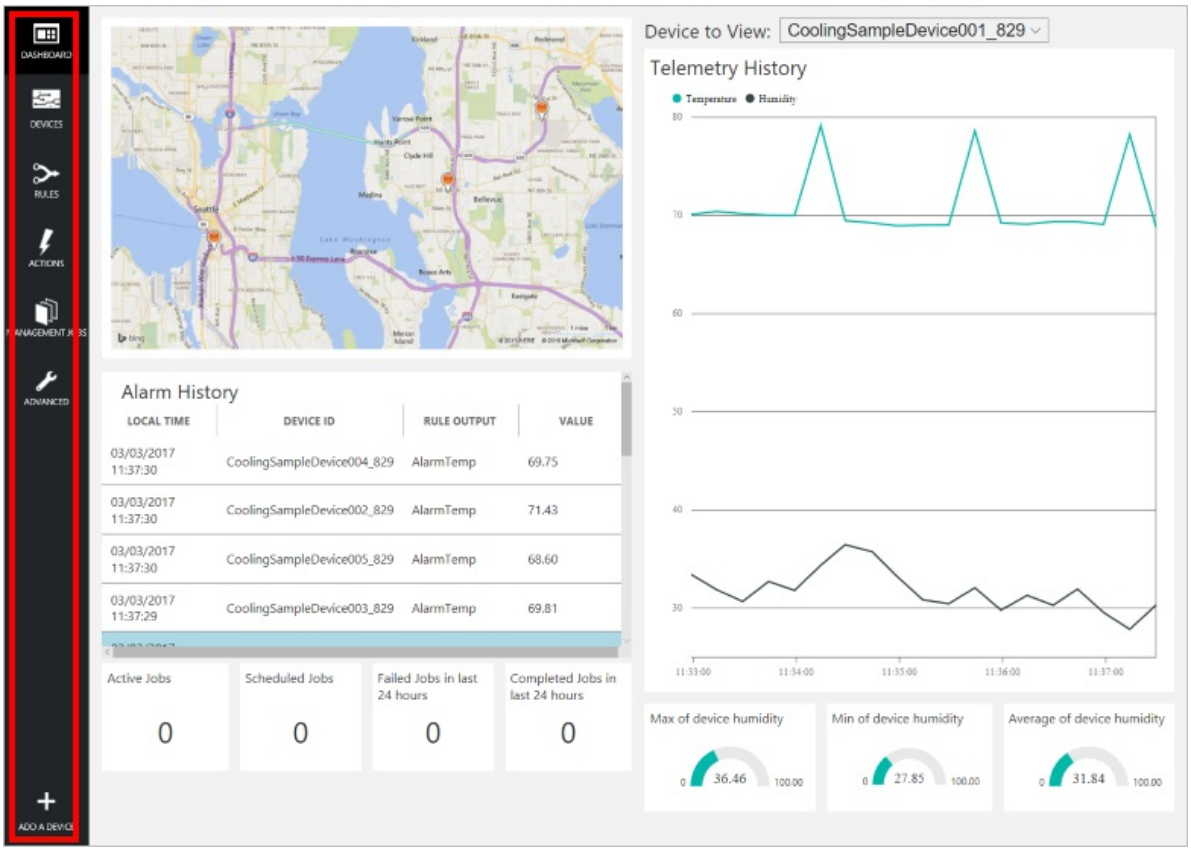
## View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. You can navigate to other areas of the solution portal using the menu on the left-hand side of the page.

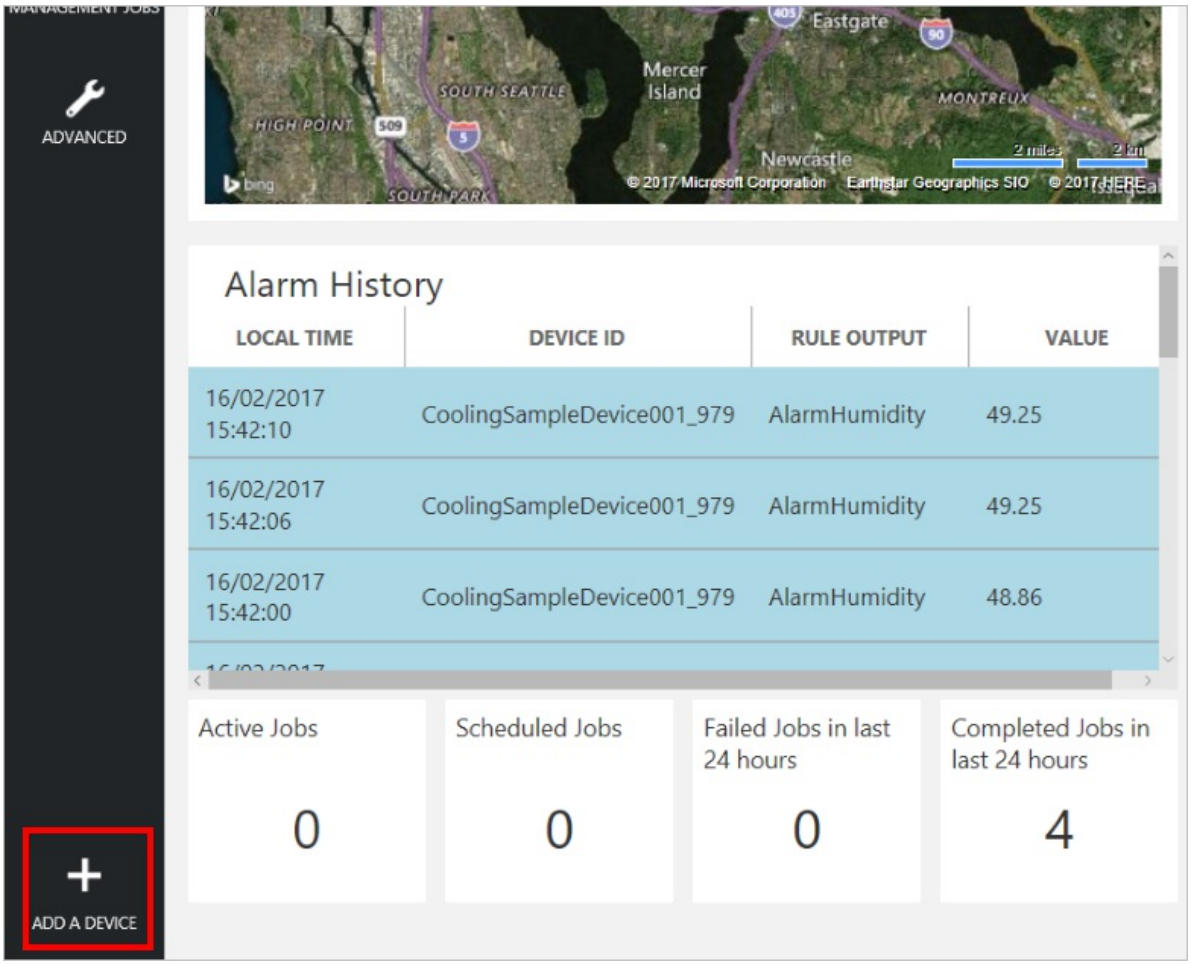



## Add a device

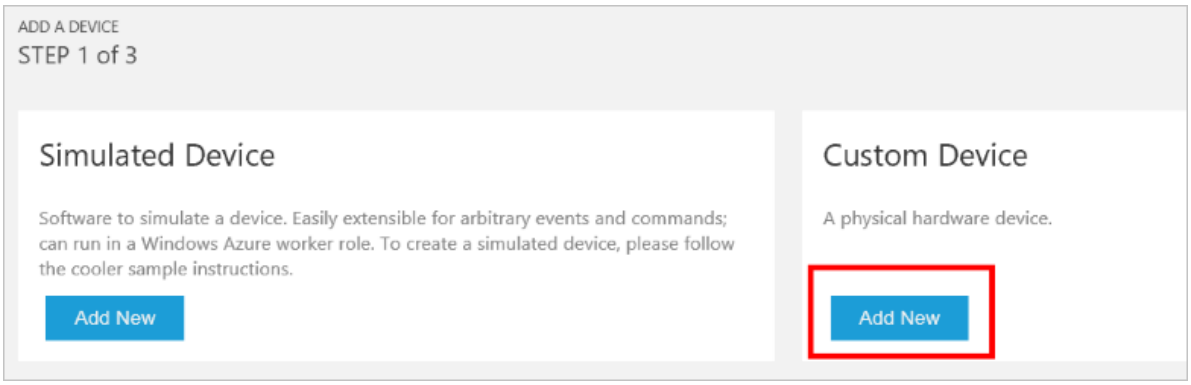
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

If you haven't already done so, add a custom device to your remote monitoring solution. Complete the following steps in the solution dashboard:

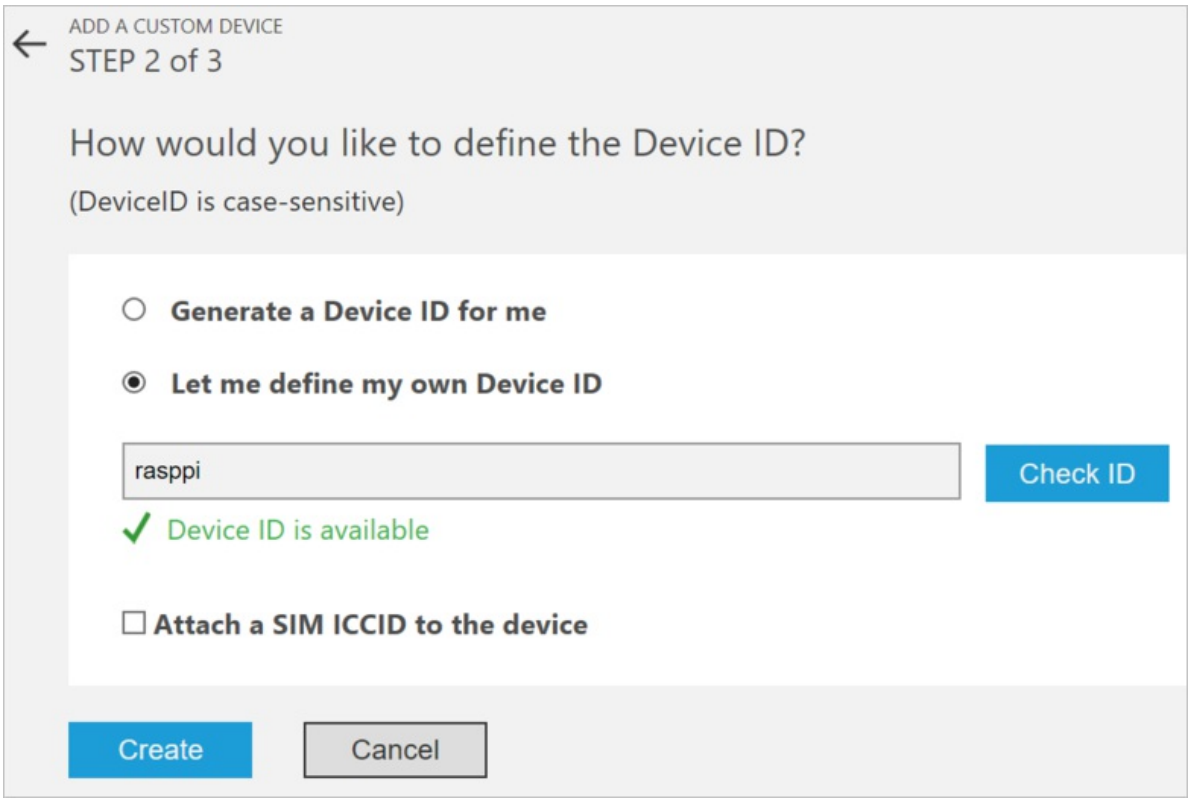
1. In the lower left-hand corner of the dashboard, click **Add a device**.



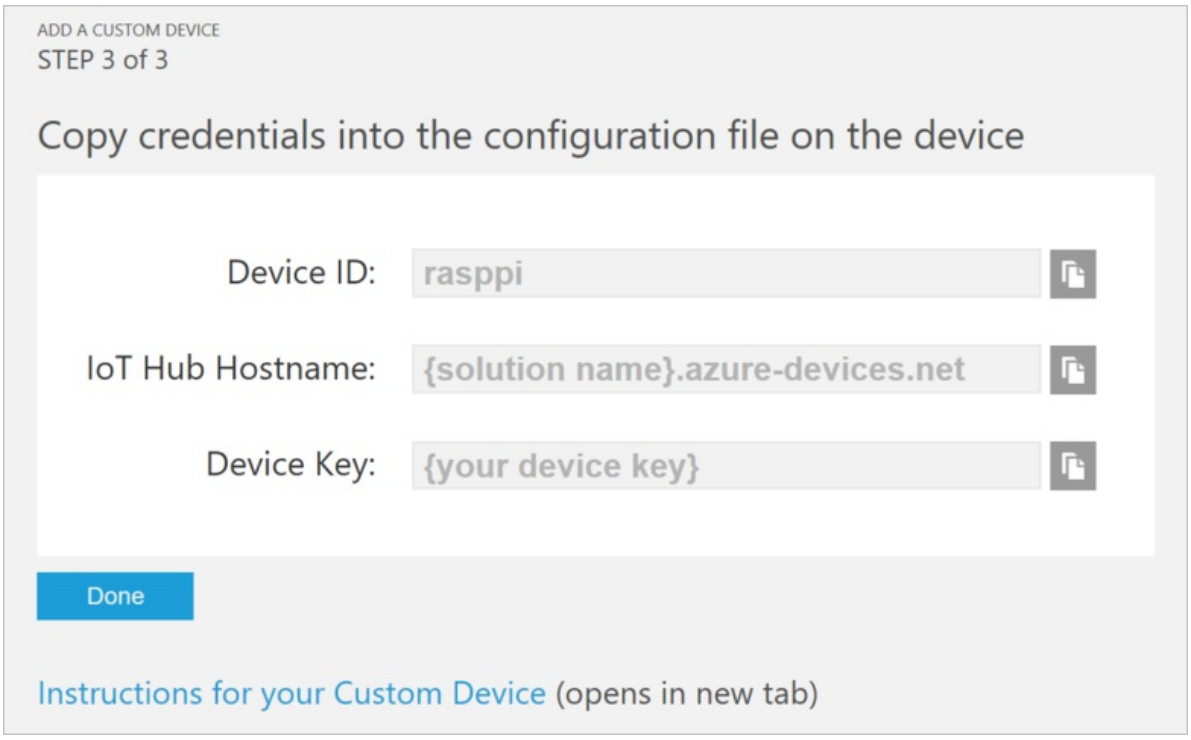

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **rasppi**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Raspberry Pi needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Raspberry Pi

**Install Raspbian**

If this is the first time you are using your Raspberry Pi, you need to install the Raspbian operating system using NOOBS on the SD card included in the kit. The Raspberry Pi Software Guide describes how to install an operating

system on your Raspberry Pi. This tutorial assumes you have installed the Raspbian operating system on your Raspberry Pi.

> **NOTE**
> The SD card included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 already has NOOBS installed. You can boot the Raspberry Pi from this card and choose to install the Raspbian OS.

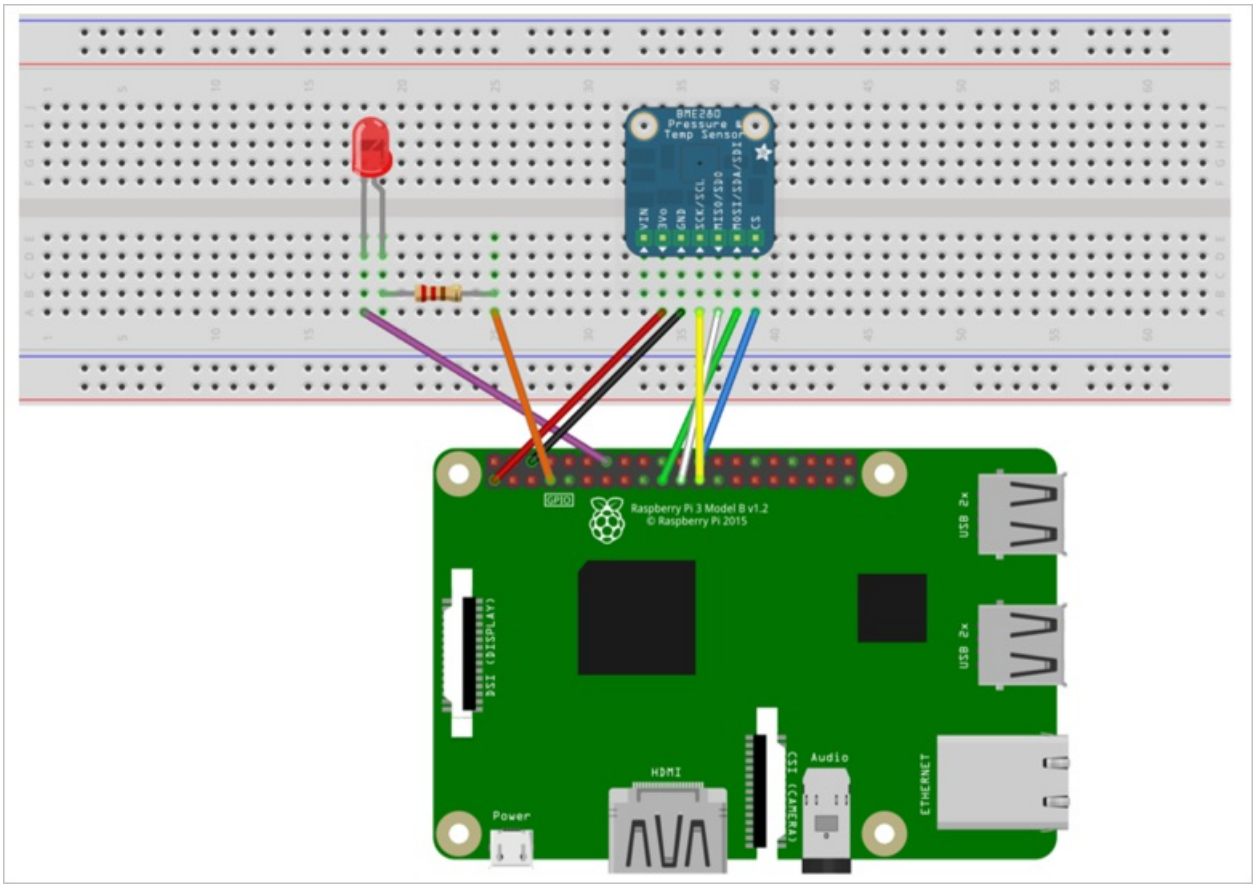
### Set up the hardware

This tutorial uses the BME280 sensor included in the Microsoft Azure IoT Starter Kit for Raspberry Pi 3 to generate telemetry data. It uses an LED to indicate when the Raspberry Pi processes a method invocation from the solution dashboard.

The components on the bread board are:

- Red LED
- 220-Ohm resistor (red, red, brown)
- BME280 sensor

The following diagram shows how to connect your hardware:




The following table summarizes the connections from the Raspberry Pi to the components on the breadboard:

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| GND (Pin 14) | LED -ve pin (18A) | Purple |
| GPCLK0 (Pin 7) | Resistor (25A) | Orange |
| SPI_CE0 (Pin 24) | CS (39A) | Blue |

| RASPBERRY PI | BREADBOARD | COLOR |
|---|---|---|
| SPI_SCLK (Pin 23) | SCK (36A) | Yellow |
| SPI_MISO (Pin 21) | SDO (37A) | White |
| SPI_MOSI (Pin 19) | SDI (38A) | Green |
| GND (Pin 6) | GND (35A) | Black |
| 3.3 V (Pin 1) | 3Vo (34A) | Red |

To complete the hardware setup, you need to:

- Connect your Raspberry Pi to the power supply included in the kit.
- Connect your Raspberry Pi to your network using the Ethernet cable included in your kit. Alternatively, you can set up Wireless Connectivity for your Raspberry Pi.

You have now completed the hardware setup of your Raspberry Pi.

### Sign in and access the terminal

You have two options to access a terminal environment on your Raspberry Pi:

- If you have a keyboard and monitor connected to your Raspberry Pi, you can use the Raspbian GUI to access a terminal window.
- Access the command line on your Raspberry Pi using SSH from your desktop machine.

#### Use a terminal Window in the GUI

The default credentials for Raspbian are username **pi** and password **raspberry**. In the task bar in the GUI, you can launch the **Terminal** utility using the icon that looks like a monitor.

#### Sign in with SSH

You can use SSH for command-line access to your Raspberry Pi. The article SSH (Secure Shell) describes how to configure SSH on your Raspberry Pi, and how to connect from Windows or Linux & Mac OS.

Sign in with username **pi** and password **raspberry**.

#### Optional: Share a folder on your Raspberry Pi

Optionally, you may want to share a folder on your Raspberry Pi with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Raspberry Pi instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Raspberry Pi. Alternatively, use the built-in SFTP server with an SFTP client on your desktop.

### Enable SPI

Before you can run the sample application, you must enable the Serial Peripheral Interface (SPI) bus on the Raspberry Pi. The Raspberry Pi communicates with the BME280 sensor device over the SPI bus. Use the following command to edit the configuration file:

```
sudo nano /boot/config.txt
```

Find the line:

`#dtparam=spi=on`

- To uncomment the line, delete the `#` at the start.
- Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).
- To enable SPI, reboot the Raspberry Pi. Rebooting disconnects the terminal, you need to sign in again when the Raspberry Pi restarts:

```
sudo reboot
```

# Download and configure the sample

You can now download and configure the remote monitoring client application on your Raspberry Pi.

### Install Node.js

If you haven't done so already, install Node.js on your Raspberry Pi. The IoT SDK for Node.js requires version 0.11.5 of Node.js or later. The following steps show you how to install Node.js v6.10.2 on your Raspberry Pi:

1. Use the following command to update your Raspberry Pi:

```
sudo apt-get update
```

2. Use the following command to download the Node.js binaries to your Raspberry Pi:

```
wget https://nodejs.org/dist/v6.10.2/node-v6.10.2-linux-armv7l.tar.gz
```

3. Use the following command to install the binaries:

```
sudo tar -C /usr/local --strip-components 1 -xzf node-v6.10.2-linux-armv7l.tar.gz
```

4. Use the following command to verify you have installed Node.js v6.10.2 successfully:

```
node --version
```

### Clone the repositories

If you haven't done so already, clone the required repositories by running the following commands on your Pi:

```
cd ~
git clone --recursive https://github.com/Azure-Samples/iot-remote-monitoring-node-raspberrypi-
getstartedkit.git
```

### Update the device connection string

Open the sample configuration file in the **nano** editor using the following command:

```
nano ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/advanced/config/deviceinfo
```

Replace the placeholder values with the device id and IoT Hub information you created and saved at the start of this tutorial.

When you are done, the contents of the deviceinfo file should look like the following example:

```
yourdeviceid
HostName=youriothubname.azure-devices.net;DeviceId=yourdeviceid;SharedAccessKey=yourdevicekey
```

Save your changes (**Ctrl-O**, **Enter**) and exit the editor (**Ctrl-X**).

## Run the sample

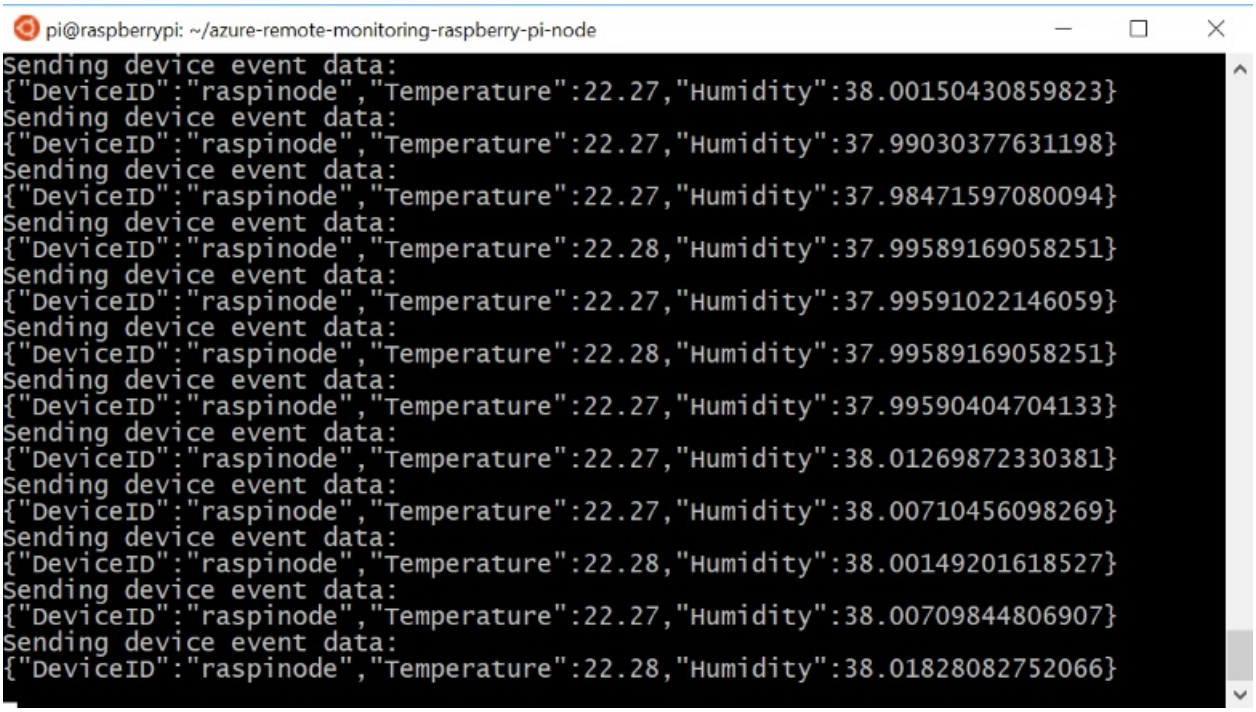
Run the following commands to install the prerequisite packages for the sample:

```
cd ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/advance/1.0
npm install
```

You can now run the sample program on the Raspberry Pi. Enter the command:

```
sudo node ~/iot-remote-monitoring-node-raspberrypi-getstartedkit/advanced/1.0/remote_monitoring.js
```

The following sample output is an example of the output you see at the command prompt on the Raspberry Pi:

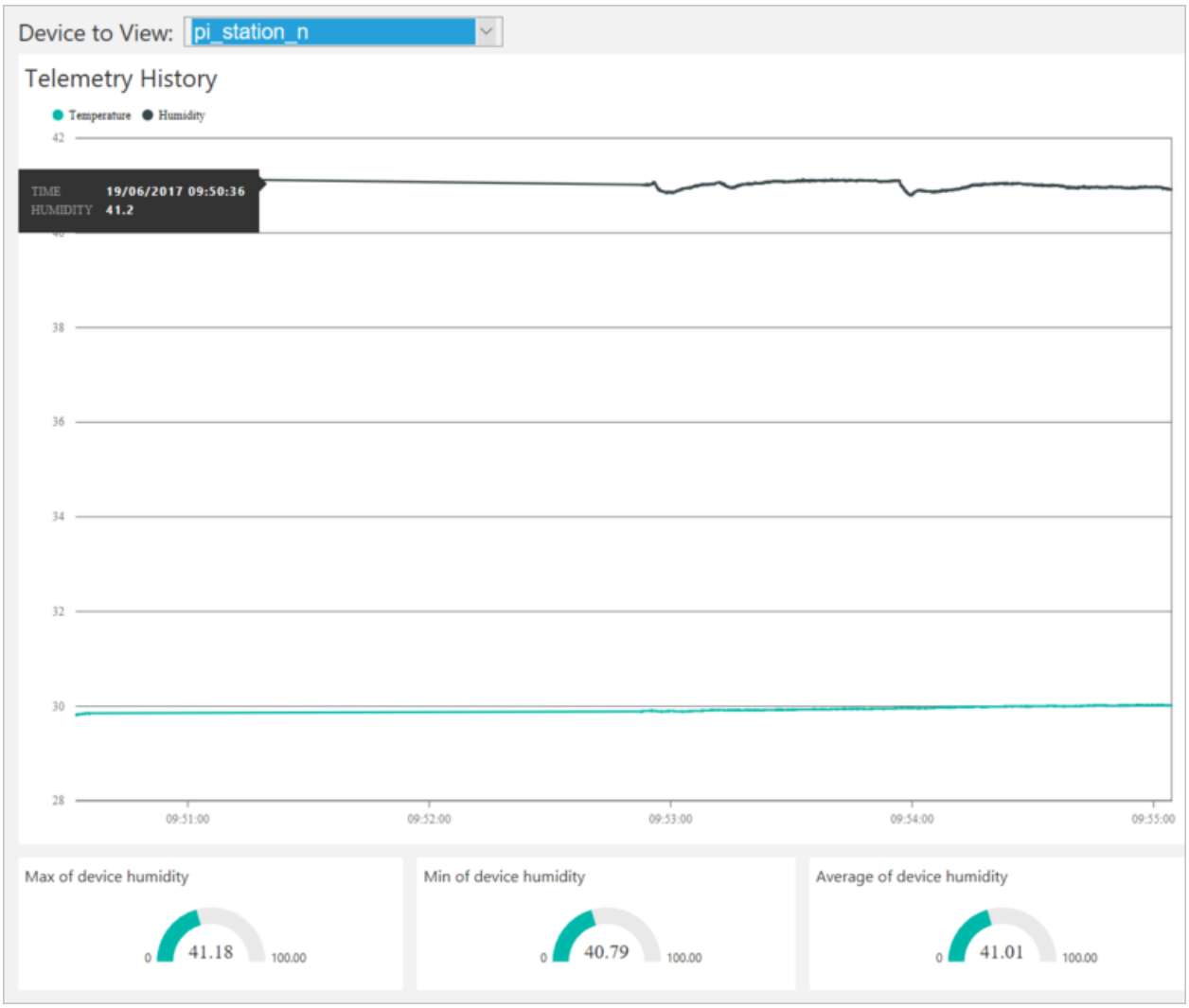



Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The Raspberry Pi is now sending telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send messages to your Raspberry Pi from the solution dashboard.

- Navigate to the solution dashboard.
- Select your device in the **Device to View** dropdown.
- The telemetry from the Raspberry Pi displays on the dashboard.



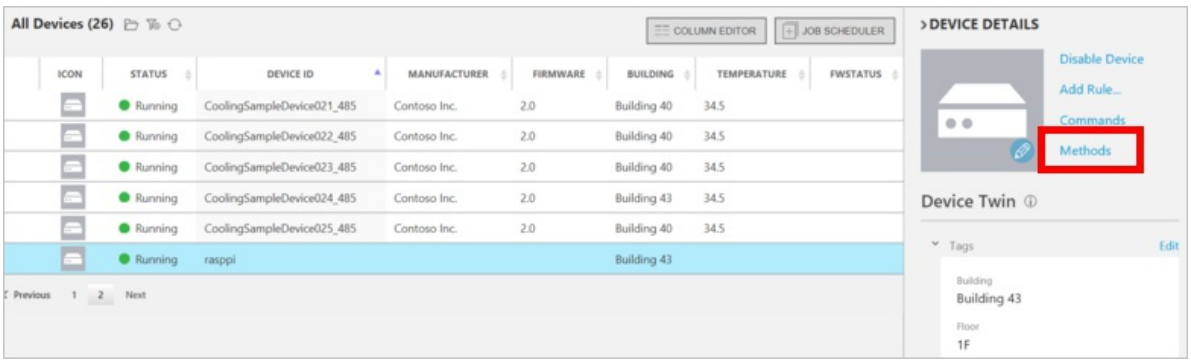

## Initiate the firmware update

The firmware update process downloads and installs an updated version of the device client application on the Raspberry Pi. For more information about the firmware update process, see the description of the firmware update pattern in Overview of device management with IoT Hub.

You initiate the firmware update process by invoking a method on the device. This method is asynchronous, and returns as soon as the update process begins. The device uses reported properties to notify the solution about the progress of the update.

You invoke methods on your Raspberry Pi from the solution dashboard. When the Raspberry Pi first connects to the remote monitoring solution, it sends information about the methods it supports.

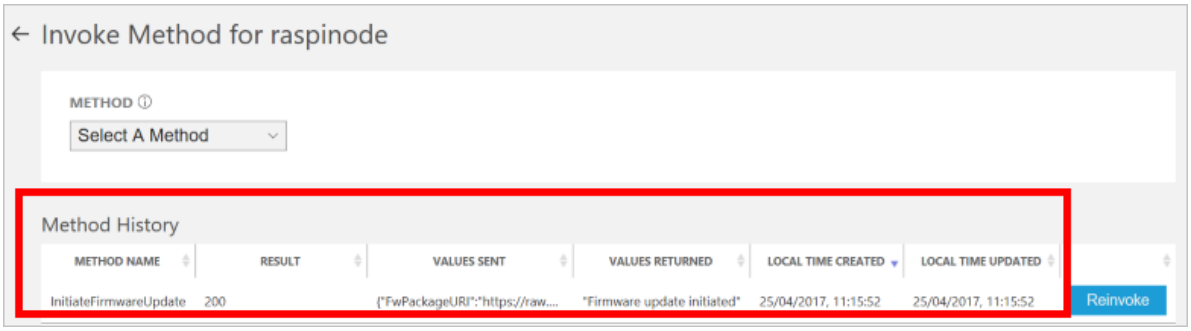
1. In the solution dashboard, click **Devices** to visit the **Devices** page. Select your Raspberry Pi in the **Device List**. Then choose **Methods**:




2. On the **Invoke Method** page, choose **InitiateFirmwareUpdate** in the **Method** dropdown.

3. In the **FWPackageURI** field, enter **https://raw.githubusercontent.com/Azure-Samples/iot-remote-monitoring-node-raspberrypi-getstartedkit/master/advanced/2.0/raspberry.js**. This file contains the implementation of version 2.0 of the firmware.

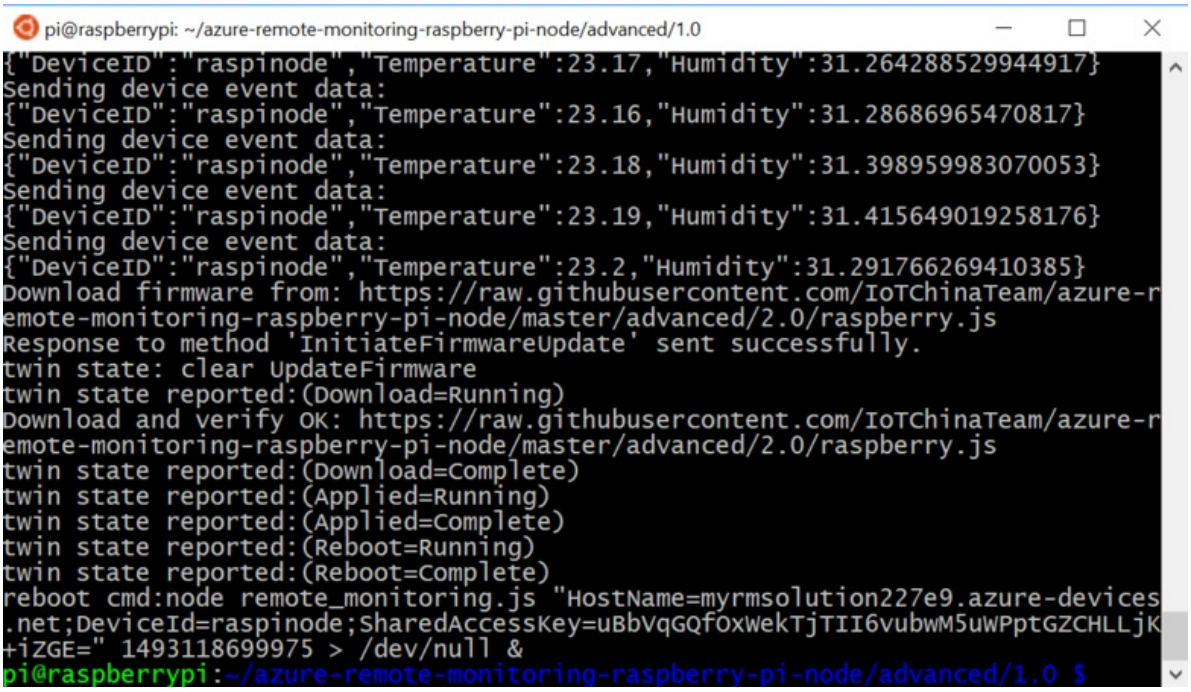
4. Choose **InvokeMethod**. The app on the Raspberry Pi sends an acknowledgment back to the solution dashboard. It then starts the firmware update process by downloading the new version of the firmware:




## Observe the firmware update process

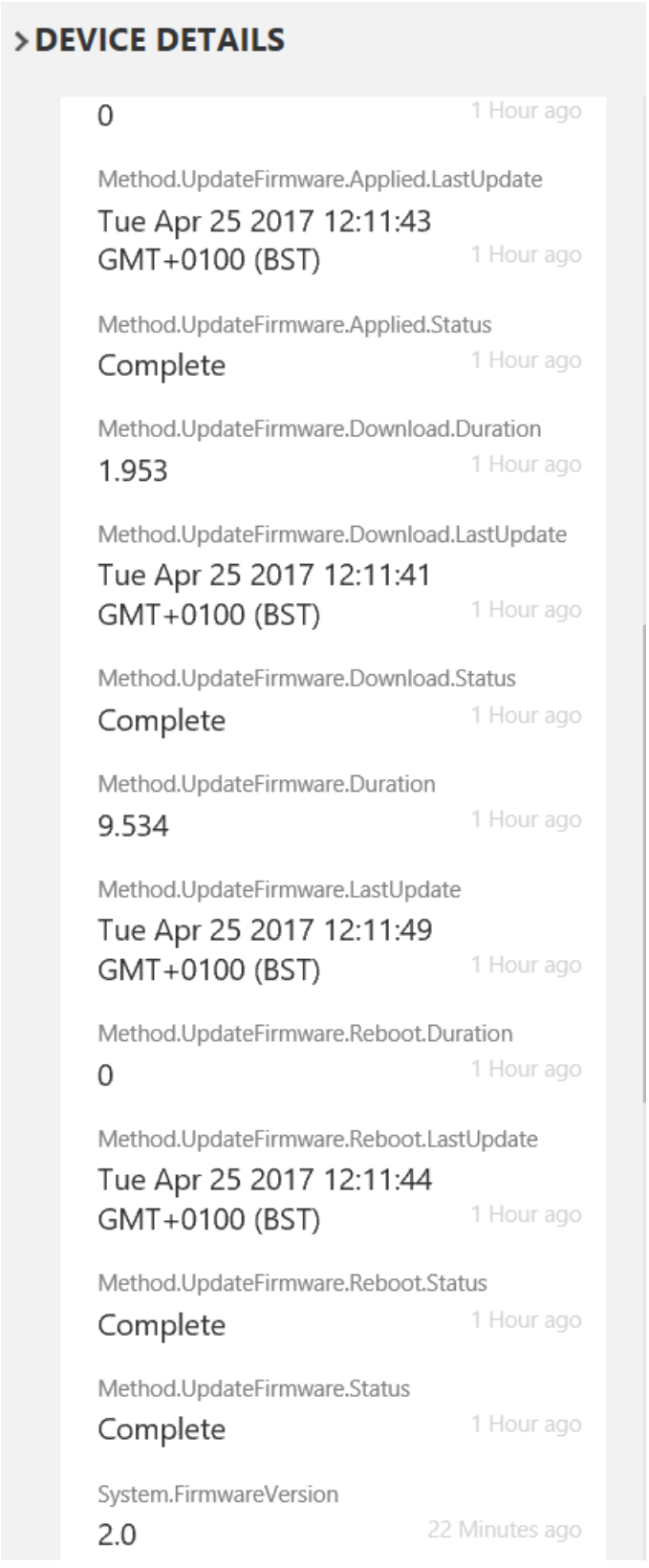
You can observe the firmware update process as it runs on the device and by viewing the reported properties in the solution dashboard:

1. You can view the progress in of the update process on the Raspberry Pi:




> **NOTE**
>
> The remote monitoring app restarts silently when the update completes. Use the command `ps -ef` to verify it is running. If you want to terminate the process, use the `kill` command with the process id.

2. You can view the status of the firmware update, as reported by the device, in the solution portal. The following screenshot shows the status and duration of each stage of the update process, and the new firmware version:




If you navigate back to the dashboard, you can verify the device is still sending telemetry following the firmware update.

**WARNING**

If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

# Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Azure IoT Edge gateway to the remote monitoring preconfigured solution and send simulated telemetry

7/25/2017 • 8 min to read • Edit Online

This tutorial shows you how to use Azure IoT Edge to simulate temperature and humidity data to send to the remote monitoring preconfigured solution. The tutorial uses:

- Azure IoT Edge to implement a sample gateway.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your Intel NUC gateway device to communicate with your computer and the remote monitoring solution.
- Configure the IoT Edge gateway to send simulated telemetry that you can view on the solution dashboard.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Intel NUC.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Intel NUC.

IoT Commercial Gateway Kit. This tutorial uses the following items from the kit:

- Intel® NUC Kit DE3815TYKE with 4G Memory and Bluetooth expansion card
- Power adaptor

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.

3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

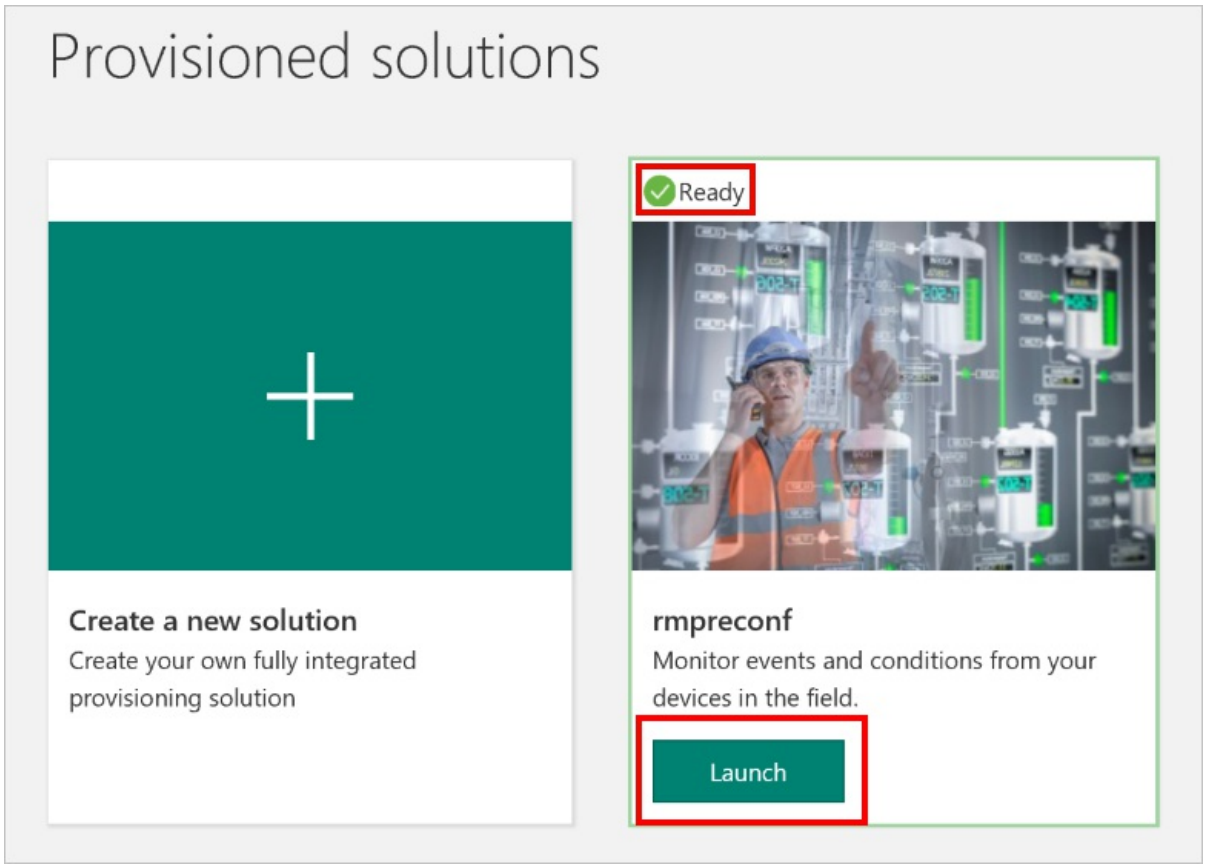
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

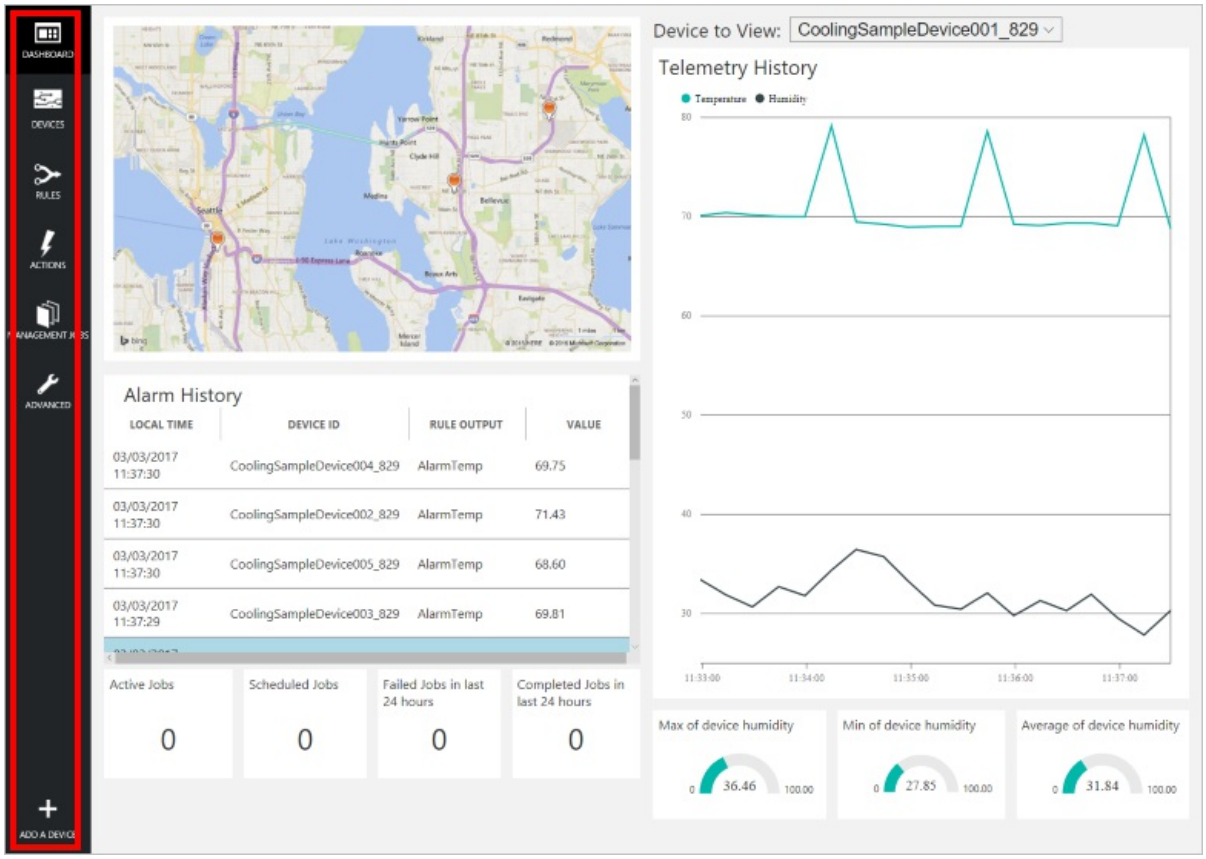
## View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. Navigate to other areas of the solution portal using the menu on the left-hand side of the page.

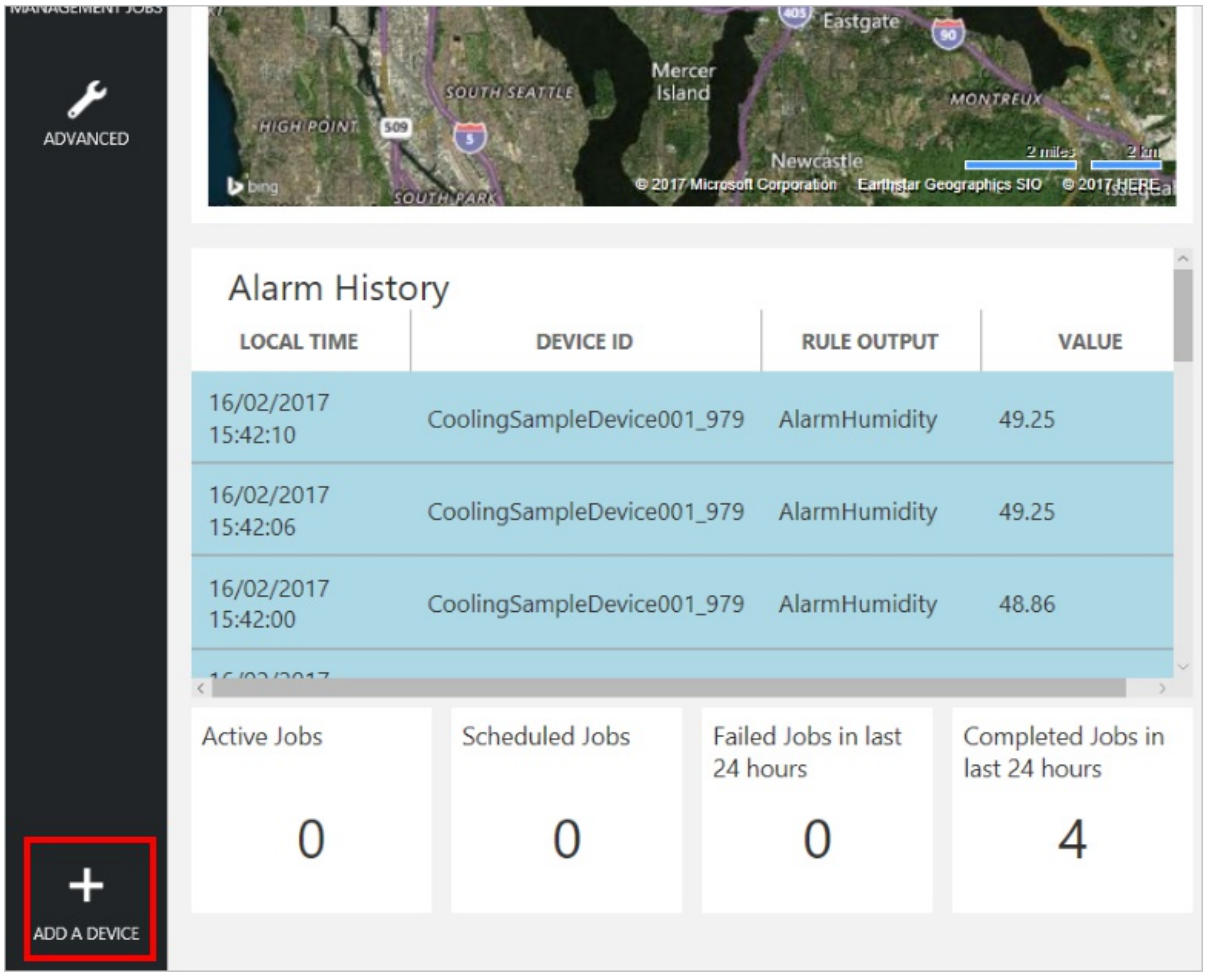



## Add a device

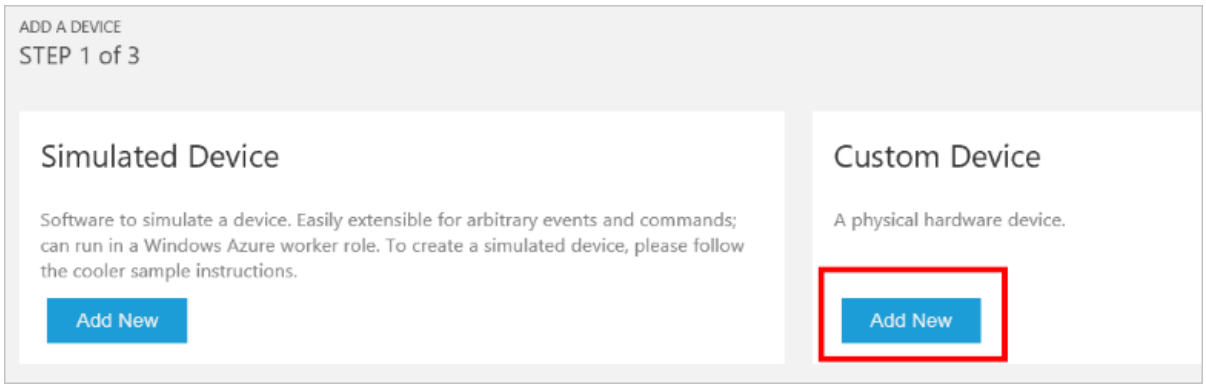
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

To add a device to your remote monitoring solution, complete the following steps in the solution dashboard:

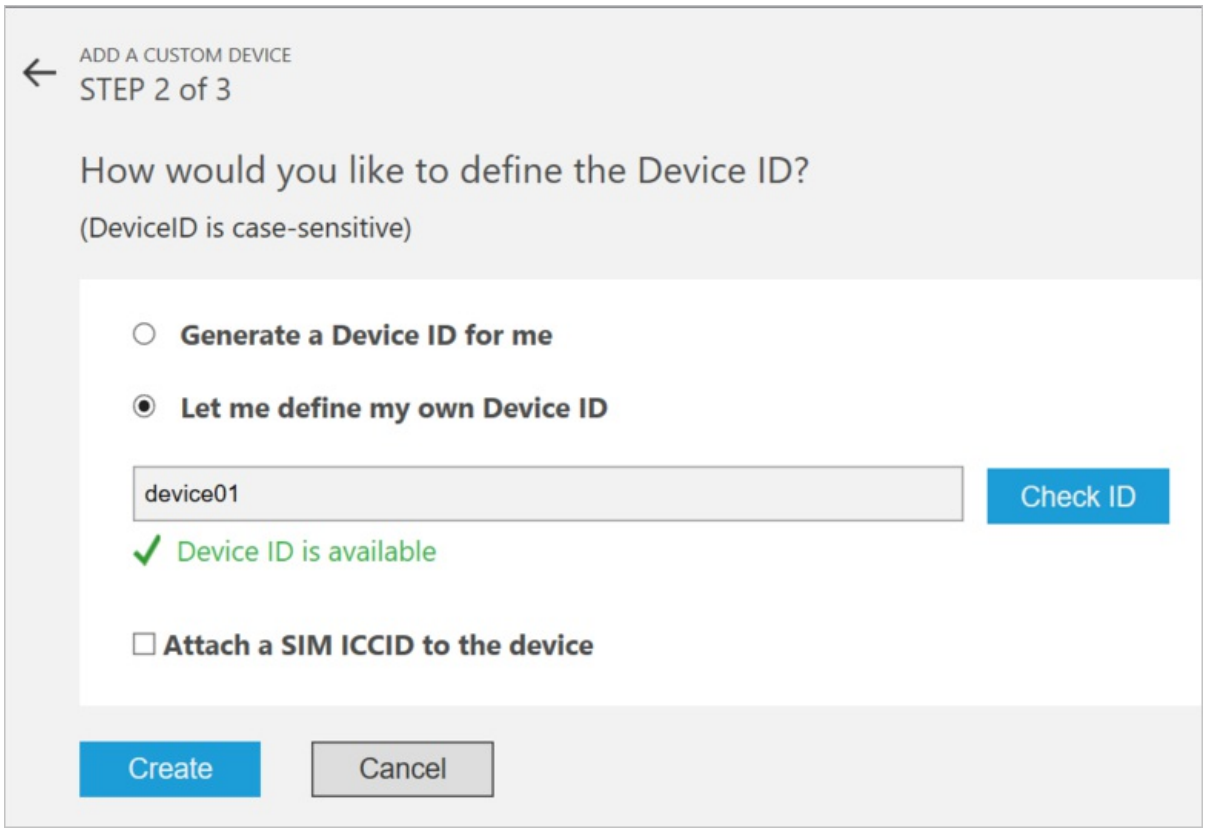
1. In the lower left-hand corner of the dashboard, click **Add a device**.



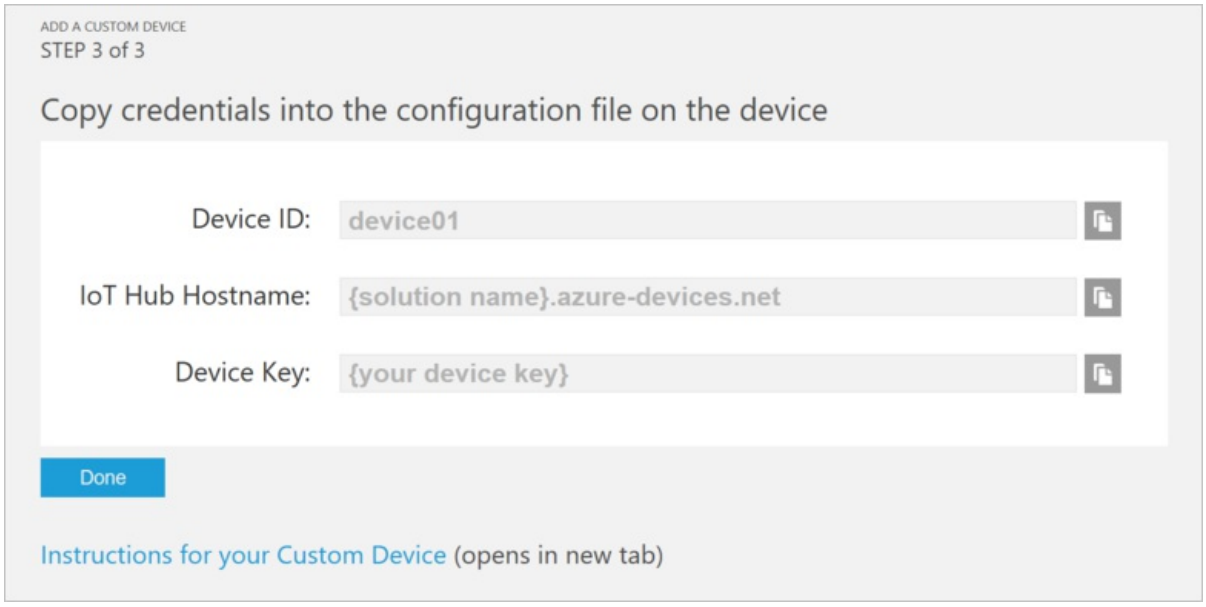

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **device01**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Intel NUC needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

Repeat the previous steps to add a second device using a Device ID such as **device02**. The sample sends data from two simulated devices in the gateway to the remote monitoring solution.

## Prepare your Intel NUC

To complete the hardware setup, you need to:

- Connect your Intel NUC to the power supply included in the kit.

- Connect your Intel NUC to your network using an Ethernet cable.

You have now completed the hardware setup of your Intel NUC gateway device.

### Sign in and access the terminal

You have two options to access a terminal environment on your Intel NUC:

- If you have a keyboard and monitor connected to your Intel NUC, you can access the shell directly. The default credentials are username **root** and password **root**.
- Access the shell on your Intel NUC using SSH from your desktop machine.

#### Sign in with SSH

To sign in with SSH, you need the IP address of your Intel NUC. If you have a keyboard and monitor connected to your Intel NUC, use the `ifconfig` command to find the IP address. Alternatively, connect to your router to list the addresses of devices on your network.

Sign in with username **root** and password **root**.

#### Optional: Share a folder on your Intel NUC

Optionally, you may want to share a folder on your Intel NUC with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Intel NUC instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Intel NUC. Alternatively, use the SFTP server on the Intel NUC with an SFTP client on your desktop machine.

## Build IoT Edge

This tutorial uses custom IoT Edge modules to communicate with the remote monitoring preconfigured solution. Therefore, you need to build the IoT Edge modules from custom source code. The following sections describe how to install IoT Edge and build the custom IoT Edge module.

### Install IoT Edge

The following steps describe how to install the pre-compiled IoT Edge software on the Intel NUC:

1. Configure the required smart package repositories by running the following commands on the Intel NUC:

   ```
   smart channel --add IoT_Cloud type=rpm-md name="IoT_Cloud" baseurl=http://iotdk.intel.com/repos/iot-cloud/wrlinux7/rcpl13/ -y
   smart channel --add WR_Repo type=rpm-md baseurl=https://distro.windriver.com/release/idp-3-xt/public_feeds/WR-IDP-3-XT-Intel-Baytrail-public-repo/RCPL13/corei7_64/
   ```

   Enter `y` when the command prompts you to **Include this channel?**.

2. Update the smart package manager by running the following command:

   ```
   smart update
   ```

3. Install the Azure IoT Edge package by running the following command:

   ```
   smart config --set rpm-check-signatures=false
   smart install packagegroup-cloud-azure -y
   ```

4. Verify the installation by running the "Hello world" sample. This sample writes a hello world message to the log.txT file every five seconds. The following commands run the "Hello world" sample:

```
cd /usr/share/azureiotgatewaysdk/samples/hello_world/
./hello_world hello_world.json
```

Ignore any **invalid argument** messages when you stop the sample.

Use the following command to view the contents of the log file:

```
cat log.txt | more
```

**Troubleshooting**

If you receive the error "No package provides util-linux-dev", try rebooting the Intel NUC.

# Build the custom IoT Edge module

You can now build the custom IoT Edge module that enables the gateway to send messages to the remote monitoring solution. For more information about configuring a gateway and IoT Edge modules, see Azure IoT Edge concepts.

Download the source code for the custom IoT Edge modules from GitHub using the following commands:

```
cd ~
git clone https://github.com/Azure-Samples/iot-remote-monitoring-c-intel-nuc-gateway-getting-started.git
```

Build the custom IoT Edge module using the following commands:

```
cd ~/iot-remote-monitoring-c-intel-nuc-gateway-getting-started/simulator
chmod u+x build.sh
sed -i -e 's/\r$//' build.sh
./build.sh
```

The build script places the libsimulator.so custom IoT Edge module in the build folder.

# Configure and run the IoT Edge gateway

You can now configure the IoT Edge gateway to send simulated telemetry to your remote monitoring dashboard. For more information about configuring a gateway and IoT Edge modules, see Azure IoT Edge concepts.

> **TIP**
> In this tutorial, you use the standard `vi` text editor on the Intel NUC. If you have not used `vi` before, you should complete an introductory tutorial, such as Unix - The vi Editor Tutorial to familiarize yourself with this editor. Alternatively, you can install the more user-friendly nano editor using the command `smart install nano -y`.

Open the sample configuration file in the **vi** editor using the following command:

```
vi ~/iot-remote-monitoring-c-intel-nuc-gateway-getting-started/simulator/remote_monitoring.json
```

Locate the following lines in the configuration for the IoTHub module:

```
"args": {
  "IoTHubName": "<<Azure IoT Hub Name>>",
  "IoTHubSuffix": "<<Azure IoT Hub Suffix>>",
  "Transport": "http"
}
```

Replace the placeholder values with the IoT Hub information you created and saved at the start of this tutorial. The value for IoTHubName looks like **yourrmsolution37e08**, and the value for IoTSuffix is typically **azure-devices.net**.

Locate the following lines in the configuration for the mapping module:

```
args": [
  {
    "macAddress": "AA:BB:CC:DD:EE:FF",
    "deviceId": "<<Azure IoT Hub Device ID>>",
    "deviceKey": "<<Azure IoT Hub Device Key>>"
  },
  {
    "macAddress": "AA:BB:CC:DD:EE:FF",
    "deviceId": "<<Azure IoT Hub Device ID>>",
    "deviceKey": "<<Azure IoT Hub Device Key>>"
  }
]
```

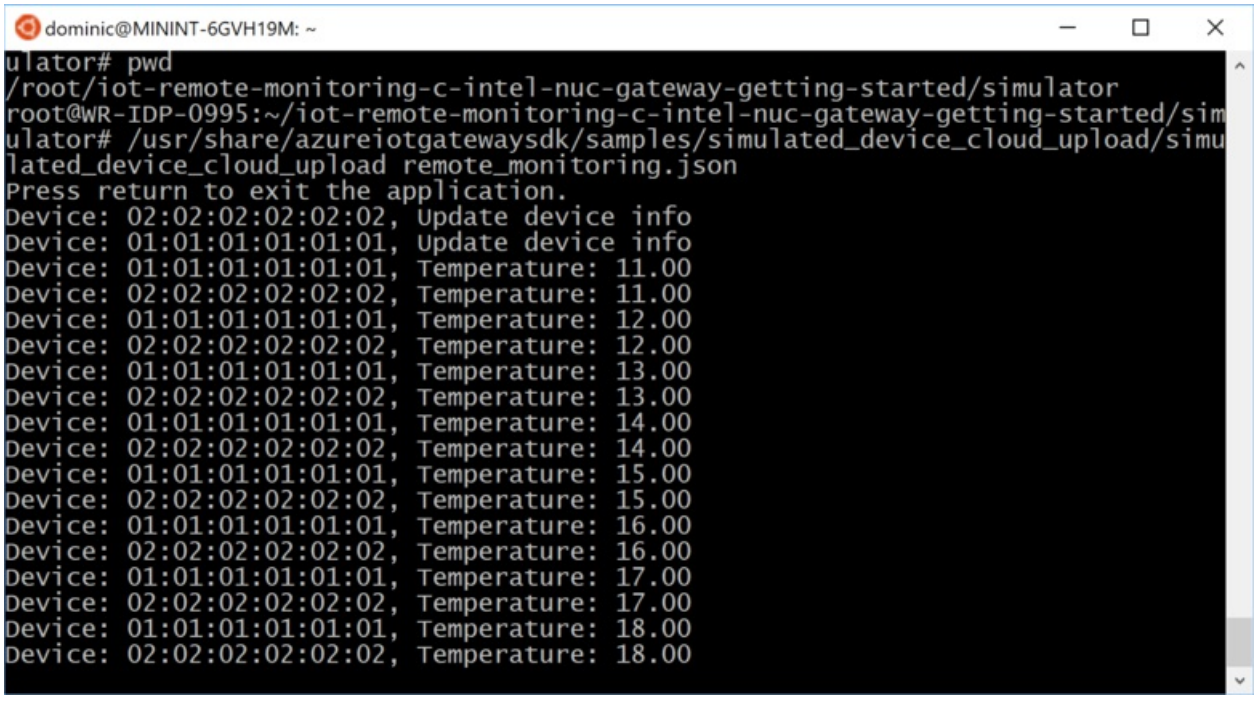
Replace the **deviceID** and **deviceKey** placeholders with the IDs and keys for the two devices you created in the remote monitoring solution previously.

Save your changes.

You can now run the IoT Edge gateway using the following commands:

```
cd ~/iot-remote-monitoring-c-intel-nuc-gateway-getting-started/simulator
/usr/share/azureiotgatewaysdk/samples/simulated_device_cloud_upload/simulated_device_cloud_upload
remote_monitoring.json
```

The gateway starts on the Intel NUC and sends simulated telemetry to the remote monitoring solution:

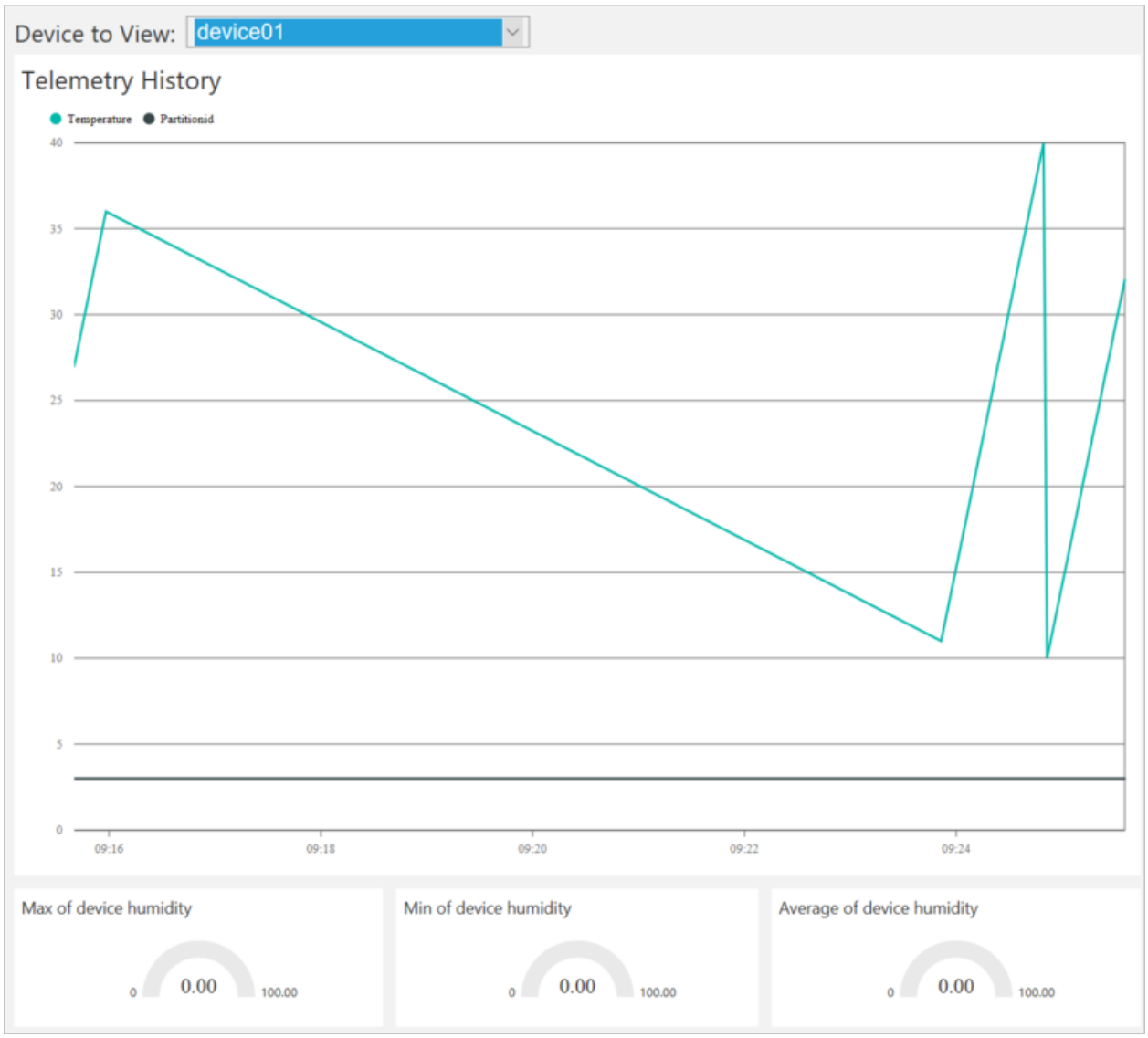


Press **Ctrl-C** to exit the program at any time.

## View the telemetry

The IoT Edge gateway is now sending simulated telemetry to the remote monitoring solution. You can view the telemetry on the solution dashboard.

- Navigate to the solution dashboard.
- Select one of the two devices you configured in the gateway in the **Device to View** dropdown.
- The telemetry from the gateway devices displays on the dashboard.




**WARNING**

If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your Azure IoT Edge gateway to the remote monitoring preconfigured solution and send telemetry from a SensorTag

7/25/2017 • 10 min to read • Edit Online

This tutorial shows you how to use Azure IoT Edge to send temperature and humidity data from SensorTag device to the remote monitoring preconfigured solution. The SensorTag connects to the Intel NUC gateway using Bluetooth. The tutorial uses:

- Azure IoT Edge to implement a sample gateway.
- The IoT Suite remote monitoring preconfigured solution as the cloud-based back end.

## Overview

In this tutorial, you complete the following steps:

- Deploy an instance of the remote monitoring preconfigured solution to your Azure subscription. This step automatically deploys and configures multiple Azure services.
- Set up your Intel NUC gateway device to communicate with your computer and the remote monitoring solution.
- Set up your Intel NUC gateway to receive telemetry from a SensorTag device and send it to the remote monitoring dashboard.

## Prerequisites

To complete this tutorial, you need an active Azure subscription.

> **NOTE**
> If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

**Required software**

You need SSH client on your desktop machine to enable you to remotely access the command line on the Intel NUC.

- Windows does not include an SSH client. We recommend using PuTTY.
- Most Linux distributions and Mac OS include the command-line SSH utility.

**Required hardware**

A desktop computer to enable you to connect remotely to the command line on the Intel NUC.

IoT Commercial Gateway Kit. This tutorial uses the following items from the kit:

- Intel® NUC Kit DE3815TYKE with 4G Memory and Bluetooth expansion card
- Power adaptor

Texas Instruments BLE SensorTag. This tutorial retrieves telemetry data over Bluetooth from the SensorTag device.

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

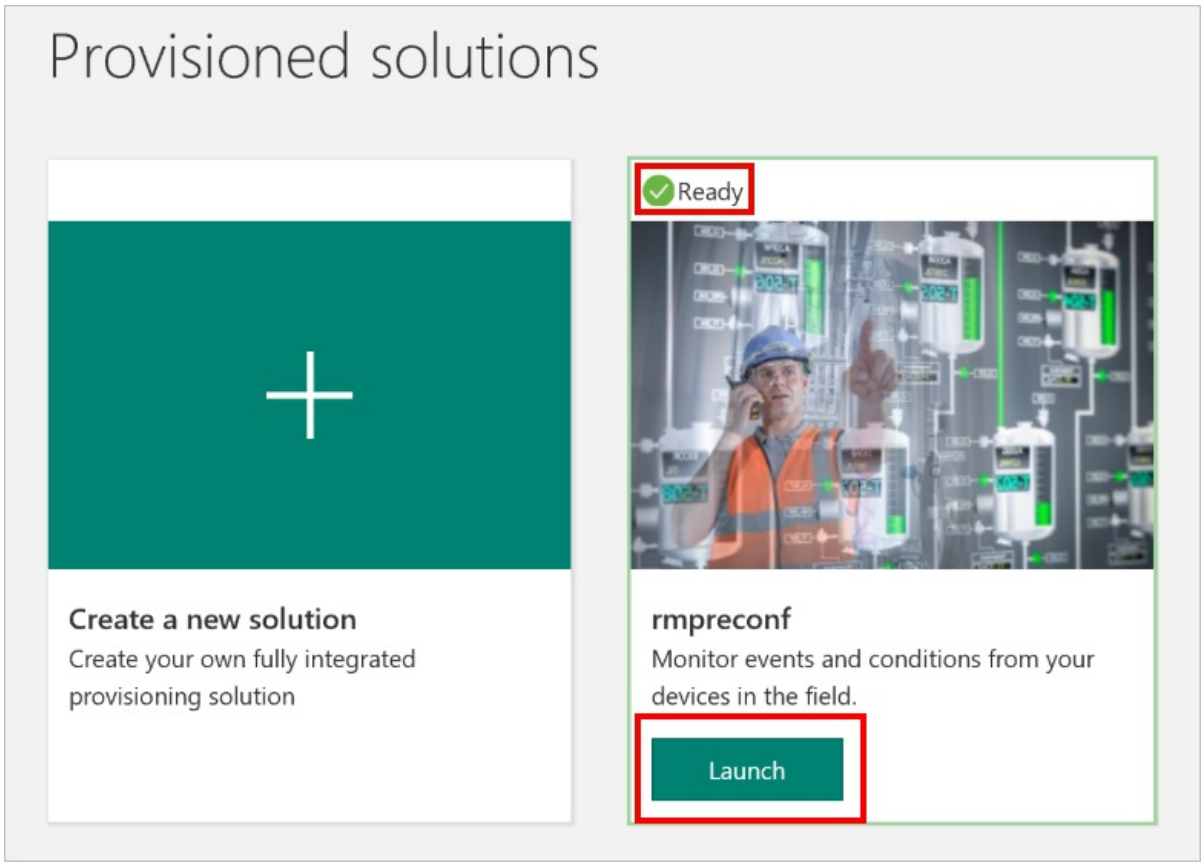
Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

> **WARNING**
> The remote monitoring solution provisions a set of Azure services in your Azure subscription. The deployment reflects a real enterprise architecture. To avoid unnecessary Azure consumption charges, delete your instance of the preconfigured solution at azureiotsuite.com when you have finished with it. If you need the preconfigured solution again, you can easily recreate it. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes.

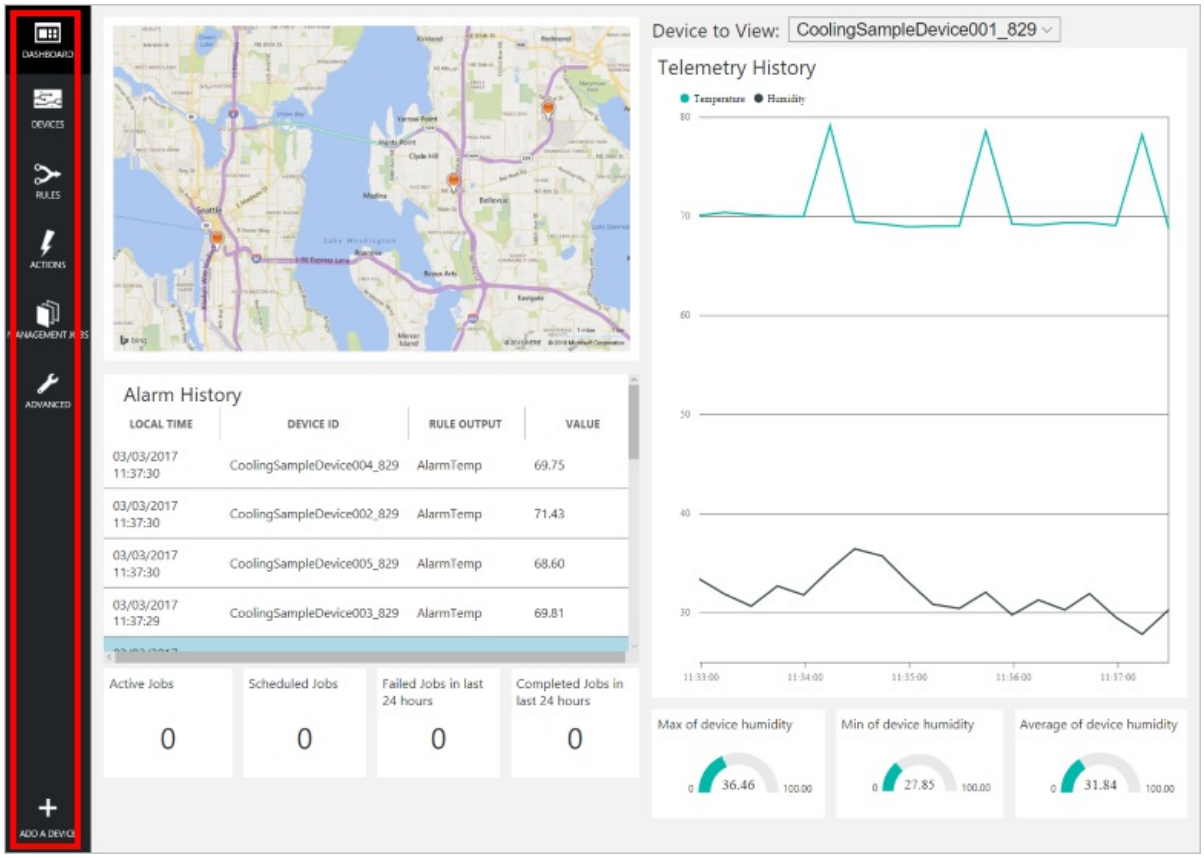
# View the solution dashboard

The solution dashboard enables you to manage the deployed solution. For example, you can view telemetry, add devices, and invoke methods.

1. When the provisioning is complete and the tile for your preconfigured solution indicates **Ready**, choose **Launch** to open your remote monitoring solution portal in a new tab.




2. By default, the solution portal shows the *dashboard*. Navigate to other areas of the solution portal using the menu on the left-hand side of the page.

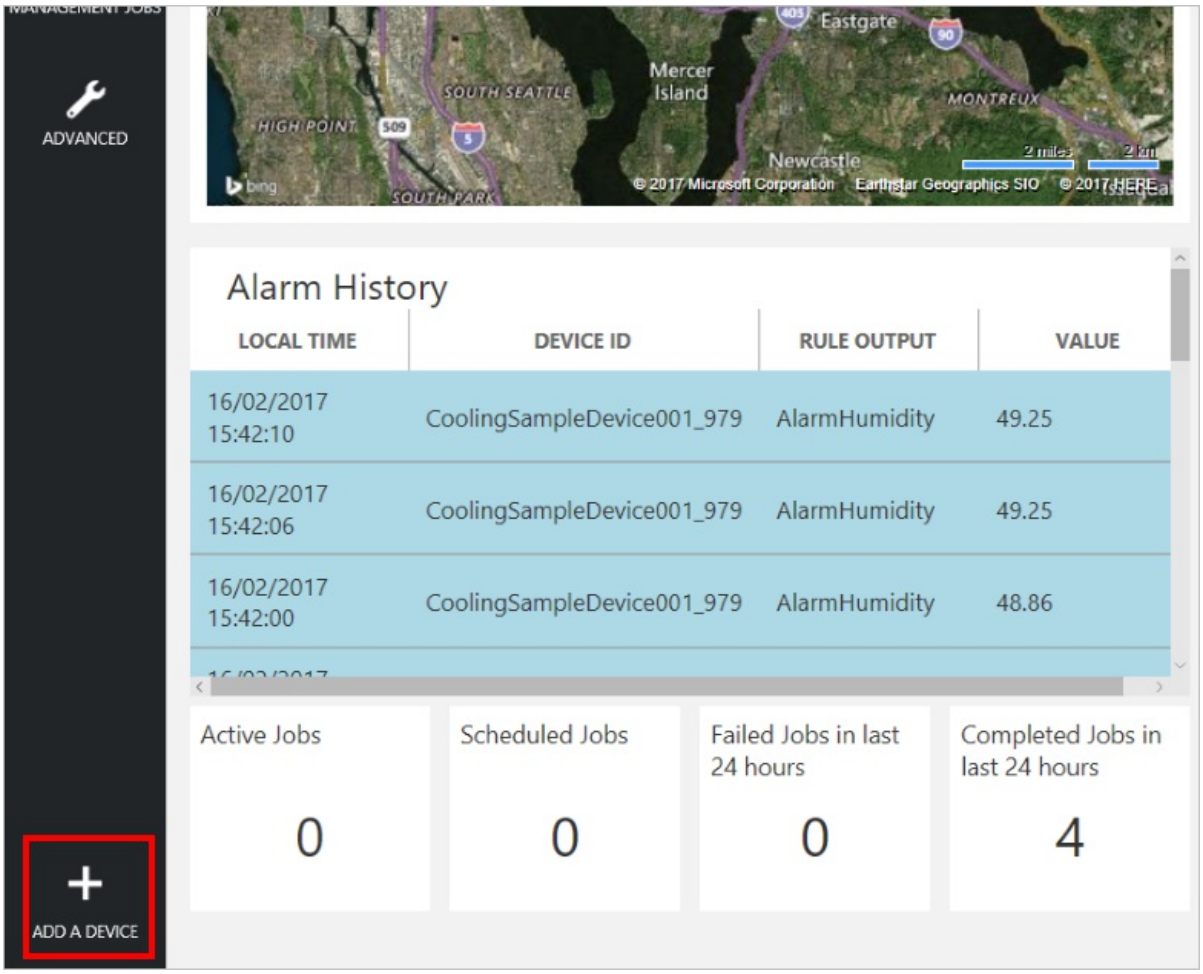



## Add a device

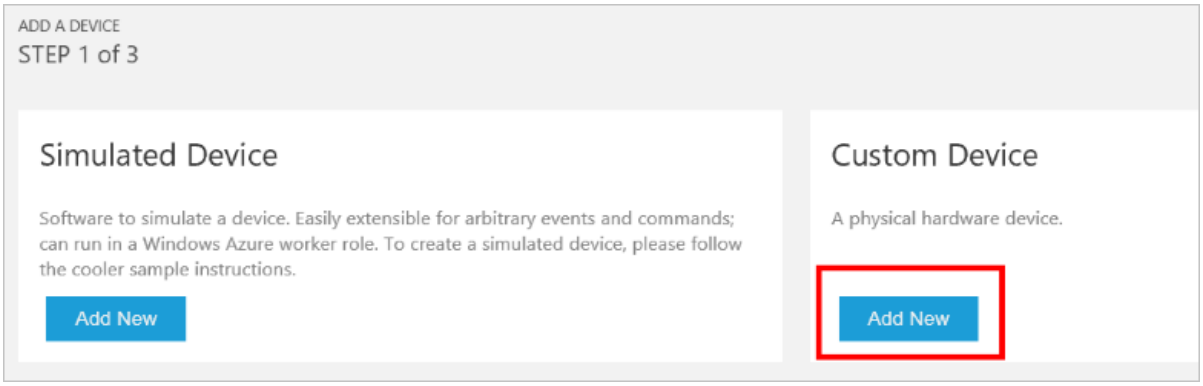
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

To add a device to your remote monitoring solution, complete the following steps in the solution dashboard:

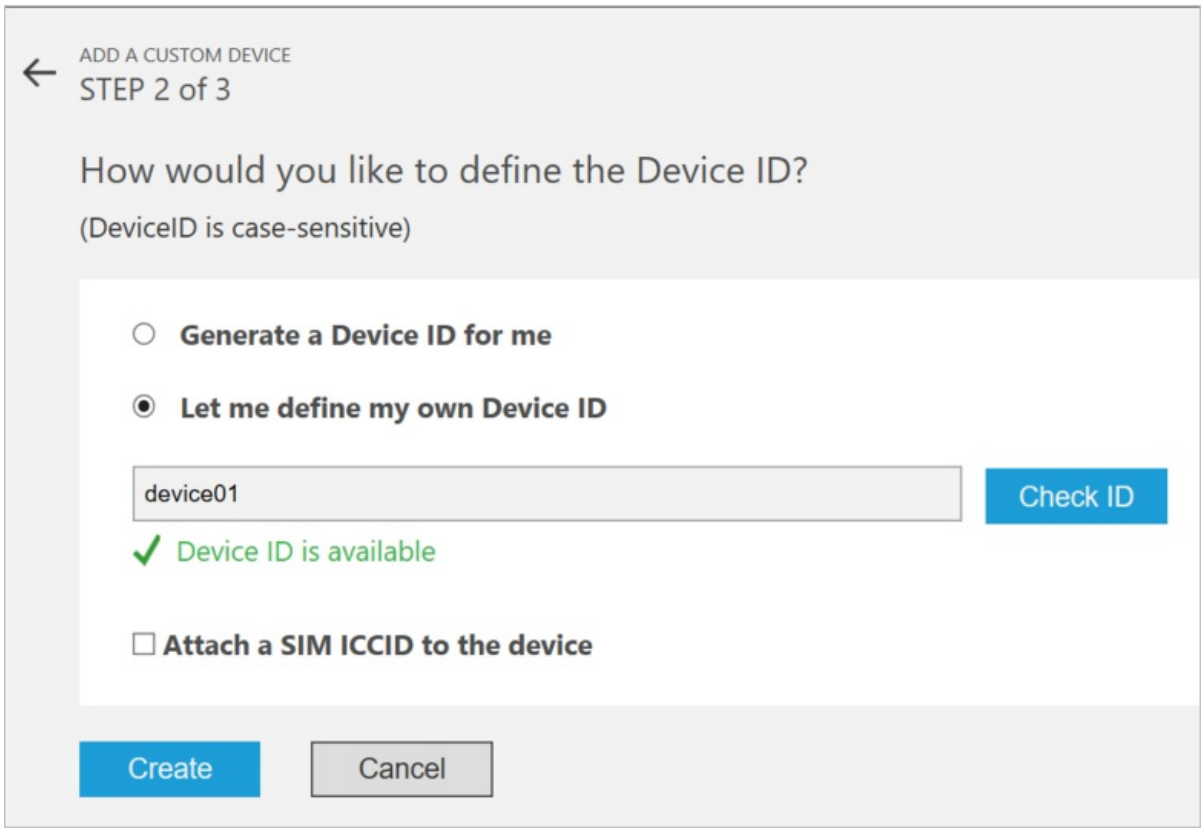
1. In the lower left-hand corner of the dashboard, click **Add a device**.



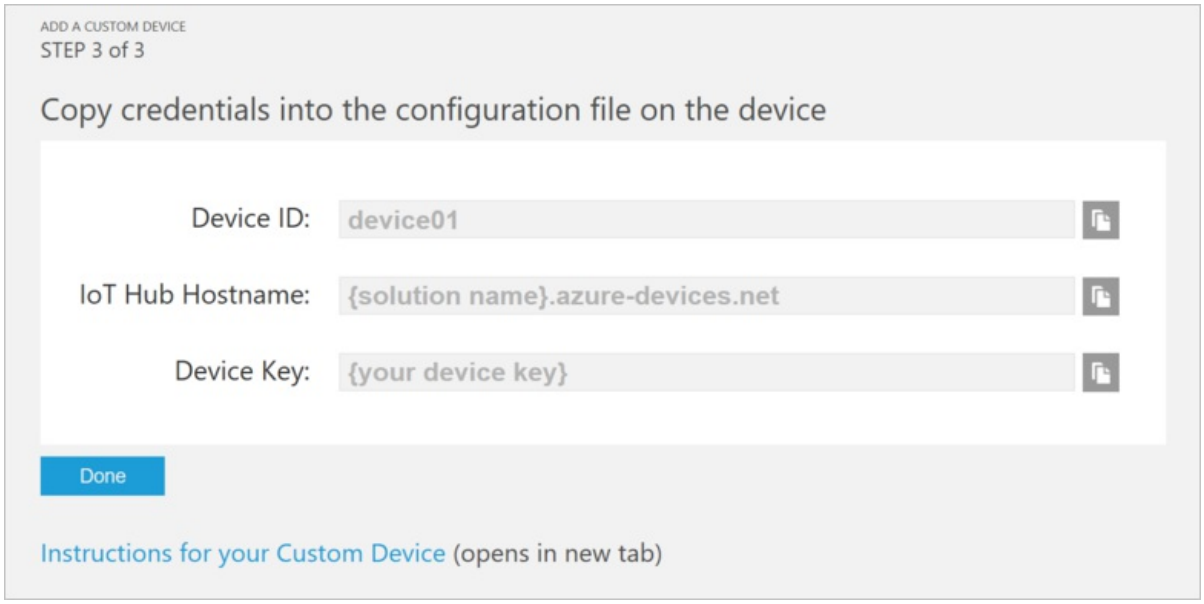

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **device01**, click **Check ID** to verify you haven't already used the name in your solution, and then click **Create** to provision the device.




4. Make a note the device credentials (**Device ID**, **IoT Hub Hostname**, and **Device Key**). Your client application on the Intel NUC needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Prepare your Intel NUC

To complete the hardware setup, you need to:

- Connect your Intel NUC to the power supply included in the kit.
- Connect your Intel NUC to your network using an Ethernet cable.

You have now completed the hardware setup of your Intel NUC gateway device.

### Sign in and access the terminal

You have two options to access a terminal environment on your Intel NUC:

- If you have a keyboard and monitor connected to your Intel NUC, you can access the shell directly. The default credentials are username **root** and password **root**.
- Access the shell on your Intel NUC using SSH from your desktop machine.

#### Sign in with SSH

To sign in with SSH, you need the IP address of your Intel NUC. If you have a keyboard and monitor connected to your Intel NUC, use the `ifconfig` command to find the IP address. Alternatively, connect to your router to list the addresses of devices on your network.

Sign in with username **root** and password **root**.

#### Optional: Share a folder on your Intel NUC

Optionally, you may want to share a folder on your Intel NUC with your desktop environment. Sharing a folder enables you to use your preferred desktop text editor (such as Visual Studio Code or Sublime Text) to edit files on your Intel NUC instead of using `nano` or `vi`.

To share a folder with Windows, configure a Samba server on the Intel NUC. Alternatively, use the SFTP server on the Intel NUC with an SFTP client on your desktop machine.

## Configure Bluetooth connectivity

Configure Bluetooth on the Intel NUC to enable the SensorTag device to connect and send telemetry.

### Find the MAC address of the SensorTag

1. In the shell on the Intel NUC, run the following command to unblock the Bluetooth service:

   ```
   sudo rfkill unblock bluetooth
   ```

2. Run the following commands to start the Bluetooth service on the Intel NUC and enter the Bluetooth shell:

   ```
   sudo systemctl start bluetooth
   bluetoothctl
   ```

3. Run the following command to power on the Bluetooth controller:

   ```
   power on
   ```

   When the controller is on, you see a message **Changing power on succeeded**.

4. Run the following command to scan for nearby Bluetooth devices:

   ```
   scan on
   ```

5. Press the power button on the SensorTag to make it discoverable. The green LED flashes.

6. When you see a message in the shell that the controller has discovered the SensorTag, make a note of the MAC address of the device. The MAC address looks like **A0:E6:F8:B5:F6:00**. You need the MAC address later in the tutorial when you configure the gateway.

7. Run the following command to turn off Bluetooth scanning:

```
scan off
```

8. Run the following command to verify that you can connect to the SensorTag device:

   ```
   connect <SensorTag MAC address>
   ```

   If you connect successfully, the shell shows the message **Connection successful** and prints information about the SensorTag device. If you cannot connect, check the SensorTag is still powered on.

9. You can now disconnect from the SensorTag and exit the Bluetooth shell by running the following commands:

   ```
   disconnect
   exit
   ```

# Build IoT Edge

This tutorial uses custom IoT Edge modules to communicate with the remote monitoring preconfigured solution. Therefore, you need to build the IoT Edge modules from custom source code. The following sections describe how to install IoT Edge and build the custom IoT Edge module.

### Install IoT Edge

The following steps describe how to install the pre-compiled IoT Edge software on the Intel NUC:

1. Configure the required smart package repositories by running the following commands on the Intel NUC:

   ```
   smart channel --add IoT_Cloud type=rpm-md name="IoT_Cloud" baseurl=http://iotdk.intel.com/repos/iot-cloud/wrlinux7/rcpl13/ -y
   smart channel --add WR_Repo type=rpm-md baseurl=https://distro.windriver.com/release/idp-3-xt/public_feeds/WR-IDP-3-XT-Intel-Baytrail-public-repo/RCPL13/corei7_64/
   ```

   Enter `y` when the command prompts you to **Include this channel?**.

2. Update the smart package manager by running the following command:

   ```
   smart update
   ```

3. Install the Azure IoT Edge package by running the following command:

   ```
   smart config --set rpm-check-signatures=false
   smart install packagegroup-cloud-azure -y
   ```

4. Verify the installation by running the "Hello world" sample. This sample writes a hello world message to the log.txT file every five seconds. The following commands run the "Hello world" sample:

   ```
   cd /usr/share/azureiotgatewaysdk/samples/hello_world/
   ./hello_world hello_world.json
   ```

   Ignore any **invalid argument** messages when you stop the sample.

   Use the following command to view the contents of the log file:

```
cat log.txt | more
```

**Troubleshooting**

If you receive the error "No package provides util-linux-dev", try rebooting the Intel NUC.

## Build the custom IoT Edge module

You can now build the custom IoT Edge module that enables the gateway to send messages to the remote monitoring solution. For more information about configuring a gateway and IoT Edge modules, see Azure IoT Edge concepts.

Download the source code for the custom IoT Edge modules from GitHub using the following commands:

```
cd ~
git clone https://github.com/Azure-Samples/iot-remote-monitoring-c-intel-nuc-gateway-getting-started.git
```

Build the custom IoT Edge module using the following commands:

```
cd ~/iot-remote-monitoring-c-intel-nuc-gateway-getting-started/basic
chmod u+x build.sh
sed -i -e 's/\r$//' build.sh
./build.sh
```

The build script places the libsensor2remotemonitoring.so custom IoT Edge module in the build folder.

## Configure and run the IoT Edge gateway

You can now configure the IoT Edge gateway to send telemetry from your SensorTag device to your remote monitoring dashboard. For more information about configuring a gateway and IoT Edge modules, see Azure IoT Edge concepts.

> **TIP**
> In this tutorial, you use the standard `vi` text editor on the Intel NUC. If you have not used `vi` before, you should complete an introductory tutorial, such as Unix - The vi Editor Tutorial to familiarize yourself with this editor. Alternatively, you can install the more user-friendly nano editor using the command `smart install nano -y`.

Open the sample configuration file in the **vi** editor using the following command:

```
vi ~/iot-remote-monitoring-c-intel-nuc-gateway-getting-started/basic/remote_monitoring.json
```

Locate the following lines in the configuration for the IoTHub module:

```
"args": {
  "IoTHubName": "<<Azure IoT Hub Name>>",
  "IoTHubSuffix": "<<Azure IoT Hub Suffix>>",
  "Transport": "http"
}
```

Replace the placeholder values with the IoT Hub information you created and saved at the start of this tutorial. The value for IoTHubName looks like **yourrmsolution37e08**, and the value for IoTSuffix is typically **azure-devices.net**.

Locate the following lines in the configuration for the mapping module:

```
args": [
  {
    "macAddress": "<<AA:BB:CC:DD:EE:FF>>",
    "deviceId": "<<Azure IoT Hub Device ID>>",
    "deviceKey": "<<Azure IoT Hub Device Key>>"
  }
]
```

Replace the **macAddress** placeholder with the MAC address of your SensorTag you noted previously. Replace the **deviceID** and **deviceKey** placeholders with the IDs and keys for the two devices you created in the remote monitoring solution previously.

Locate the following lines in the configuration for the SensorTag module:

```
"args": {
  "controller_index": 0,
  "device_mac_address": "<<AA:BB:CC:DD:EE:FF>>",
  ...
}
```

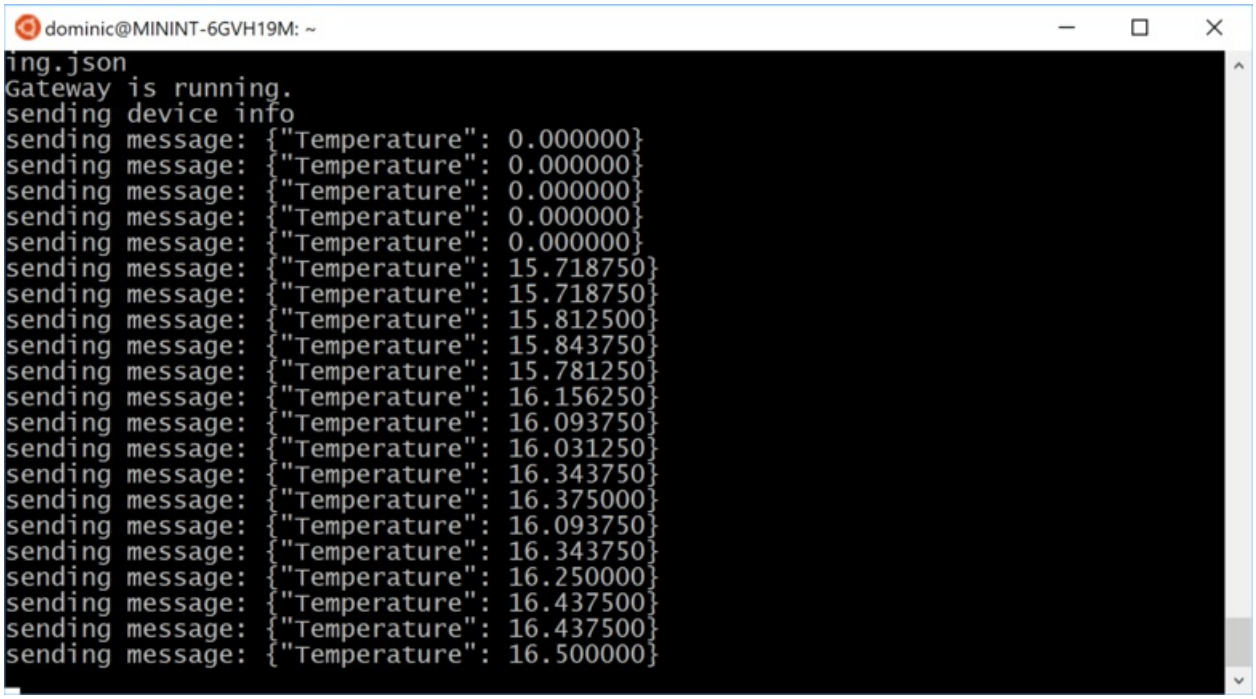
Replace the **device_mac_address** placeholder with the MAC address of your SensorTag you noted previously.

Save your changes.

You can now run the gateway using the following commands:

```
cd ~/iot-remote-monitoring-c-intel-nuc-gateway-getting-started/basic
/usr/share/azureiotgatewaysdk/samples/ble_gateway/ble_gateway remote_monitoring.json
```

The IoT Edge gateway starts on the Intel NUC and sends telemetry from the SensorTag to the remote monitoring solution:

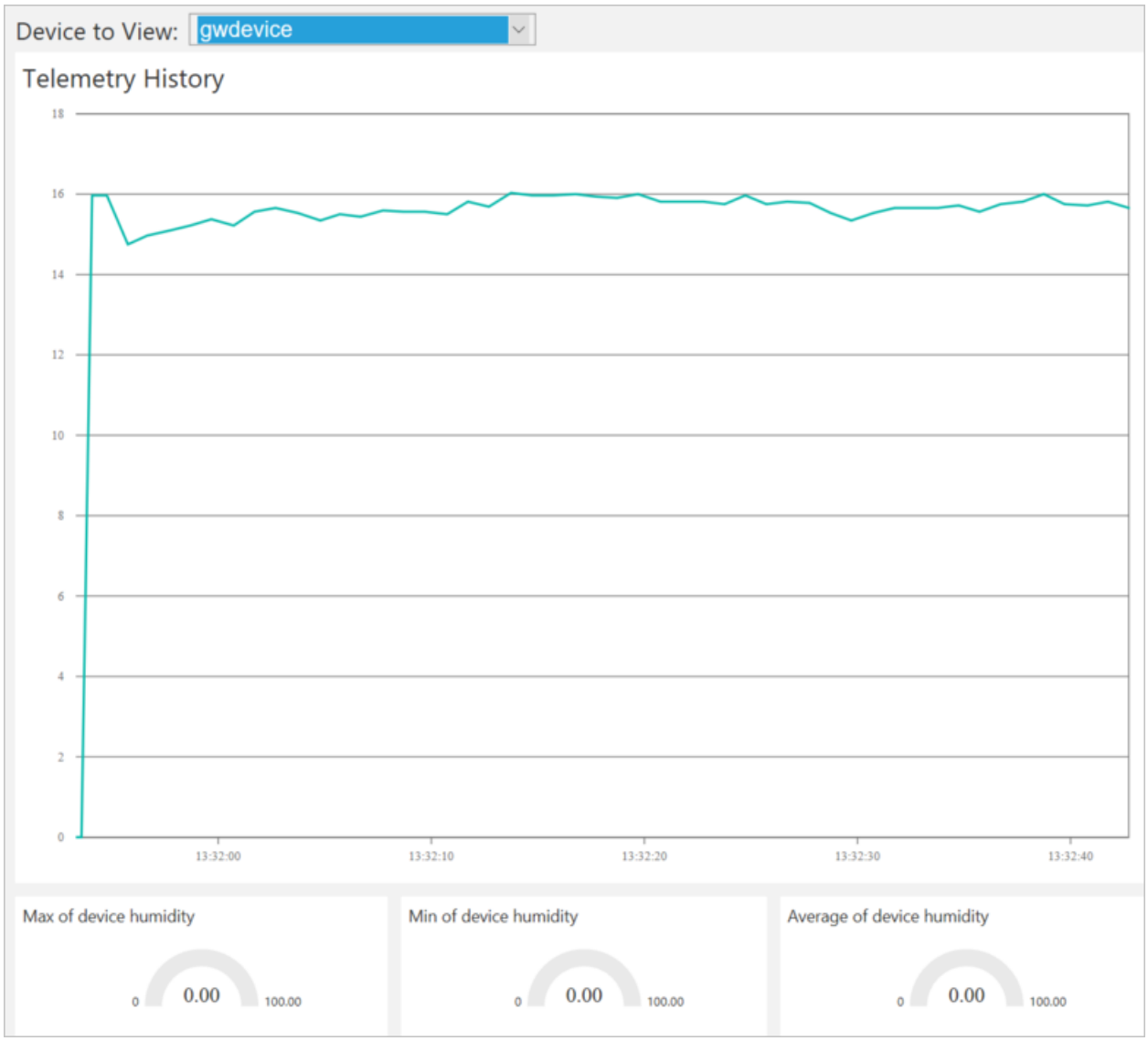



Press **Ctrl-C** to exit the program at any time.

# View the telemetry

The gateway is now sending telemetry from the SensorTag device to the remote monitoring solution. You can view the telemetry on the solution dashboard. You can also send commands to your SensorTag device through the gateway from the solution dashboard.

- Navigate to the solution dashboard.
- Select the device you configured in the gateway that represents the SensorTag in the **Device to View** dropdown.
- The telemetry from the SensorTag device displays on the dashboard.




**WARNING**

If you leave the remote monitoring solution running in your Azure account, you are billed for the time it runs. For more information about reducing consumption while the remote monitoring solution runs, see Configuring Azure IoT Suite preconfigured solutions for demo purposes. Delete the preconfigured solution from your Azure account when you have finished using it.

## Next steps

Visit the Azure IoT Dev Center for more samples and documentation on Azure IoT.

# Connect your device to the remote monitoring preconfigured solution (Windows)

8/24/2017 • 11 min to read • Edit Online

## Scenario overview

In this scenario, you create a device that sends the following telemetry to the remote monitoring preconfigured solution:

- External temperature
- Internal temperature
- Humidity

For simplicity, the code on the device generates sample values, but we encourage you to extend the sample by connecting real sensors to your device and sending real telemetry.

The device is also able to respond to methods invoked from the solution dashboard and desired property values set in the solution dashboard.

To complete this tutorial, you need an active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

## Before you start

Before you write any code for your device, you must provision your remote monitoring preconfigured solution and provision a new custom device in that solution.

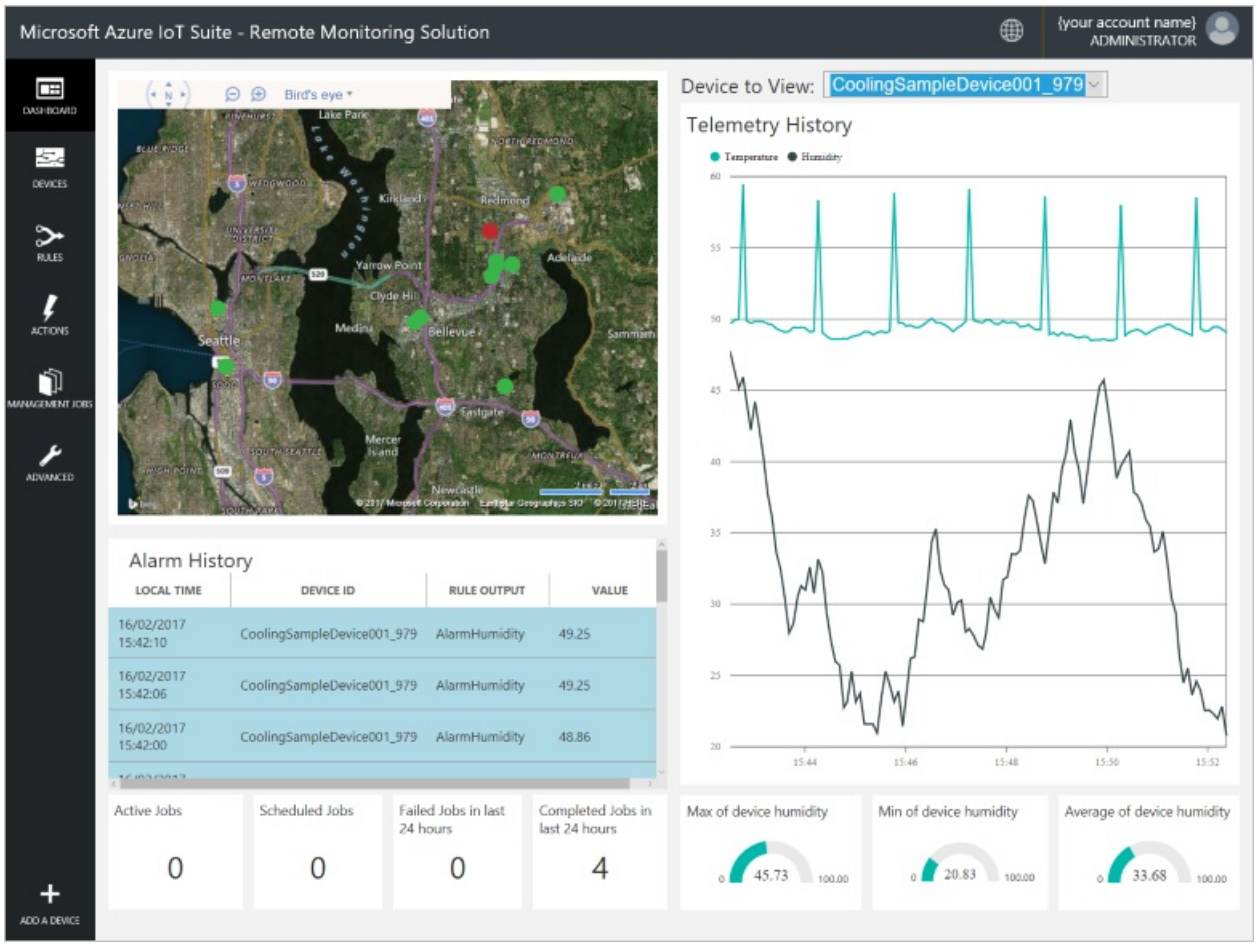
**Provision your remote monitoring preconfigured solution**

The device you create in this tutorial sends data to an instance of the remote monitoring preconfigured solution. If you haven't already provisioned the remote monitoring preconfigured solution in your Azure account, use the following steps:

1. On the https://www.azureiotsuite.com/ page, click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** panel to create your solution.
3. On the **Create Remote monitoring solution** page, enter a **Solution name** of your choice, select the **Region** you want to deploy to, and select the Azure subscription to want to use. Then click **Create solution**.
4. Wait until the provisioning process completes.

> **WARNING**
> The preconfigured solutions use billable Azure services. Be sure to remove the preconfigured solution from your subscription when you are done with it to avoid any unnecessary charges. You can completely remove a preconfigured solution from your subscription by visiting the https://www.azureiotsuite.com/ page.

When the provisioning process for the remote monitoring solution finishes, click **Launch** to open the solution dashboard in your browser.



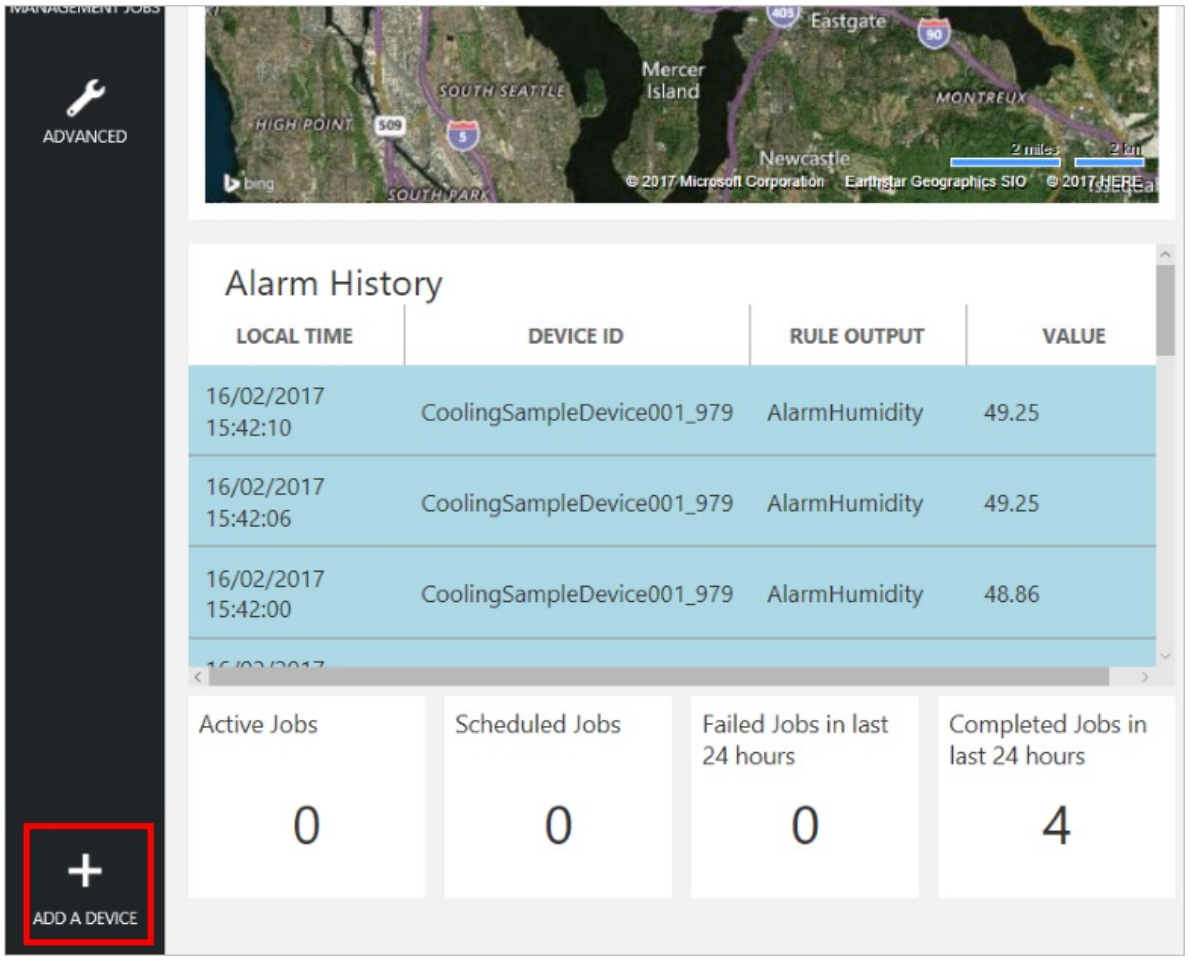

**Provision your device in the remote monitoring solution**

> **NOTE**
>
> If you have already provisioned a device in your solution, you can skip this step. You need to know the device credentials when you create the client application.

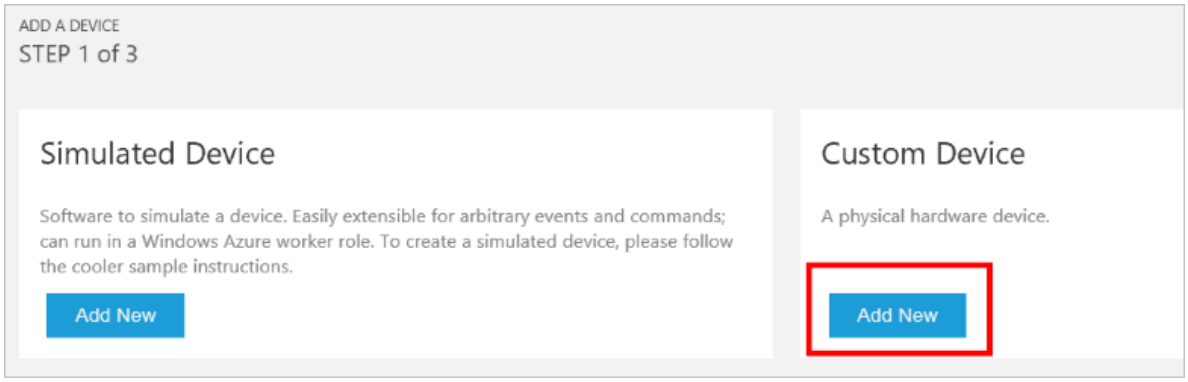
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

To add a device to your remote monitoring solution, complete the following steps in the solution dashboard:

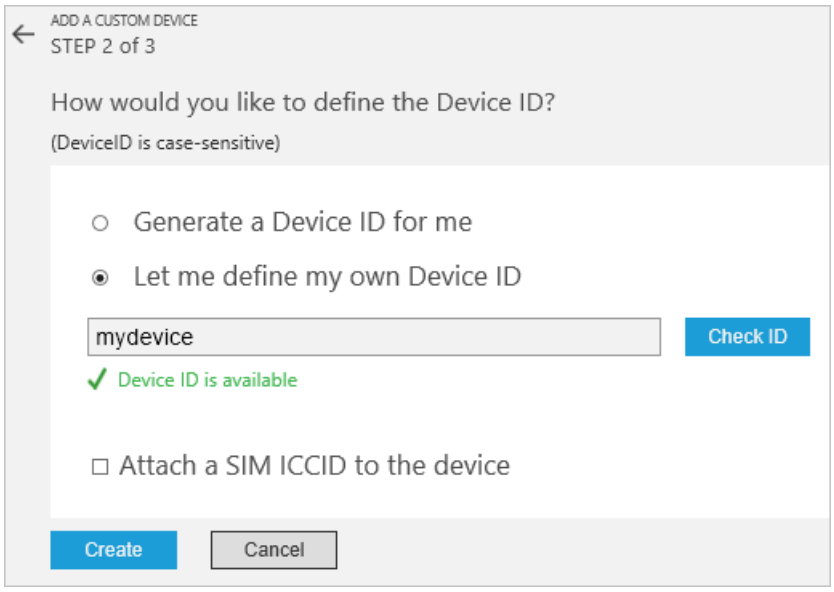
1. In the lower left-hand corner of the dashboard, click **Add a device**.



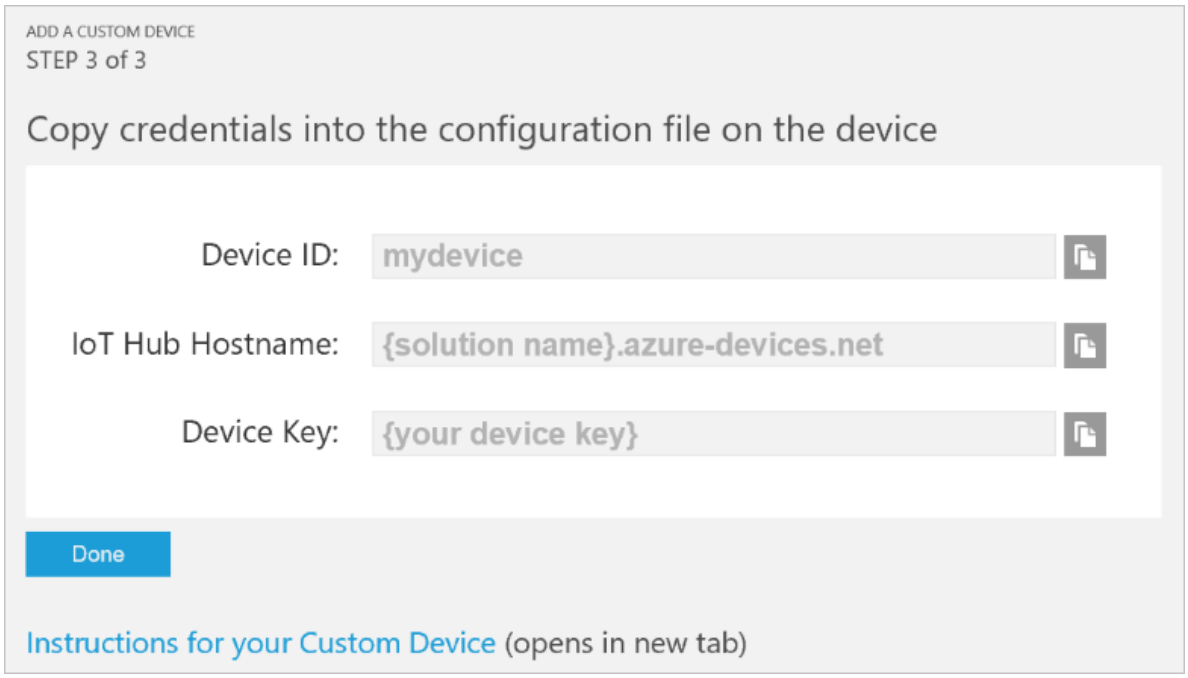

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **mydevice**, click **Check ID** to verify that name isn't already in use, and then click **Create** to provision the device.




4. Make a note the device credentials (Device ID, IoT Hub Hostname, and Device Key). Your client application needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Create a C sample solution on Windows

The following steps show you how to create a client application that communicates with the remote monitoring preconfigured solution. This application is written in C and built and run on Windows.

Create a starter project in Visual Studio 2015 or Visual Studio 2017 and add the IoT Hub device client NuGet packages:

1. In Visual Studio, create a C console application using the Visual C++ **Win32 Console Application** template. Name the project **RMDevice**.
2. On the **Application Settings** page in the **Win32 Application Wizard**, ensure that **Console application** is selected, and uncheck **Precompiled header** and **Security Development Lifecycle (SDL) checks**.
3. In **Solution Explorer**, delete the files stdafx.h, targetver.h, and stdafx.cpp.

4. In **Solution Explorer**, rename the file RMDevice.cpp to RMDevice.c.
5. In **Solution Explorer**, right-click on the **RMDevice** project and then click **Manage NuGet packages**. Click **Browse**, then search for and install the following NuGet packages:

   - Microsoft.Azure.IoTHub.Serializer
   - Microsoft.Azure.IoTHub.IoTHubClient
   - Microsoft.Azure.IoTHub.MqttTransport
6. In **Solution Explorer**, right-click on the **RMDevice** project and then click **Properties** to open the project's **Property Pages** dialog box. For details, see Setting Visual C++ Project Properties.
7. Click the **Linker** folder, then click the **Input** property page.
8. Add **crypt32.lib** to the **Additional Dependencies** property. Click **OK** and then **OK** again to save the project property values.

Add the Parson JSON library to the **RMDevice** project and add the required `#include` statements:

1. In a suitable folder on your computer, clone the Parson GitHub repository using the following command:

   ```
   git clone https://github.com/kgabis/parson.git
   ```

2. Copy the parson.h and parson.c files from the local copy of the Parson repository to your **RMDevice** project folder.

3. In Visual Studio, right-click the **RMDevice** project, click **Add**, and then click **Existing Item**.

4. In the **Add Existing Item** dialog, select the parson.h and parson.c files in the **RMDevice** project folder. Then click **Add** to add these two files to your project.

5. In Visual Studio, open the RMDevice.c file. Replace the existing `#include` statements with the following code:

   ```
   #include "iothubtransportmqtt.h"
   #include "schemalib.h"
   #include "iothub_client.h"
   #include "serializer_devicetwin.h"
   #include "schemaserializer.h"
   #include "azure_c_shared_utility/threadapi.h"
   #include "azure_c_shared_utility/platform.h"
   #include "parson.h"
   ```

   > **NOTE**
   > Now you can verify that your project has the correct dependencies set up by building it.

## Specify the behavior of the IoT device

The IoT Hub serializer client library uses a model to specify the format of the messages the device exchanges with IoT Hub.

1. Add the following variable declarations after the `#include` statements. Replace the placeholder values [Device Id] and [Device Key] with values you noted for your device in the remote monitoring solution dashboard. Use the IoT Hub Hostname from the solution dashboard to replace [IoTHub Name]. For example, if your IoT Hub Hostname is **contoso.azure-devices.net**, replace [IoTHub Name] with **contoso**:

```
static const char* deviceId = "[Device Id]";
static const char* connectionString = "HostName=[IoTHub Name].azure-devices.net;DeviceId=[Device
Id];SharedAccessKey=[Device Key]";
```

2. Add the following code to define the model that enables the device to communicate with IoT Hub. This model specifies that the device:

   - Can send temperature, external temperature, humidity, and a device id as telemetry.
   - Can send metadata about the device to IoT Hub. The device sends basic metadata in a **DeviceInfo** object at startup.
   - Can send reported properties, to the device twin in IoT Hub. These reported properties are grouped into configuration, device, and system properties.
   - Can receive and act on desired properties set in the device twin in IoT Hub.
   - Can respond to the **Reboot** and **InitiateFirmwareUpdate** direct methods invoked through the solution portal. The device sends information about the direct methods it supports using reported properties.

```
// Define the Model
BEGIN_NAMESPACE(Contoso);

/* Reported properties */
DECLARE_STRUCT(SystemProperties,
 ascii_char_ptr, Manufacturer,
 ascii_char_ptr, FirmwareVersion,
 ascii_char_ptr, InstalledRAM,
 ascii_char_ptr, ModelNumber,
 ascii_char_ptr, Platform,
 ascii_char_ptr, Processor,
 ascii_char_ptr, SerialNumber
);

DECLARE_STRUCT(LocationProperties,
 double, Latitude,
 double, Longitude
);

DECLARE_STRUCT(ReportedDeviceProperties,
 ascii_char_ptr, DeviceState,
 LocationProperties, Location
);

DECLARE_MODEL(ConfigProperties,
 WITH_REPORTED_PROPERTY(double, TemperatureMeanValue),
 WITH_REPORTED_PROPERTY(uint8_t, TelemetryInterval)
);

/* Part of DeviceInfo */
DECLARE_STRUCT(DeviceProperties,
 ascii_char_ptr, DeviceID,
 _Bool, HubEnabledState
);

DECLARE_DEVICETWIN_MODEL(Thermostat,
 /* Telemetry (temperature, external temperature and humidity) */
 WITH_DATA(double, Temperature),
 WITH_DATA(double, ExternalTemperature),
 WITH_DATA(double, Humidity),
 WITH_DATA(ascii_char_ptr, DeviceId),

 /* DeviceInfo */
 WITH_DATA(ascii_char_ptr, ObjectType),
 WITH_DATA(_Bool, IsSimulatedDevice),
 WITH_DATA(ascii_char_ptr, Version),
 WITH_DATA(DeviceProperties, DeviceProperties),

 /* Device twin properties */
 WITH_REPORTED_PROPERTY(ReportedDeviceProperties, Device),
 WITH_REPORTED_PROPERTY(ConfigProperties, Config),
 WITH_REPORTED_PROPERTY(SystemProperties, System),

 WITH_DESIRED_PROPERTY(double, TemperatureMeanValue, onDesiredTemperatureMeanValue),
 WITH_DESIRED_PROPERTY(uint8_t, TelemetryInterval, onDesiredTelemetryInterval),

 /* Direct methods implemented by the device */
 WITH_METHOD(Reboot),
 WITH_METHOD(InitiateFirmwareUpdate, ascii_char_ptr, FwPackageURI),

 /* Register direct methods with solution portal */
 WITH_REPORTED_PROPERTY(ascii_char_ptr_no_quotes, SupportedMethods)
);

END_NAMESPACE(Contoso);
```

# Implement the behavior of the device

Now add code that implements the behavior defined in the model.

1. Add the following functions that handle the desired properties set in the solution dashboard. These desired properties are defined in the model:

```
void onDesiredTemperatureMeanValue(void* argument)
{
  /* By convention 'argument' is of the type of the MODEL */
  Thermostat* thermostat = argument;
  printf("Received a new desired_TemperatureMeanValue = %f\r\n", thermostat->TemperatureMeanValue);

}

void onDesiredTelemetryInterval(void* argument)
{
  /* By convention 'argument' is of the type of the MODEL */
  Thermostat* thermostat = argument;
  printf("Received a new desired_TelemetryInterval = %d\r\n", thermostat->TelemetryInterval);
}
```

2. Add the following functions that handle the direct methods invoked through the IoT hub. These direct methods are defined in the model:

```
/* Handlers for direct methods */
METHODRETURN_HANDLE Reboot(Thermostat* thermostat)
{
  (void)(thermostat);

  METHODRETURN_HANDLE result = MethodReturn_Create(201, "\"Rebooting\"");
  printf("Received reboot request\r\n");
  return result;
}

METHODRETURN_HANDLE InitiateFirmwareUpdate(Thermostat* thermostat, ascii_char_ptr FwPackageURI)
{
  (void)(thermostat);

  METHODRETURN_HANDLE result = MethodReturn_Create(201, "\"Initiating Firmware Update\"");
  printf("Recieved firmware update request. Use package at: %s\r\n", FwPackageURI);
  return result;
}
```

3. Add the following function that sends a message to the preconfigured solution:

```
/* Send data to IoT Hub */
static void sendMessage(IOTHUB_CLIENT_HANDLE iotHubClientHandle, const unsigned char* buffer, size_t
size)
{
  IOTHUB_MESSAGE_HANDLE messageHandle = IoTHubMessage_CreateFromByteArray(buffer, size);
  if (messageHandle == NULL)
  {
    printf("unable to create a new IoTHubMessage\r\n");
  }
  else
  {
    if (IoTHubClient_SendEventAsync(iotHubClientHandle, messageHandle, NULL, NULL) !=
IOTHUB_CLIENT_OK)
    {
      printf("failed to hand over the message to IoTHubClient");
    }
    else
    {
      printf("IoTHubClient accepted the message for delivery\r\n");
    }

    IoTHubMessage_Destroy(messageHandle);
  }
  free((void*)buffer);
}
```

4. Add the following callback handler that runs when the device has sent new reported property values to the preconfigured solution:

```
/* Callback after sending reported properties */
void deviceTwinCallback(int status_code, void* userContextCallback)
{
  (void)(userContextCallback);
  printf("IoTHub: reported properties delivered with status_code = %u\n", status_code);
}
```

5. Add the following function to connect your device to the preconfigured solution in the cloud, and exchange data. This function performs the following steps:

   - Initializes the platform.
   - Registers the Contoso namespace with the serialization library.
   - Initializes the client with the device connection string.
   - Create an instance of the **Thermostat** model.
   - Creates and sends reported property values.
   - Sends a **DeviceInfo** object.
   - Creates a loop to send telemetry every second.
   - Deinitializes all resources.

```
  void remote_monitoring_run(void)
  {
    if (platform_init() != 0)
    {
      printf("Failed to initialize the platform.\n");
    }
    else
    {
      if (SERIALIZER_REGISTER_NAMESPACE(Contoso) == NULL)
      {
        printf("Unable to SERIALIZER_REGISTER_NAMESPACE\n");
      }
```

```
    else
    {
      IOTHUB_CLIENT_HANDLE iotHubClientHandle =
IoTHubClient_CreateFromConnectionString(connectionString, MQTT_Protocol);
      if (iotHubClientHandle == NULL)
      {
        printf("Failure in IoTHubClient_CreateFromConnectionString\n");
      }
      else
      {
#ifdef MBED_BUILD_TIMESTAMP
        // For mbed add the certificate information
        if (IoTHubClient_SetOption(iotHubClientHandle, "TrustedCerts", certificates) !=
IOTHUB_CLIENT_OK)
        {
            printf("Failed to set option \"TrustedCerts\"\n");
        }
#endif // MBED_BUILD_TIMESTAMP
        Thermostat* thermostat = IoTHubDeviceTwin_CreateThermostat(iotHubClientHandle);
        if (thermostat == NULL)
        {
          printf("Failure in IoTHubDeviceTwin_CreateThermostat\n");
        }
        else
        {
          /* Set values for reported properties */
          thermostat->Config.TemperatureMeanValue = 55.5;
          thermostat->Config.TelemetryInterval = 3;
          thermostat->Device.DeviceState = "normal";
          thermostat->Device.Location.Latitude = 47.642877;
          thermostat->Device.Location.Longitude = -122.125497;
          thermostat->System.Manufacturer = "Contoso Inc.";
          thermostat->System.FirmwareVersion = "2.22";
          thermostat->System.InstalledRAM = "8 MB";
          thermostat->System.ModelNumber = "DB-14";
          thermostat->System.Platform = "Plat 9.75";
          thermostat->System.Processor = "i3-7";
          thermostat->System.SerialNumber = "SER21";
          /* Specify the signatures of the supported direct methods */
          thermostat->SupportedMethods = "{\"Reboot\": \"Reboot the device\",
\"InitiateFirmwareUpdate--FwPackageURI-string\": \"Updates device Firmware. Use parameter
FwPackageURI to specifiy the URI of the firmware file\"}";

          /* Send reported properties to IoT Hub */
          if (IoTHubDeviceTwin_SendReportedStateThermostat(thermostat, deviceTwinCallback,
NULL) != IOTHUB_CLIENT_OK)
          {
            printf("Failed sending serialized reported state\n");
          }
          else
          {
            printf("Send DeviceInfo object to IoT Hub at startup\n");

            thermostat->ObjectType = "DeviceInfo";
            thermostat->IsSimulatedDevice = 0;
            thermostat->Version = "1.0";
            thermostat->DeviceProperties.HubEnabledState = 1;
            thermostat->DeviceProperties.DeviceID = (char*)deviceId;

            unsigned char* buffer;
            size_t bufferSize;

            if (SERIALIZE(&buffer, &bufferSize, thermostat->ObjectType, thermostat->Version,
thermostat->IsSimulatedDevice, thermostat->DeviceProperties) != CODEFIRST_OK)
            {
              (void)printf("Failed serializing DeviceInfo\n");
            }
            else
            {
```

```
              sendMessage(iotHubClientHandle, buffer, bufferSize);
            }

            /* Send telemetry */
            thermostat->Temperature = 50;
            thermostat->ExternalTemperature = 55;
            thermostat->Humidity = 50;
            thermostat->DeviceId = (char*)deviceId;

            while (1)
            {
              unsigned char*buffer;
              size_t bufferSize;

              (void)printf("Sending sensor value Temperature = %f, Humidity = %f\n",
thermostat->Temperature, thermostat->Humidity);

              if (SERIALIZE(&buffer, &bufferSize, thermostat->DeviceId, thermostat-
>Temperature, thermostat->Humidity, thermostat->ExternalTemperature) != CODEFIRST_OK)
              {
                (void)printf("Failed sending sensor value\r\n");
              }
              else
              {
                sendMessage(iotHubClientHandle, buffer, bufferSize);
              }

              ThreadAPI_Sleep(1000);
            }

            IoTHubDeviceTwin_DestroyThermostat(thermostat);
          }
        }
        IoTHubClient_Destroy(iotHubClientHandle);
      }
      serializer_deinit();
    }
  }
  platform_deinit();
}
```

For reference, here is a sample **Telemetry** message sent to the preconfigured solution:

```
{"DeviceId":"mydevice01", "Temperature":50, "Humidity":50, "ExternalTemperature":55}
```

## Build and run the sample

Add code to invoke the **remote_monitoring_run** function and then build and run the device application.

1. Replace the **main** function with following code to invoke the **remote_monitoring_run** function:

```
int main()
{
  remote_monitoring_run();
  return 0;
}
```

2. Click **Build** and then **Build Solution** to build the device application.

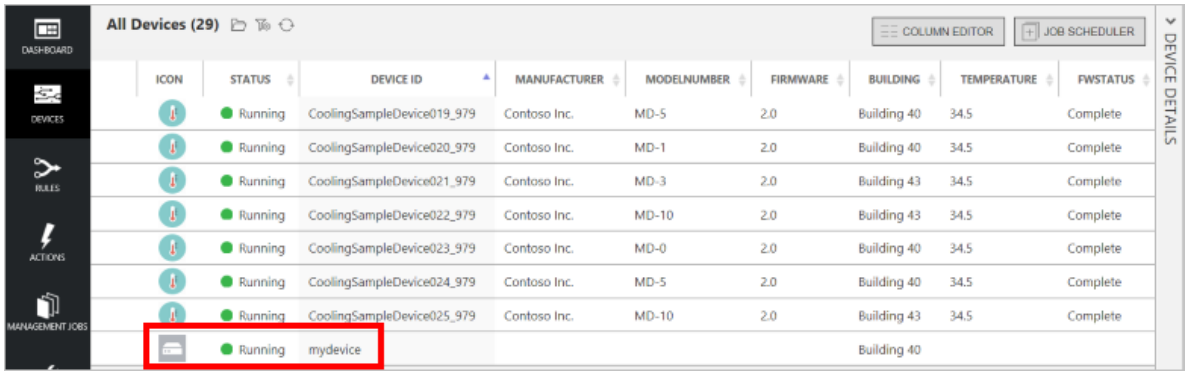
3. In **Solution Explorer**, right-click the **RMDevice** project, click **Debug**, and then click **Start new instance** to run the sample. The console displays messages as the application sends sample telemetry to the preconfigured solution, receives desired property values set in the solution dashboard, and responds to

methods invoked from the solution dashboard.

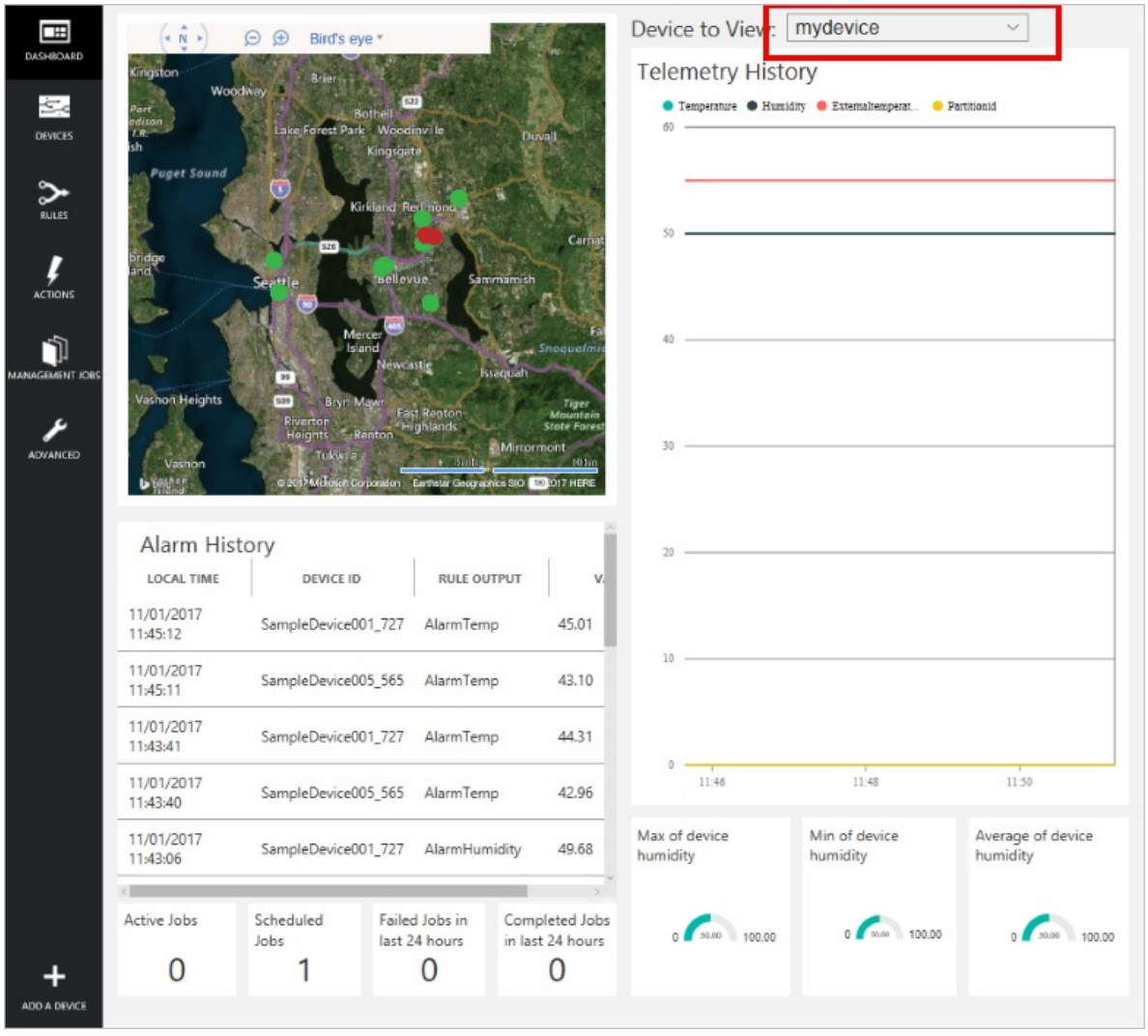
## View device telemetry in the dashboard

The dashboard in the remote monitoring solution enables you to view the telemetry your devices send to IoT Hub.

1. In your browser, return to the remote monitoring solution dashboard, click **Devices** in the left-hand panel to navigate to the **Devices list**.
2. In the **Devices list**, you should see that the status of your device is **Running**. If not, click **Enable Device** in the **Device Details** panel.



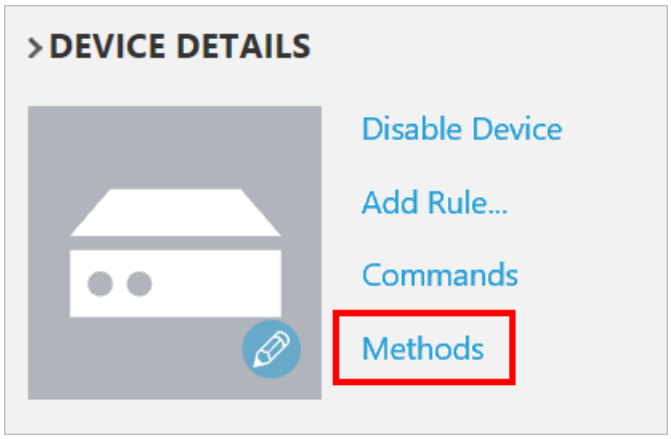

3. Click **Dashboard** to return to the dashboard, select your device in the **Device to View** drop-down to view its telemetry. The telemetry from the sample application is 50 units for internal temperature, 55 units for external temperature, and 50 units for humidity.



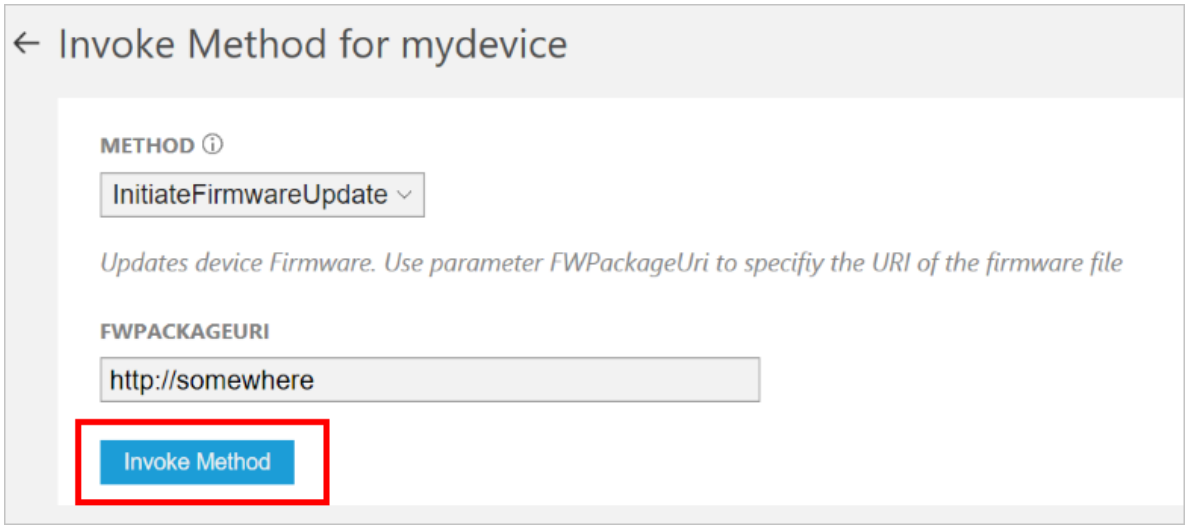
# Invoke a method on your device

The dashboard in the remote monitoring solution enables you to invoke methods on your devices through IoT Hub. For example, in the remote monitoring solution you can invoke a method to simulate rebooting a device.

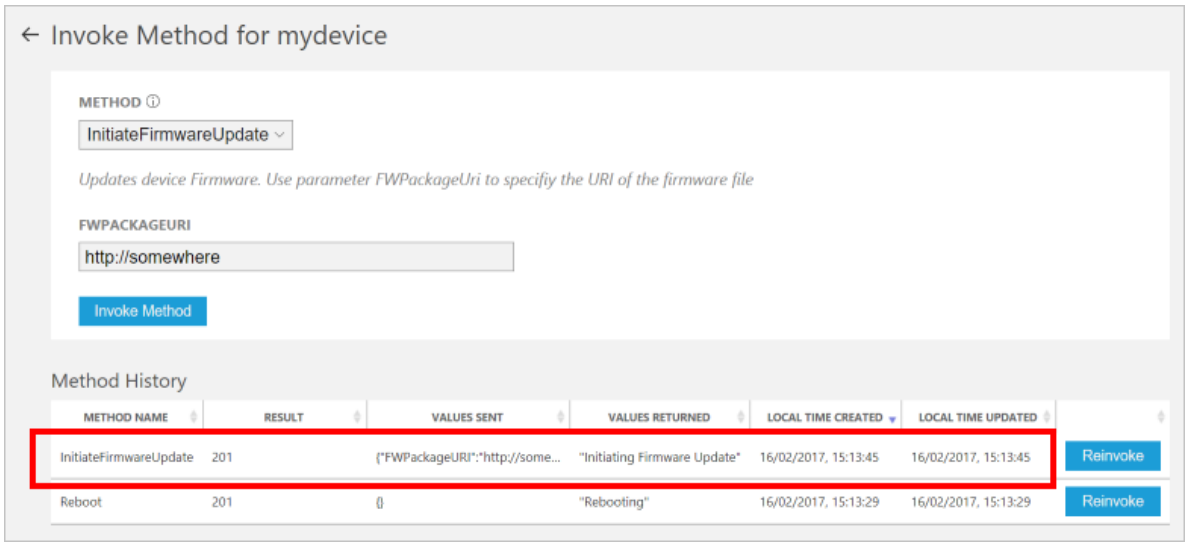
1. In the remote monitoring solution dashboard, click **Devices** in the left-hand panel to navigate to the **Devices list**.
2. Click **Device ID** for your device in the **Devices list**.
3. In the **Device details** panel, click **Methods**.




4. In the **Method** drop-down, select **InitiateFirmwareUpdate**, and then in **FWPACKAGEURI** enter a dummy URL. Click **Invoke Method** to call the method on the device.




5. You see a message in the console running your device code when the device handles the method. The results of the method are added to the history in the solution portal:




## Next steps

The article Customizing preconfigured solutions describes some ways you can extend this sample. Possible extensions include using real sensors and implementing additional commands.

You can learn more about the permissions on the azureiotsuite.com site.

# Connect your device to the remote monitoring preconfigured solution (Linux)

8/24/2017 • 11 min to read • Edit Online

## Scenario overview

In this scenario, you create a device that sends the following telemetry to the remote monitoring preconfigured solution:

- External temperature
- Internal temperature
- Humidity

For simplicity, the code on the device generates sample values, but we encourage you to extend the sample by connecting real sensors to your device and sending real telemetry.

The device is also able to respond to methods invoked from the solution dashboard and desired property values set in the solution dashboard.

To complete this tutorial, you need an active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

## Before you start

Before you write any code for your device, you must provision your remote monitoring preconfigured solution and provision a new custom device in that solution.

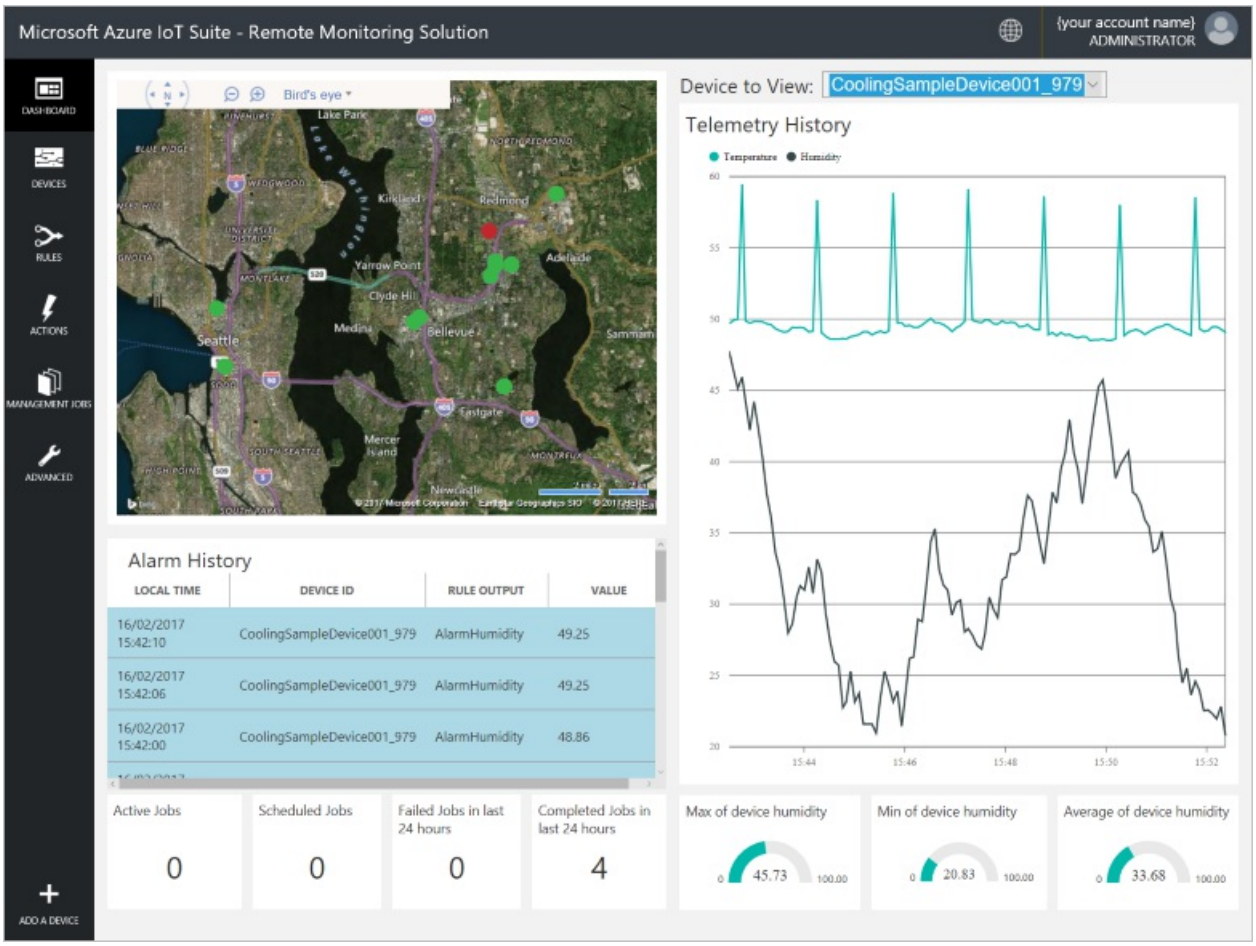
**Provision your remote monitoring preconfigured solution**

The device you create in this tutorial sends data to an instance of the remote monitoring preconfigured solution. If you haven't already provisioned the remote monitoring preconfigured solution in your Azure account, use the following steps:

1. On the https://www.azureiotsuite.com/ page, click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** panel to create your solution.
3. On the **Create Remote monitoring solution** page, enter a **Solution name** of your choice, select the **Region** you want to deploy to, and select the Azure subscription to want to use. Then click **Create solution**.
4. Wait until the provisioning process completes.

> **WARNING**
> The preconfigured solutions use billable Azure services. Be sure to remove the preconfigured solution from your subscription when you are done with it to avoid any unnecessary charges. You can completely remove a preconfigured solution from your subscription by visiting the https://www.azureiotsuite.com/ page.

When the provisioning process for the remote monitoring solution finishes, click **Launch** to open the solution dashboard in your browser.



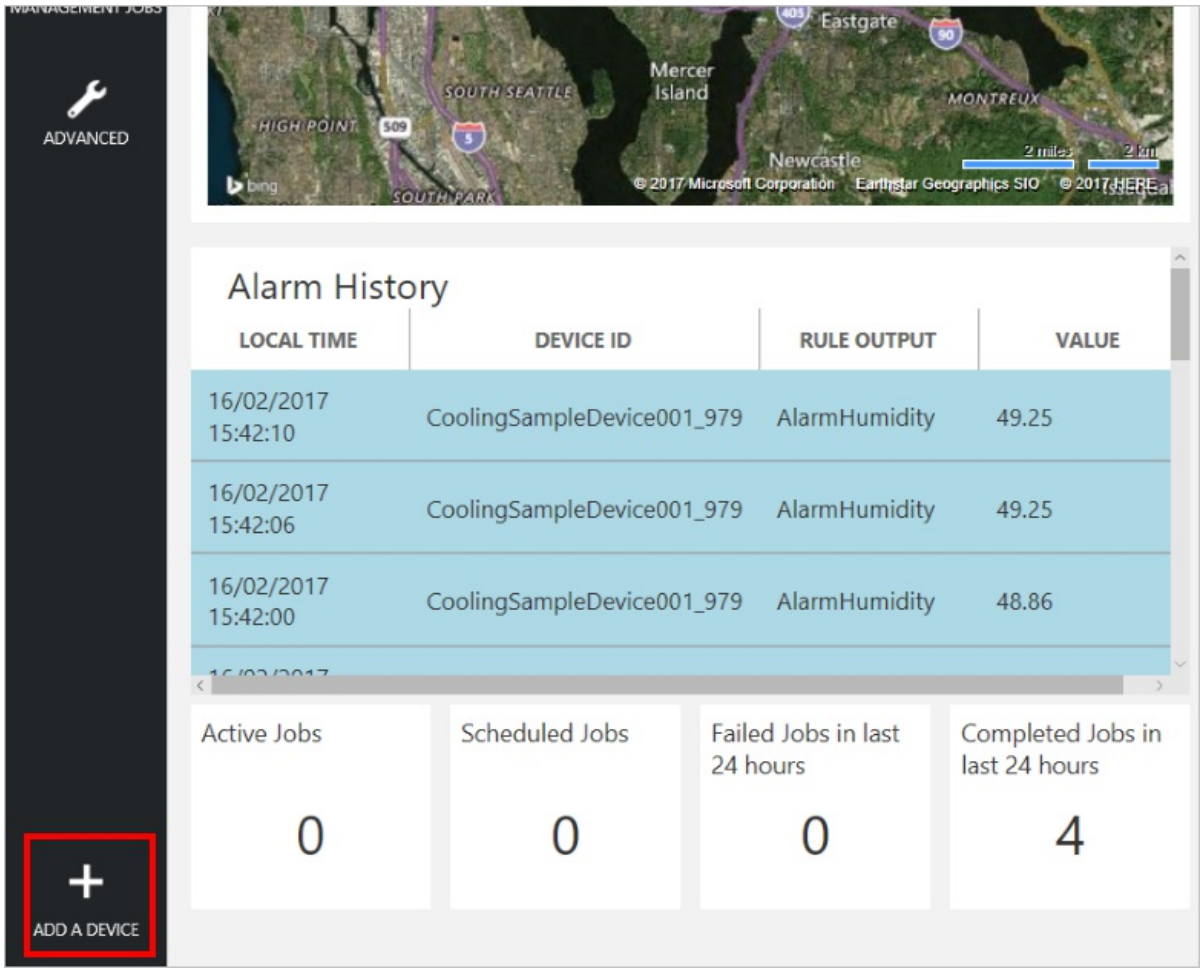

**Provision your device in the remote monitoring solution**

> **NOTE**
> If you have already provisioned a device in your solution, you can skip this step. You need to know the device credentials when you create the client application.

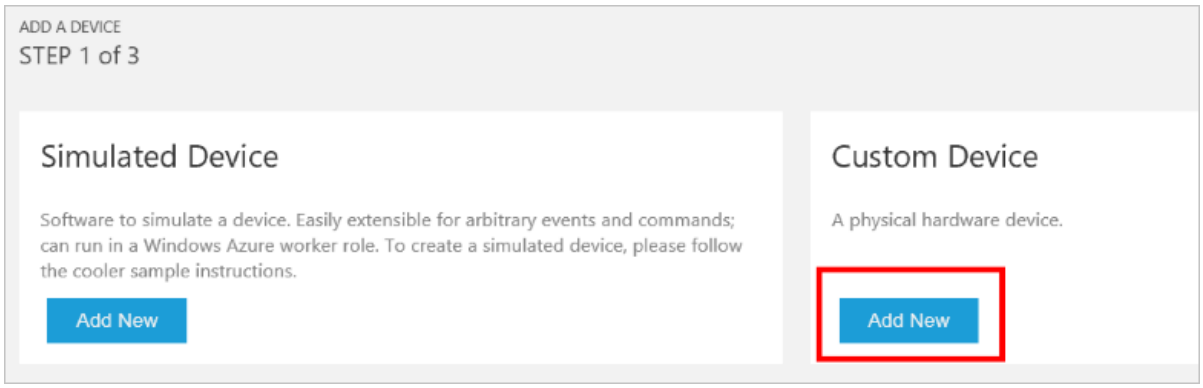
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

To add a device to your remote monitoring solution, complete the following steps in the solution dashboard:

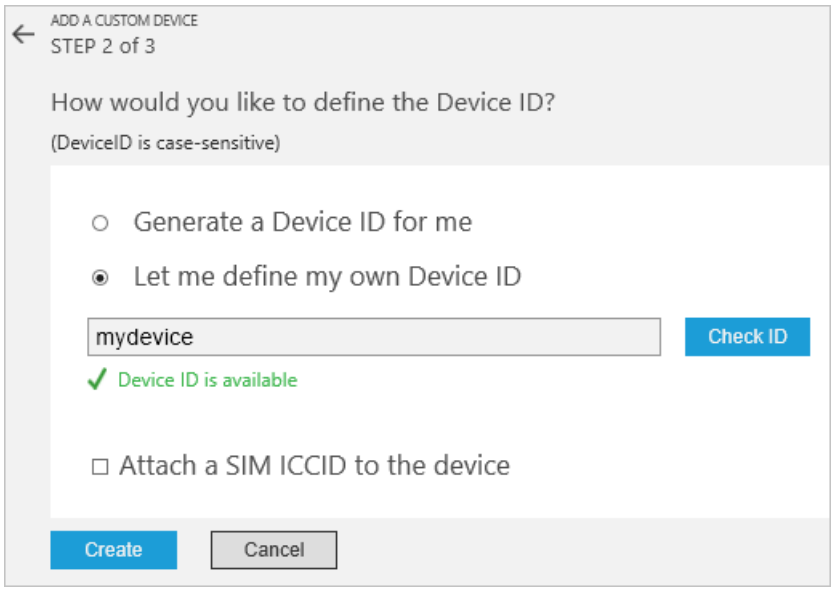
1. In the lower left-hand corner of the dashboard, click **Add a device**.



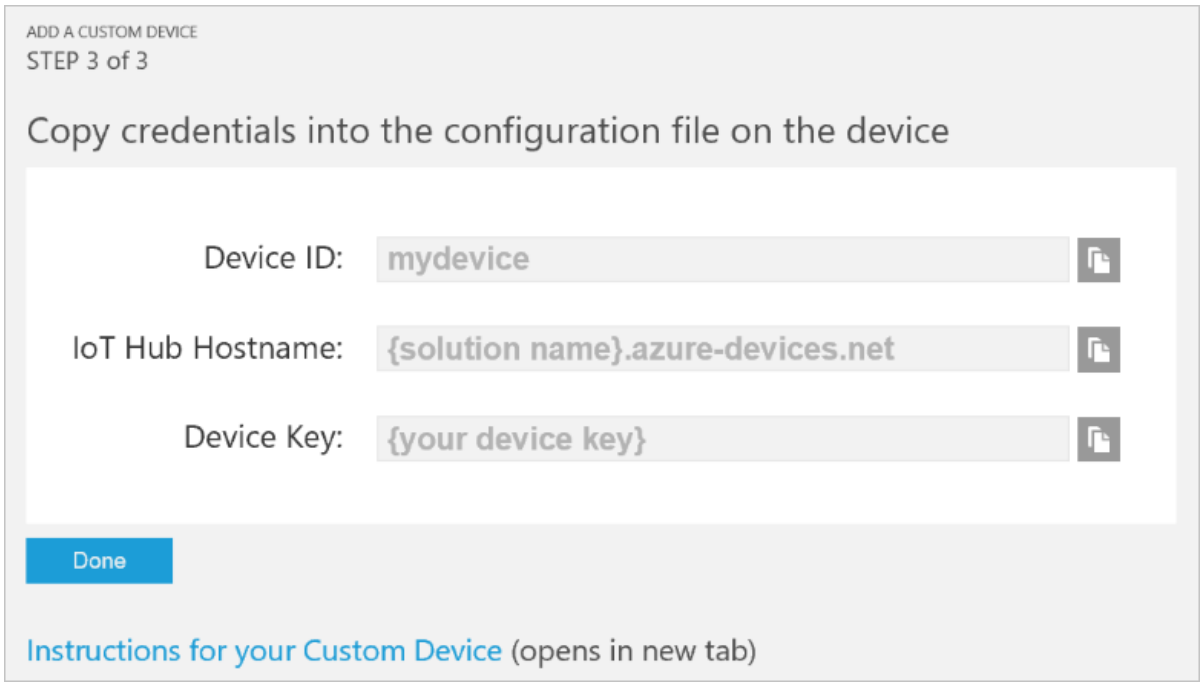

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **mydevice**, click **Check ID** to verify that name isn't already in use, and then click **Create** to provision the device.




4. Make a note the device credentials (Device ID, IoT Hub Hostname, and Device Key). Your client application needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Build and run a sample C client Linux

The following steps show you how to create a client application that communicates with the remote monitoring preconfigured solution. This application is written in C and built and run on Ubuntu Linux.

To complete these steps, you need a device running Ubuntu version 15.04 or 15.10. Before proceeding, install the prerequisite packages on your Ubuntu device using the following command:

```
sudo apt-get install cmake gcc g++
```

## Install the client libraries on your device

The Azure IoT Hub client libraries are available as a package you can install on your Ubuntu device using the **apt-get** command. Complete the following steps to install the package that contains the IoT Hub client library and header files on your Ubuntu computer:

1. In a shell, add the AzureIoT repository to your computer:

```
sudo add-apt-repository ppa:aziotsdklinux/ppa-azureiot
sudo apt-get update
```

2. Install the azure-iot-sdk-c-dev package

```
sudo apt-get install -y azure-iot-sdk-c-dev
```

## Install the Parson JSON parser

The IoT Hub client libraries use the Parson JSON parser to parse message payloads. In a suitable folder on your computer, clone the Parson GitHub repository using the following command:

```
git clone https://github.com/kgabis/parson.git
```

## Prepare your project

On your Ubuntu machine, create a folder called **remote_monitoring**. In the **remote_monitoring** folder:

- Create the four files **main.c**, **remote_monitoring.c**, **remote_monitoring.h**, and **CMakeLists.txt**.
- Create folder called **parson**.

Copy the files **parson.c** and **parson.h** from your local copy of the Parson repository into the **remote_monitoring/parson** folder.

In a text editor, open the **remote_monitoring.c** file. Add the following `#include` statements:

```
#include "iothubtransportmqtt.h"
#include "schemalib.h"
#include "iothub_client.h"
#include "serializer_devicetwin.h"
#include "schemaserializer.h"
#include "azure_c_shared_utility/threadapi.h"
#include "azure_c_shared_utility/platform.h"
#include "parson.h"
```

## Specify the behavior of the IoT device

The IoT Hub serializer client library uses a model to specify the format of the messages the device exchanges with IoT Hub.

1. Add the following variable declarations after the `#include` statements. Replace the placeholder values [Device Id] and [Device Key] with values you noted for your device in the remote monitoring solution dashboard. Use the IoT Hub Hostname from the solution dashboard to replace [IoTHub Name]. For example, if your IoT Hub Hostname is **contoso.azure-devices.net**, replace [IoTHub Name] with **contoso**:

```
static const char* deviceId = "[Device Id]";
static const char* connectionString = "HostName=[IoTHub Name].azure-devices.net;DeviceId=[Device
Id];SharedAccessKey=[Device Key]";
```

2. Add the following code to define the model that enables the device to communicate with IoT Hub. This model specifies that the device:

   - Can send temperature, external temperature, humidity, and a device id as telemetry.
   - Can send metadata about the device to IoT Hub. The device sends basic metadata in a **DeviceInfo** object at startup.
   - Can send reported properties, to the device twin in IoT Hub. These reported properties are grouped into configuration, device, and system properties.
   - Can receive and act on desired properties set in the device twin in IoT Hub.
   - Can respond to the **Reboot** and **InitiateFirmwareUpdate** direct methods invoked through the solution portal. The device sends information about the direct methods it supports using reported properties.

```
// Define the Model
BEGIN_NAMESPACE(Contoso);

/* Reported properties */
DECLARE_STRUCT(SystemProperties,
 ascii_char_ptr, Manufacturer,
 ascii_char_ptr, FirmwareVersion,
 ascii_char_ptr, InstalledRAM,
 ascii_char_ptr, ModelNumber,
 ascii_char_ptr, Platform,
 ascii_char_ptr, Processor,
 ascii_char_ptr, SerialNumber
);

DECLARE_STRUCT(LocationProperties,
 double, Latitude,
 double, Longitude
);

DECLARE_STRUCT(ReportedDeviceProperties,
 ascii_char_ptr, DeviceState,
 LocationProperties, Location
);

DECLARE_MODEL(ConfigProperties,
 WITH_REPORTED_PROPERTY(double, TemperatureMeanValue),
 WITH_REPORTED_PROPERTY(uint8_t, TelemetryInterval)
);

/* Part of DeviceInfo */
DECLARE_STRUCT(DeviceProperties,
 ascii_char_ptr, DeviceID,
 _Bool, HubEnabledState
);

DECLARE_DEVICETWIN_MODEL(Thermostat,
 /* Telemetry (temperature, external temperature and humidity) */
 WITH_DATA(double, Temperature),
 WITH_DATA(double, ExternalTemperature),
 WITH_DATA(double, Humidity),
 WITH_DATA(ascii_char_ptr, DeviceId),

 /* DeviceInfo */
 WITH_DATA(ascii_char_ptr, ObjectType),
 WITH_DATA(_Bool, IsSimulatedDevice),
 WITH_DATA(ascii_char_ptr, Version),
 WITH_DATA(DeviceProperties, DeviceProperties),

 /* Device twin properties */
 WITH_REPORTED_PROPERTY(ReportedDeviceProperties, Device),
 WITH_REPORTED_PROPERTY(ConfigProperties, Config),
 WITH_REPORTED_PROPERTY(SystemProperties, System),

 WITH_DESIRED_PROPERTY(double, TemperatureMeanValue, onDesiredTemperatureMeanValue),
 WITH_DESIRED_PROPERTY(uint8_t, TelemetryInterval, onDesiredTelemetryInterval),

 /* Direct methods implemented by the device */
 WITH_METHOD(Reboot),
 WITH_METHOD(InitiateFirmwareUpdate, ascii_char_ptr, FwPackageURI),

 /* Register direct methods with solution portal */
 WITH_REPORTED_PROPERTY(ascii_char_ptr_no_quotes, SupportedMethods)
);

END_NAMESPACE(Contoso);
```

# Implement the behavior of the device

Now add code that implements the behavior defined in the model.

1. Add the following functions that handle the desired properties set in the solution dashboard. These desired properties are defined in the model:

```
void onDesiredTemperatureMeanValue(void* argument)
{
  /* By convention 'argument' is of the type of the MODEL */
  Thermostat* thermostat = argument;
  printf("Received a new desired_TemperatureMeanValue = %f\r\n", thermostat->TemperatureMeanValue);

}

void onDesiredTelemetryInterval(void* argument)
{
  /* By convention 'argument' is of the type of the MODEL */
  Thermostat* thermostat = argument;
  printf("Received a new desired_TelemetryInterval = %d\r\n", thermostat->TelemetryInterval);
}
```

2. Add the following functions that handle the direct methods invoked through the IoT hub. These direct methods are defined in the model:

```
/* Handlers for direct methods */
METHODRETURN_HANDLE Reboot(Thermostat* thermostat)
{
  (void)(thermostat);

  METHODRETURN_HANDLE result = MethodReturn_Create(201, "\"Rebooting\"");
  printf("Received reboot request\r\n");
  return result;
}

METHODRETURN_HANDLE InitiateFirmwareUpdate(Thermostat* thermostat, ascii_char_ptr FwPackageURI)
{
  (void)(thermostat);

  METHODRETURN_HANDLE result = MethodReturn_Create(201, "\"Initiating Firmware Update\"");
  printf("Recieved firmware update request. Use package at: %s\r\n", FwPackageURI);
  return result;
}
```

3. Add the following function that sends a message to the preconfigured solution:

```
/* Send data to IoT Hub */
static void sendMessage(IOTHUB_CLIENT_HANDLE iotHubClientHandle, const unsigned char* buffer, size_t
size)
{
  IOTHUB_MESSAGE_HANDLE messageHandle = IoTHubMessage_CreateFromByteArray(buffer, size);
  if (messageHandle == NULL)
  {
    printf("unable to create a new IoTHubMessage\r\n");
  }
  else
  {
    if (IoTHubClient_SendEventAsync(iotHubClientHandle, messageHandle, NULL, NULL) != IOTHUB_CLIENT_OK)
    {
      printf("failed to hand over the message to IoTHubClient");
    }
    else
    {
      printf("IoTHubClient accepted the message for delivery\r\n");
    }

    IoTHubMessage_Destroy(messageHandle);
  }
  free((void*)buffer);
}
```

4. Add the following callback handler that runs when the device has sent new reported property values to the preconfigured solution:

```
/* Callback after sending reported properties */
void deviceTwinCallback(int status_code, void* userContextCallback)
{
  (void)(userContextCallback);
  printf("IoTHub: reported properties delivered with status_code = %u\n", status_code);
}
```

5. Add the following function to connect your device to the preconfigured solution in the cloud, and exchange data. This function performs the following steps:

   - Initializes the platform.
   - Registers the Contoso namespace with the serialization library.
   - Initializes the client with the device connection string.
   - Create an instance of the **Thermostat** model.
   - Creates and sends reported property values.
   - Sends a **DeviceInfo** object.
   - Creates a loop to send telemetry every second.
   - Deinitializes all resources.

```
 void remote_monitoring_run(void)
 {
   if (platform_init() != 0)
   {
     printf("Failed to initialize the platform.\n");
   }
   else
   {
     if (SERIALIZER_REGISTER_NAMESPACE(Contoso) == NULL)
     {
       printf("Unable to SERIALIZER_REGISTER_NAMESPACE\n");
     }
     else
```

```
    {
      IOTHUB_CLIENT_HANDLE iotHubClientHandle =
IoTHubClient_CreateFromConnectionString(connectionString, MQTT_Protocol);
      if (iotHubClientHandle == NULL)
      {
        printf("Failure in IoTHubClient_CreateFromConnectionString\n");
      }
      else
      {
#ifdef MBED_BUILD_TIMESTAMP
        // For mbed add the certificate information
        if (IoTHubClient_SetOption(iotHubClientHandle, "TrustedCerts", certificates) !=
IOTHUB_CLIENT_OK)
        {
            printf("Failed to set option \"TrustedCerts\"\n");
        }
#endif // MBED_BUILD_TIMESTAMP
        Thermostat* thermostat = IoTHubDeviceTwin_CreateThermostat(iotHubClientHandle);
        if (thermostat == NULL)
        {
          printf("Failure in IoTHubDeviceTwin_CreateThermostat\n");
        }
        else
        {
          /* Set values for reported properties */
          thermostat->Config.TemperatureMeanValue = 55.5;
          thermostat->Config.TelemetryInterval = 3;
          thermostat->Device.DeviceState = "normal";
          thermostat->Device.Location.Latitude = 47.642877;
          thermostat->Device.Location.Longitude = -122.125497;
          thermostat->System.Manufacturer = "Contoso Inc.";
          thermostat->System.FirmwareVersion = "2.22";
          thermostat->System.InstalledRAM = "8 MB";
          thermostat->System.ModelNumber = "DB-14";
          thermostat->System.Platform = "Plat 9.75";
          thermostat->System.Processor = "i3-7";
          thermostat->System.SerialNumber = "SER21";
          /* Specify the signatures of the supported direct methods */
          thermostat->SupportedMethods = "{\"Reboot\": \"Reboot the device\",
\"InitiateFirmwareUpdate--FwPackageURI-string\": \"Updates device Firmware. Use parameter
FwPackageURI to specifiy the URI of the firmware file\"}";

          /* Send reported properties to IoT Hub */
          if (IoTHubDeviceTwin_SendReportedStateThermostat(thermostat, deviceTwinCallback, NULL)
!= IOTHUB_CLIENT_OK)
          {
            printf("Failed sending serialized reported state\n");
          }
          else
          {
            printf("Send DeviceInfo object to IoT Hub at startup\n");

            thermostat->ObjectType = "DeviceInfo";
            thermostat->IsSimulatedDevice = 0;
            thermostat->Version = "1.0";
            thermostat->DeviceProperties.HubEnabledState = 1;
            thermostat->DeviceProperties.DeviceID = (char*)deviceId;

            unsigned char* buffer;
            size_t bufferSize;

            if (SERIALIZE(&buffer, &bufferSize, thermostat->ObjectType, thermostat->Version,
thermostat->IsSimulatedDevice, thermostat->DeviceProperties) != CODEFIRST_OK)
            {
              (void)printf("Failed serializing DeviceInfo\n");
            }
            else
            {
              sendMessage(iotHubClientHandle, buffer, bufferSize);
```

```
            }

            /* Send telemetry */
            thermostat->Temperature = 50;
            thermostat->ExternalTemperature = 55;
            thermostat->Humidity = 50;
            thermostat->DeviceId = (char*)deviceId;

            while (1)
            {
              unsigned char*buffer;
              size_t bufferSize;

              (void)printf("Sending sensor value Temperature = %f, Humidity = %f\n", thermostat-
>Temperature, thermostat->Humidity);

              if (SERIALIZE(&buffer, &bufferSize, thermostat->DeviceId, thermostat->Temperature,
thermostat->Humidity, thermostat->ExternalTemperature) != CODEFIRST_OK)
              {
                (void)printf("Failed sending sensor value\r\n");
              }
              else
              {
                sendMessage(iotHubClientHandle, buffer, bufferSize);
              }

              ThreadAPI_Sleep(1000);
            }

            IoTHubDeviceTwin_DestroyThermostat(thermostat);
          }
        }
        IoTHubClient_Destroy(iotHubClientHandle);
      }
      serializer_deinit();
    }
  }
  platform_deinit();
}
```

For reference, here is a sample **Telemetry** message sent to the preconfigured solution:

```
{"DeviceId":"mydevice01", "Temperature":50, "Humidity":50, "ExternalTemperature":55}
```

## Call the remote_monitoring_run function

In a text editor, open the **remote_monitoring.h** file. Add the following code:

```
void remote_monitoring_run(void);
```

In a text editor, open the **main.c** file. Add the following code:

```
#include "remote_monitoring.h"

int main(void)
{
    remote_monitoring_run();

    return 0;
}
```

# Build and run the application

The following steps describe how to use *CMake* to build your client application.

1. In a text editor, open the **CMakeLists.txt** file in the **remote_monitoring** folder.

2. Add the following instructions to define how to build your client application:

```
macro(compileAsC99)
  if (CMAKE_VERSION VERSION_LESS "3.1")
    if (CMAKE_C_COMPILER_ID STREQUAL "GNU")
      set (CMAKE_C_FLAGS "--std=c99 ${CMAKE_C_FLAGS}")
      set (CMAKE_CXX_FLAGS "--std=c++11 ${CMAKE_CXX_FLAGS}")
    endif()
  else()
    set (CMAKE_C_STANDARD 99)
    set (CMAKE_CXX_STANDARD 11)
  endif()
endmacro(compileAsC99)

cmake_minimum_required(VERSION 2.8.11)
compileAsC99()

set(AZUREIOT_INC_FOLDER "${CMAKE_SOURCE_DIR}" "${CMAKE_SOURCE_DIR}/parson" "/usr/include/azureiot"
"/usr/include/azureiot/inc")

include_directories(${AZUREIOT_INC_FOLDER})

set(sample_application_c_files
    ./parson/parson.c
    ./remote_monitoring.c
    ./main.c
)

set(sample_application_h_files
    ./parson/parson.h
    ./remote_monitoring.h
)

add_executable(sample_app ${sample_application_c_files} ${sample_application_h_files})

target_link_libraries(sample_app
    serializer
    iothub_client
    iothub_client_mqtt_transport
    aziotsharedutil
    umqtt
    pthread
    curl
    ssl
    crypto
    m
)
```

3. In the **remote_monitoring** folder, create a folder to store the *make* files that CMake generates and then run the **cmake** and **make** commands as follows:

```
mkdir cmake
cd cmake
cmake ../
make
```

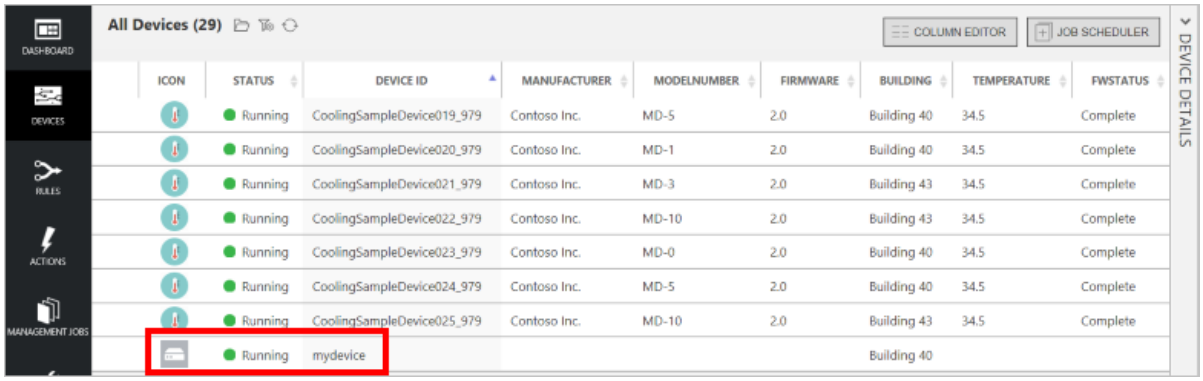
4. Run the client application and send telemetry to IoT Hub:

```
./sample_app
```

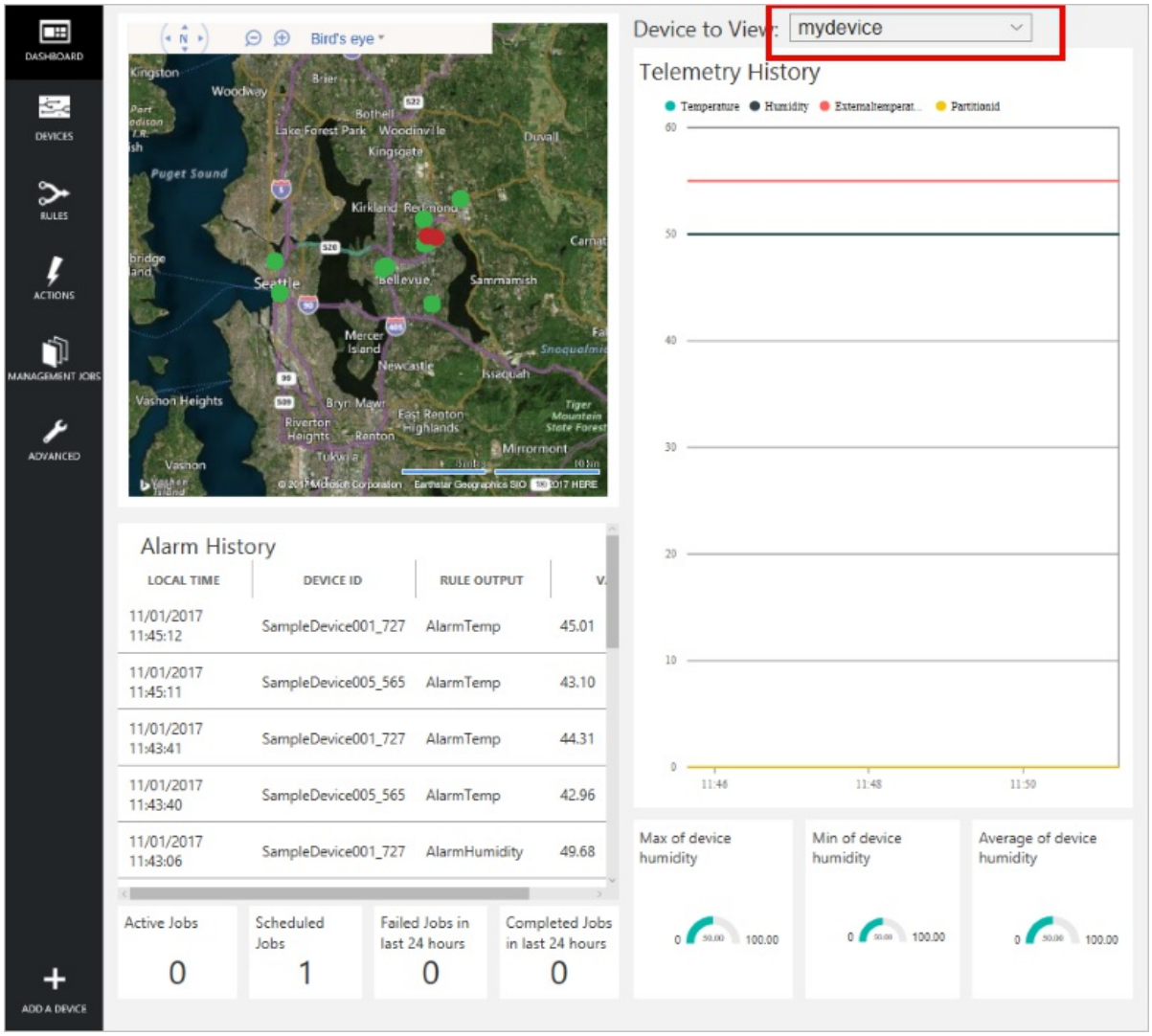
## View device telemetry in the dashboard

The dashboard in the remote monitoring solution enables you to view the telemetry your devices send to IoT Hub.

1. In your browser, return to the remote monitoring solution dashboard, click **Devices** in the left-hand panel to navigate to the **Devices list**.
2. In the **Devices list**, you should see that the status of your device is **Running**. If not, click **Enable Device** in the **Device Details** panel.



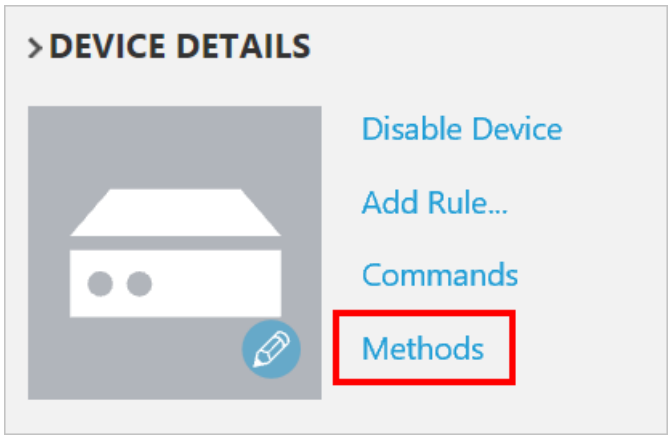

3. Click **Dashboard** to return to the dashboard, select your device in the **Device to View** drop-down to view its telemetry. The telemetry from the sample application is 50 units for internal temperature, 55 units for external temperature, and 50 units for humidity.



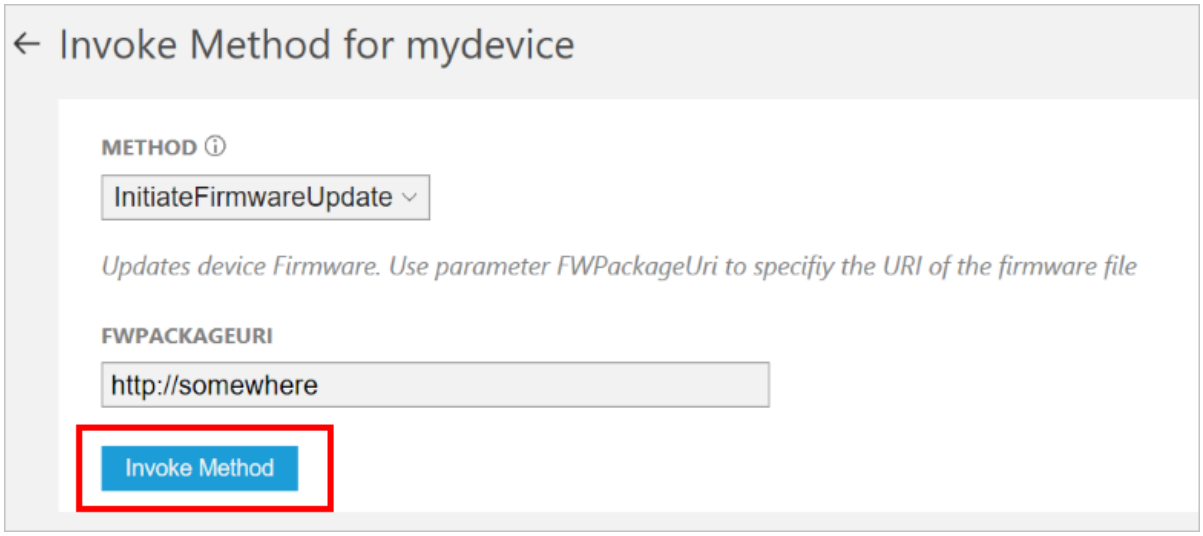
# Invoke a method on your device

The dashboard in the remote monitoring solution enables you to invoke methods on your devices through IoT Hub. For example, in the remote monitoring solution you can invoke a method to simulate rebooting a device.

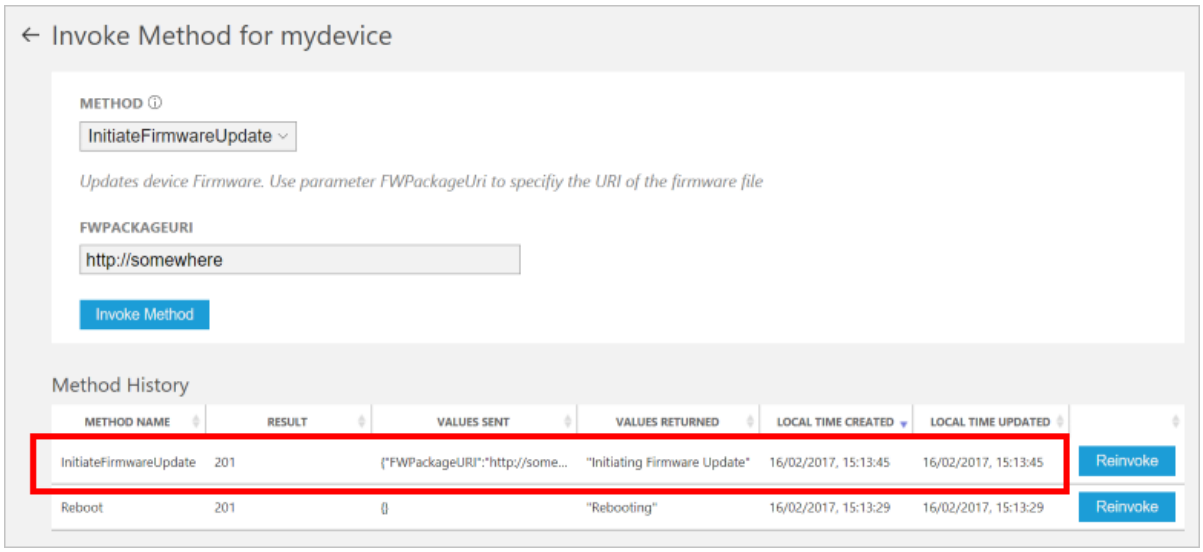
1. In the remote monitoring solution dashboard, click **Devices** in the left-hand panel to navigate to the **Devices list**.
2. Click **Device ID** for your device in the **Devices list**.
3. In the **Device details** panel, click **Methods**.




4. In the **Method** drop-down, select **InitiateFirmwareUpdate**, and then in **FWPACKAGEURI** enter a dummy URL. Click **Invoke Method** to call the method on the device.




5. You see a message in the console running your device code when the device handles the method. The results of the method are added to the history in the solution portal:




## Next steps

The article Customizing preconfigured solutions describes some ways you can extend this sample. Possible extensions include using real sensors and implementing additional commands.

You can learn more about the permissions on the azureiotsuite.com site.

# Connect your device to the remote monitoring preconfigured solution (Node.js)

8/24/2017 • 7 min to read • Edit Online

## Scenario overview

In this scenario, you create a device that sends the following telemetry to the remote monitoring preconfigured solution:

- External temperature
- Internal temperature
- Humidity

For simplicity, the code on the device generates sample values, but we encourage you to extend the sample by connecting real sensors to your device and sending real telemetry.

The device is also able to respond to methods invoked from the solution dashboard and desired property values set in the solution dashboard.

To complete this tutorial, you need an active Azure account. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

## Before you start

Before you write any code for your device, you must provision your remote monitoring preconfigured solution and provision a new custom device in that solution.

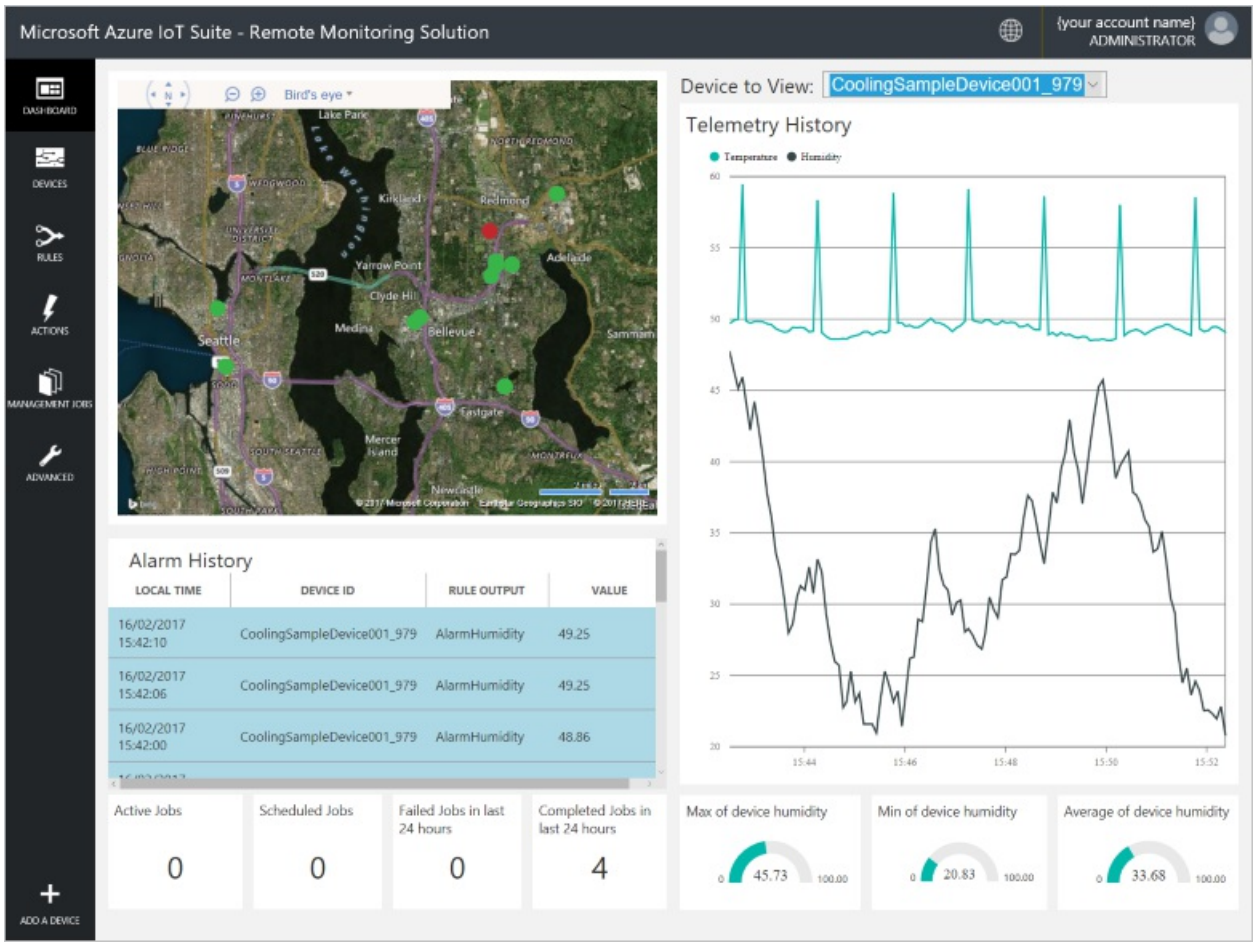
**Provision your remote monitoring preconfigured solution**

The device you create in this tutorial sends data to an instance of the remote monitoring preconfigured solution. If you haven't already provisioned the remote monitoring preconfigured solution in your Azure account, use the following steps:

1. On the https://www.azureiotsuite.com/ page, click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** panel to create your solution.
3. On the **Create Remote monitoring solution** page, enter a **Solution name** of your choice, select the **Region** you want to deploy to, and select the Azure subscription to want to use. Then click **Create solution**.
4. Wait until the provisioning process completes.

> **WARNING**
> The preconfigured solutions use billable Azure services. Be sure to remove the preconfigured solution from your subscription when you are done with it to avoid any unnecessary charges. You can completely remove a preconfigured solution from your subscription by visiting the https://www.azureiotsuite.com/ page.

When the provisioning process for the remote monitoring solution finishes, click **Launch** to open the solution dashboard in your browser.



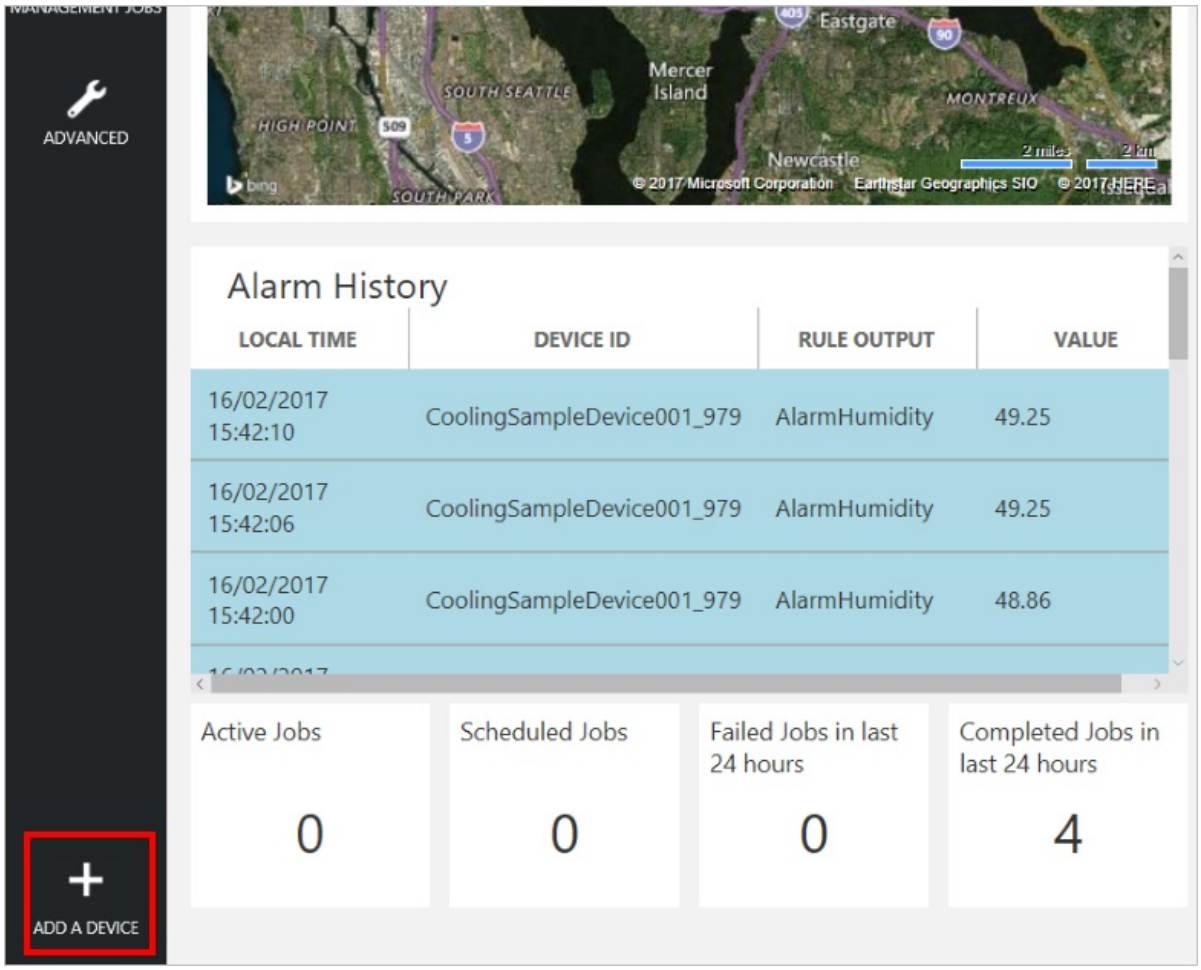

**Provision your device in the remote monitoring solution**

> **NOTE**
>
> If you have already provisioned a device in your solution, you can skip this step. You need to know the device credentials when you create the client application.

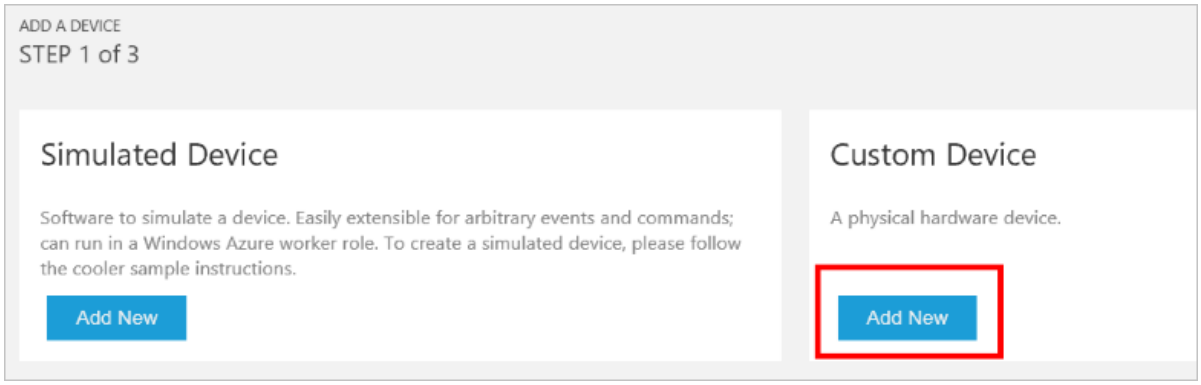
For a device to connect to the preconfigured solution, it must identify itself to IoT Hub using valid credentials. You can retrieve the device credentials from the solution dashboard. You include the device credentials in your client application later in this tutorial.

To add a device to your remote monitoring solution, complete the following steps in the solution dashboard:

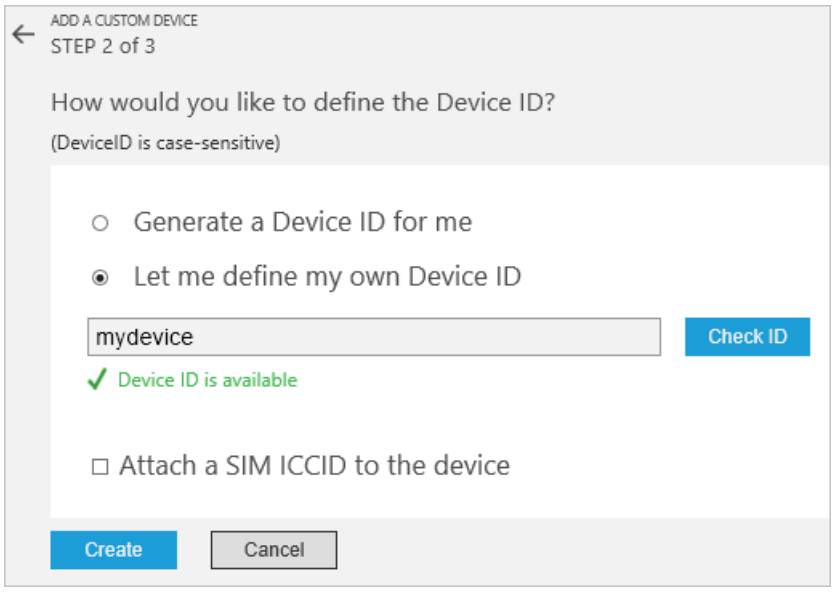
1. In the lower left-hand corner of the dashboard, click **Add a device**.



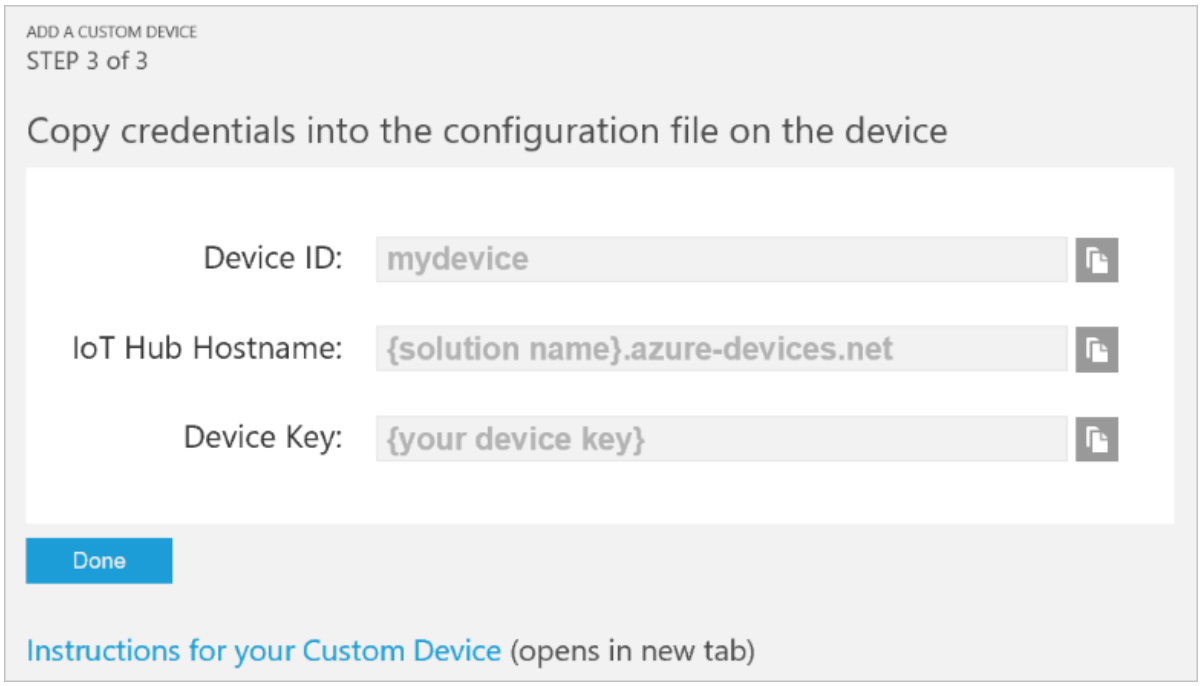

2. In the **Custom Device** panel, click **Add new**.




3. Choose **Let me define my own Device ID**. Enter a Device ID such as **mydevice**, click **Check ID** to verify that name isn't already in use, and then click **Create** to provision the device.




4. Make a note the device credentials (Device ID, IoT Hub Hostname, and Device Key). Your client application needs these values to connect to the remote monitoring solution. Then click **Done**.




5. Select your device in the device list in the solution dashboard. Then, in the **Device Details** panel, click **Enable Device**. The status of your device is now **Running**. The remote monitoring solution can now receive telemetry from your device and invoke methods on the device.

## Create a node.js sample solution

Ensure that Node.js version 0.11.5 or later is installed on your development machine. You can run `node --version` at the command line to check the version.

1. Create a folder called **RemoteMonitoring** on your development machine. Navigate to this folder in your command-line environment.

2. Run the following commands to download and install the packages you need to complete the sample app:

```
npm init
npm install azure-iot-device azure-iot-device-mqtt --save
```

3. In the **RemoteMonitoring** folder, create a file called **remote_monitoring.js**. Open this file in a text editor.

4. In the **remote_monitoring.js** file, add the following `require` statements:

```
'use strict';

var Protocol = require('azure-iot-device-mqtt').Mqtt;
var Client = require('azure-iot-device').Client;
var ConnectionString = require('azure-iot-device').ConnectionString;
var Message = require('azure-iot-device').Message;
```

5. Add the following variable declarations after the `require` statements. Replace the placeholder values [Device Id] and [Device Key] with values you noted for your device in the remote monitoring solution dashboard. Use the IoT Hub Hostname from the solution dashboard to replace [IoTHub Name]. For example, if your IoT Hub Hostname is **contoso.azure-devices.net**, replace [IoTHub Name] with **contoso**:

```
var connectionString = 'HostName=[IoTHub Name].azure-devices.net;DeviceId=[Device Id];SharedAccessKey=
[Device Key]';
var deviceId = ConnectionString.parse(connectionString).DeviceId;
```

6. Add the following variables to define some base telemetry data:

```
var temperature = 50;
var humidity = 50;
var externalTemperature = 55;
```

7. Add the following helper function to print operation results:

```
function printErrorFor(op) {
    return function printError(err) {
        if (err) console.log(op + ' error: ' + err.toString());
    };
}
```

8. Add the following helper function to use to randomize the telemetry values:

```
function generateRandomIncrement() {
    return ((Math.random() * 2) - 1);
}
```

9. Add the following definition for the **DeviceInfo** object the device sends on startup:

```
var deviceMetaData = {
    'ObjectType': 'DeviceInfo',
    'IsSimulatedDevice': 0,
    'Version': '1.0',
    'DeviceProperties': {
        'DeviceID': deviceId,
        'HubEnabledState': 1
    }
};
```

10. Add the following definition for the device twin reported values. This definition includes descriptions of the direct methods the device supports:

```
var reportedProperties = {
    "Device": {
        "DeviceState": "normal",
        "Location": {
            "Latitude": 47.642877,
            "Longitude": -122.125497
        }
    },
    "Config": {
        "TemperatureMeanValue": 56.7,
        "TelemetryInterval": 45
    },
    "System": {
        "Manufacturer": "Contoso Inc.",
        "FirmwareVersion": "2.22",
        "InstalledRAM": "8 MB",
        "ModelNumber": "DB-14",
        "Platform": "Plat 9.75",
        "Processor": "i3-9",
        "SerialNumber": "SER99"
    },
    "Location": {
        "Latitude": 47.642877,
        "Longitude": -122.125497
    },
    "SupportedMethods": {
        "Reboot": "Reboot the device",
        "InitiateFirmwareUpdate--FwPackageURI-string": "Updates device Firmware. Use parameter
FwPackageURI to specifiy the URI of the firmware file"
    },
}
```

11. Add the following function to handle the **Reboot** direct method call:

```
function onReboot(request, response) {
    // Implement actual logic here.
    console.log('Simulated reboot...');

    // Complete the response
    response.send(200, "Rebooting device", function(err) {
        if(!!err) {
            console.error('An error occurred when sending a method response:\n' + err.toString());
        } else {
            console.log('Response to method \'' + request.methodName + '\' sent successfully.' );
        }
    });
}
```

12. Add the following function to handle the **InitiateFirmwareUpdate** direct method call. This direct method uses a parameter to specify the location of the firmware image to download, and initiates the firmware update on the device asynchronously:

```
function onInitiateFirmwareUpdate(request, response) {
    console.log('Simulated firmware update initiated, using: ' + request.payload.FwPackageURI);

    // Complete the response
    response.send(200, "Firmware update initiated", function(err) {
        if(!!err) {
            console.error('An error occurred when sending a method response:\n' + err.toString());
        } else {
            console.log('Response to method \'' + request.methodName + '\' sent successfully.' );
        }
    });

    // Add logic here to perform the firmware update asynchronously
}
```

13. Add the following code to create a client instance:

```
var client = Client.fromConnectionString(connectionString, Protocol);
```

14. Add the following code to:

    - Open the connection.
    - Send the **DeviceInfo** object.
    - Set up a handler for desired properties.
    - Send reported properties.
    - Register handlers for the direct methods.
    - Start sending telemetry.

```
client.open(function (err) {
  if (err) {
      printErrorFor('open')(err);
  } else {
      console.log('Sending device metadata:\n' + JSON.stringify(deviceMetaData));
      client.sendEvent(new Message(JSON.stringify(deviceMetaData)), printErrorFor('send
metadata'));

      // Create device twin
      client.getTwin(function(err, twin) {
          if (err) {
              console.error('Could not get device twin');
          } else {
              console.log('Device twin created');

              twin.on('properties.desired', function(delta) {
                  console.log('Received new desired properties:');
                  console.log(JSON.stringify(delta));
              });

              // Send reported properties
              twin.properties.reported.update(reportedProperties, function(err) {
                  if (err) throw err;
                  console.log('twin state reported');
              });

              // Register handlers for direct methods
              client.onDeviceMethod('Reboot', onReboot);
              client.onDeviceMethod('InitiateFirmwareUpdate', onInitiateFirmwareUpdate);
          }
      });

      // Start sending telemetry
      var sendInterval = setInterval(function () {
          temperature += generateRandomIncrement();
          externalTemperature += generateRandomIncrement();
          humidity += generateRandomIncrement();

          var data = JSON.stringify({
              'DeviceID': deviceId,
              'Temperature': temperature,
              'Humidity': humidity,
              'ExternalTemperature': externalTemperature
          });

          console.log('Sending device event data:\n' + data);
          client.sendEvent(new Message(data), printErrorFor('send event'));
      }, 5000);

      client.on('error', function (err) {
          printErrorFor('client')(err);
          if (sendInterval) clearInterval(sendInterval);
          client.close(printErrorFor('client.close'));
      });
  }
});
```

15. Save the changes to the **remote_monitoring.js** file.

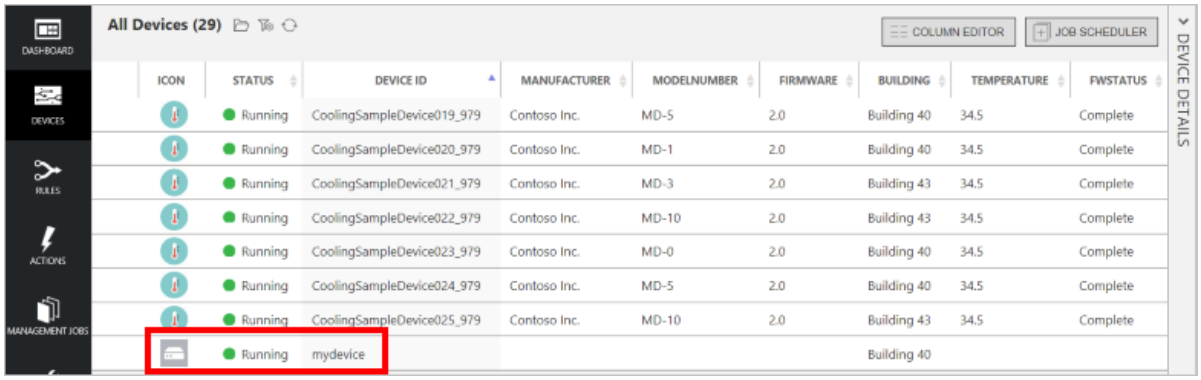
16. Run the following command at a command prompt to launch the sample application:

```
node remote_monitoring.js
```

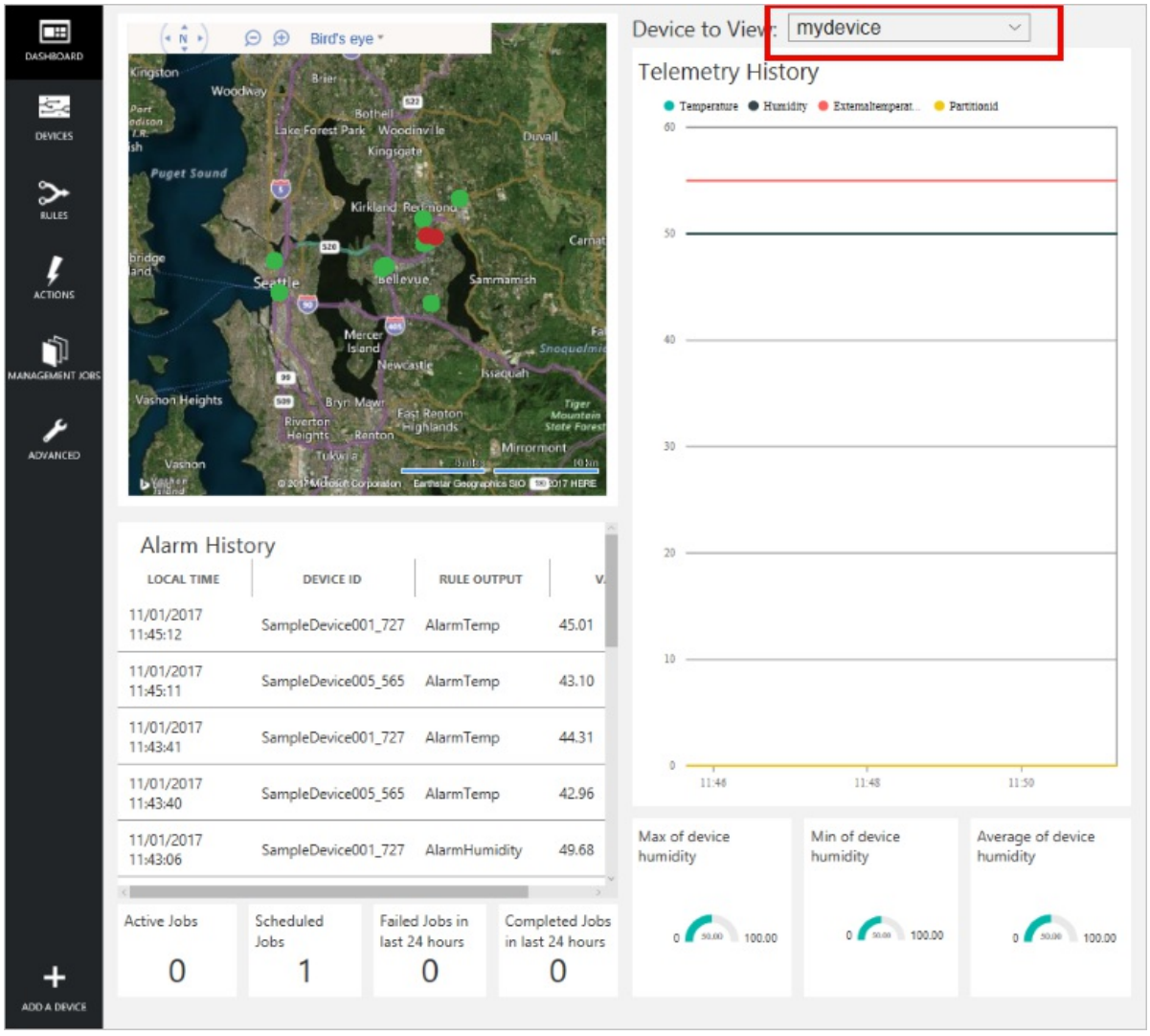
# View device telemetry in the dashboard

The dashboard in the remote monitoring solution enables you to view the telemetry your devices send to IoT Hub.

1. In your browser, return to the remote monitoring solution dashboard, click **Devices** in the left-hand panel to navigate to the **Devices list**.
2. In the **Devices list**, you should see that the status of your device is **Running**. If not, click **Enable Device** in the **Device Details** panel.



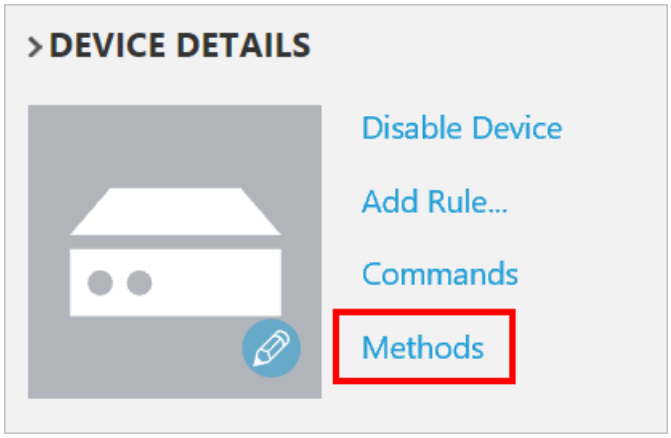

3. Click **Dashboard** to return to the dashboard, select your device in the **Device to View** drop-down to view its telemetry. The telemetry from the sample application is 50 units for internal temperature, 55 units for external temperature, and 50 units for humidity.



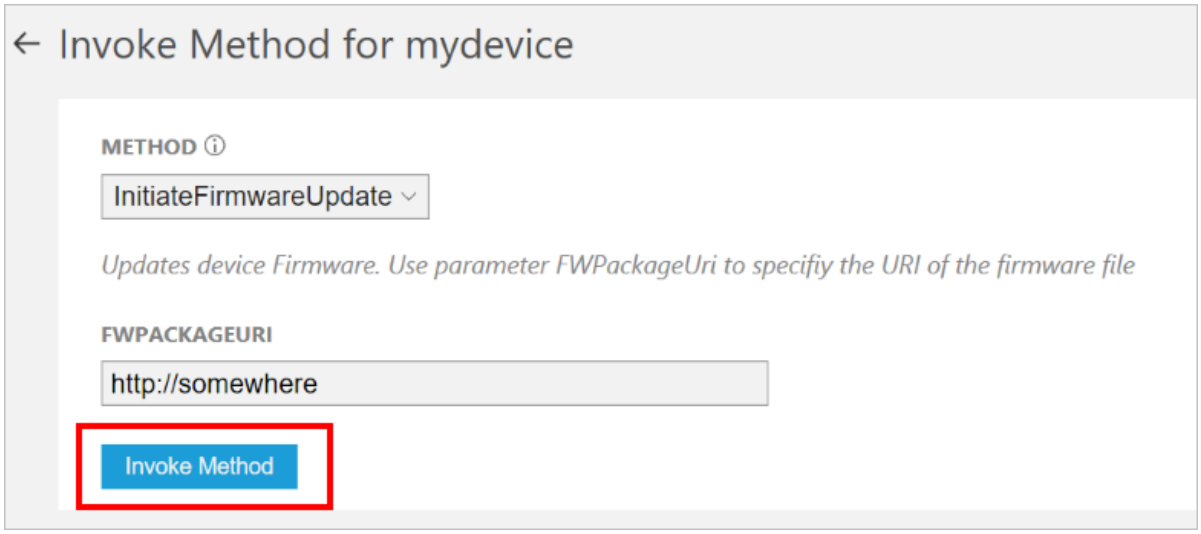

# Invoke a method on your device

The dashboard in the remote monitoring solution enables you to invoke methods on your devices through IoT Hub. For example, in the remote monitoring solution you can invoke a method to simulate rebooting a device.

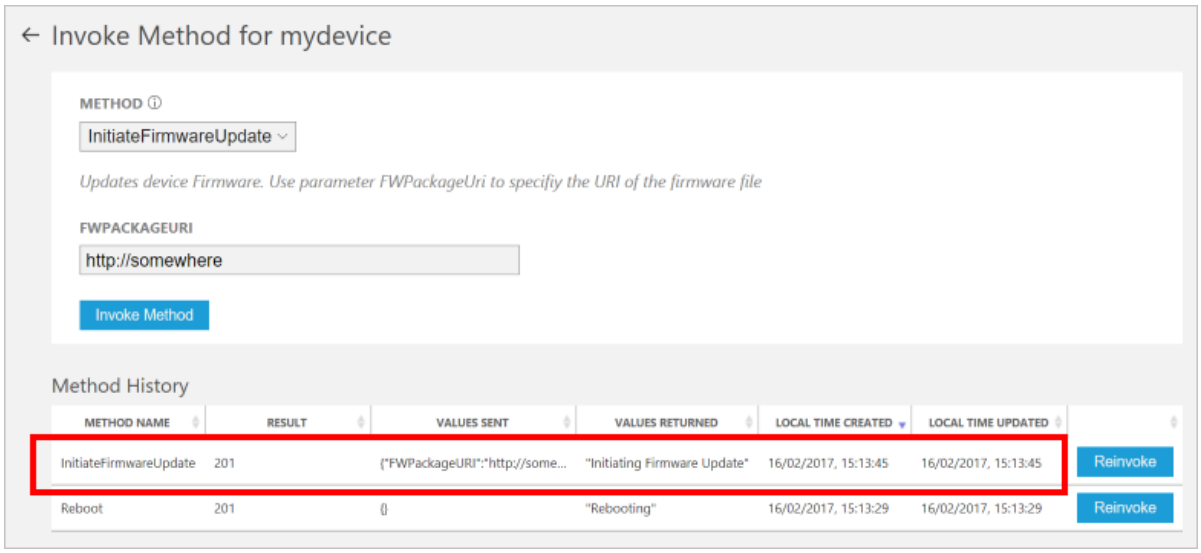
1. In the remote monitoring solution dashboard, click **Devices** in the left-hand panel to navigate to the **Devices list**.
2. Click **Device ID** for your device in the **Devices list**.
3. In the **Device details** panel, click **Methods**.




4. In the **Method** drop-down, select **InitiateFirmwareUpdate**, and then in **FWPACKAGEURI** enter a dummy URL. Click **Invoke Method** to call the method on the device.




5. You see a message in the console running your device code when the device handles the method. The results of the method are added to the history in the solution portal:



# Next steps

The article Customizing preconfigured solutions describes some ways you can extend this sample. Possible extensions include using real sensors and implementing additional commands.

You can learn more about the permissions on the azureiotsuite.com site.

# Tutorial: Connect Logic App to your Azure IoT Suite Remote Monitoring preconfigured solution

6/27/2017 • 5 min to read • Edit Online

The Microsoft Azure IoT Suite remote monitoring preconfigured solution is a great way to get started quickly with an end-to-end feature set that exemplifies an IoT solution. This tutorial walks you through how to add Logic App to your Microsoft Azure IoT Suite remote monitoring preconfigured solution. These steps demonstrate how you can take your IoT solution even further by connecting it to a business process.

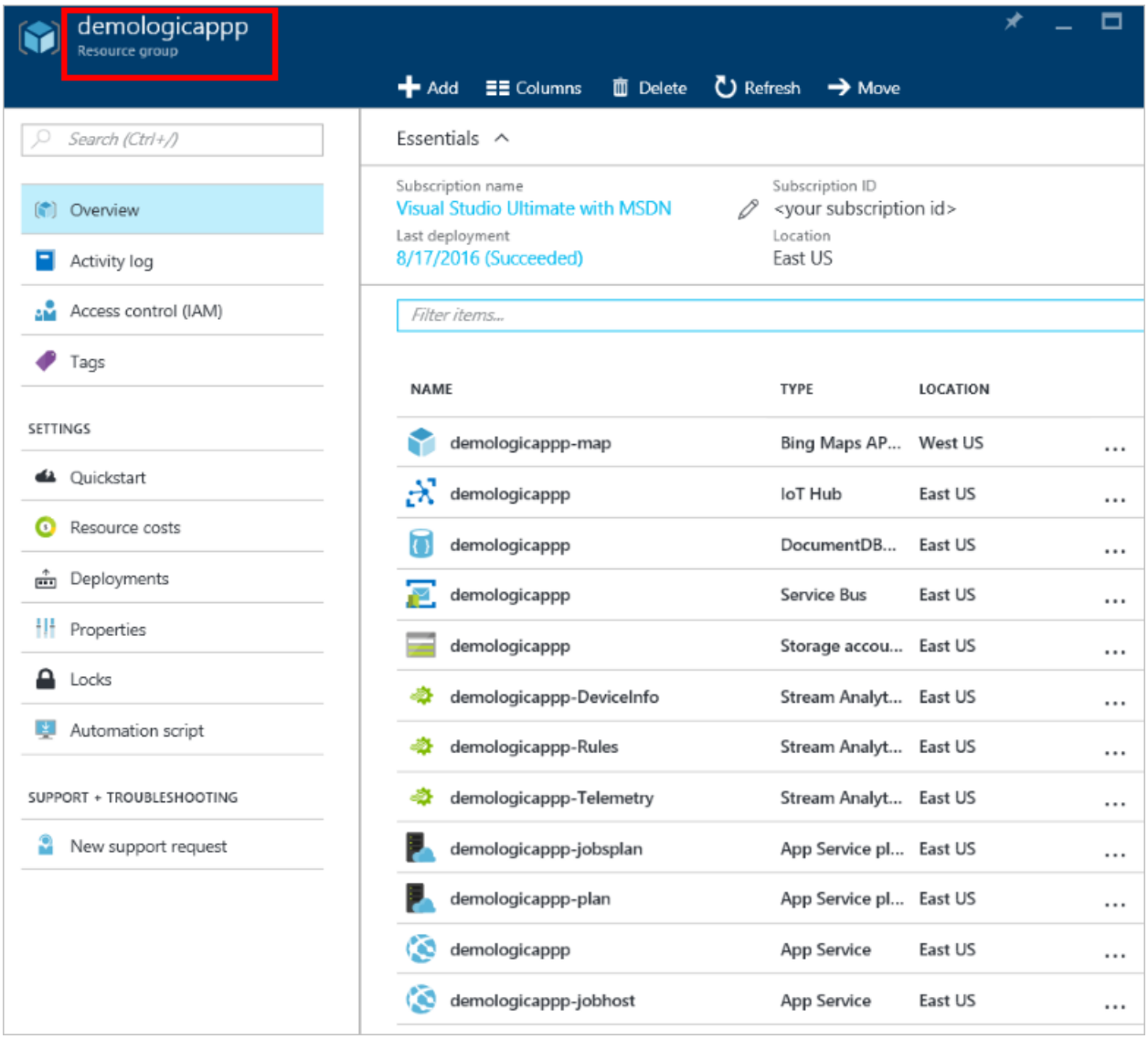
*If you're looking for a walkthrough on how to provision a remote monitoring preconfigured solution, see Tutorial: Get started with the IoT preconfigured solutions.*

Before you start this tutorial, you should:

- Provision the remote monitoring preconfigured solution in your Azure subscription.
- Create a SendGrid account to enable you to send an email that triggers your business process. You can sign up for a free trial account at SendGrid by clicking **Try for Free**. After you have registered for your free trial account, you need to create an API key in SendGrid that grants permissions to send mail. You need this API key later in the tutorial.

To complete this tutorial, you need Visual Studio 2015 or Visual Studio 2017 to modify the actions in the preconfigured solution back end.

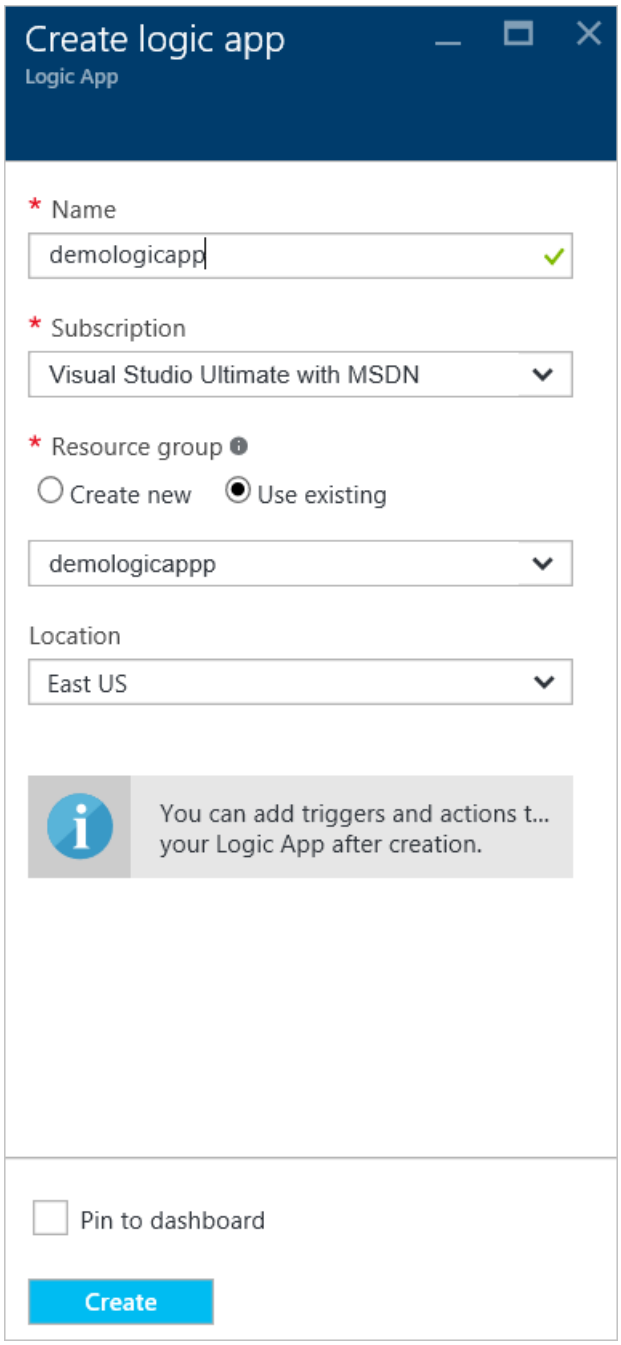
Assuming you've already provisioned your remote monitoring preconfigured solution, navigate to the resource group for that solution in the Azure portal. The resource group has the same name as the solution name you chose when you provisioned your remote monitoring solution. In the resource group, you can see all the provisioned Azure resources for your solution except for the Azure Active Directory application that you can find in the Azure Classic Portal. The following screenshot shows an example **Resource group** blade for a remote monitoring preconfigured solution:



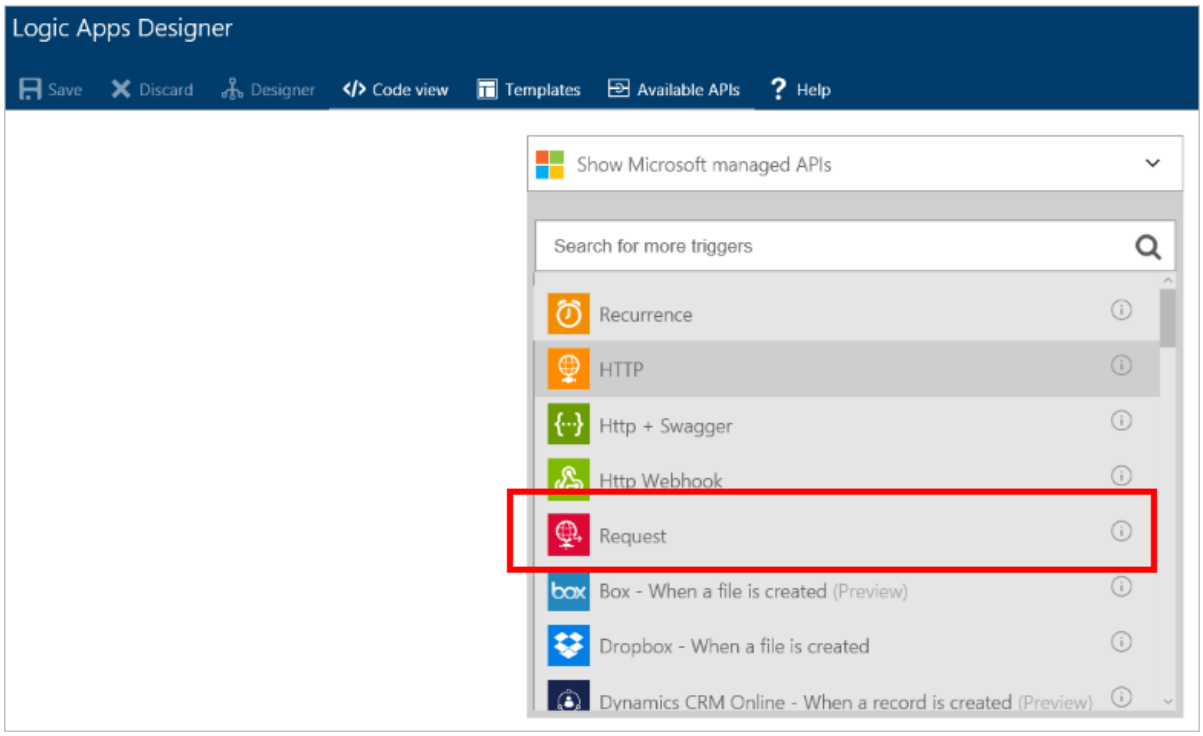

To begin, set up the logic app to use with the preconfigured solution.

## Set up the Logic App

1. Click **Add** at the top of your resource group blade in the Azure portal.
2. Search for **Logic App**, select it and then click **Create**.
3. Fill out the **Name** and use the same **Subscription** and **Resource group** that you used when you provisioned your remote monitoring solution. Click **Create**.




4. When your deployment completes, you can see the Logic App is listed as a resource in your resource group.
5. Click the Logic App to navigate to the Logic App blade, select the **Blank Logic App** template to open the **Logic Apps Designer**.



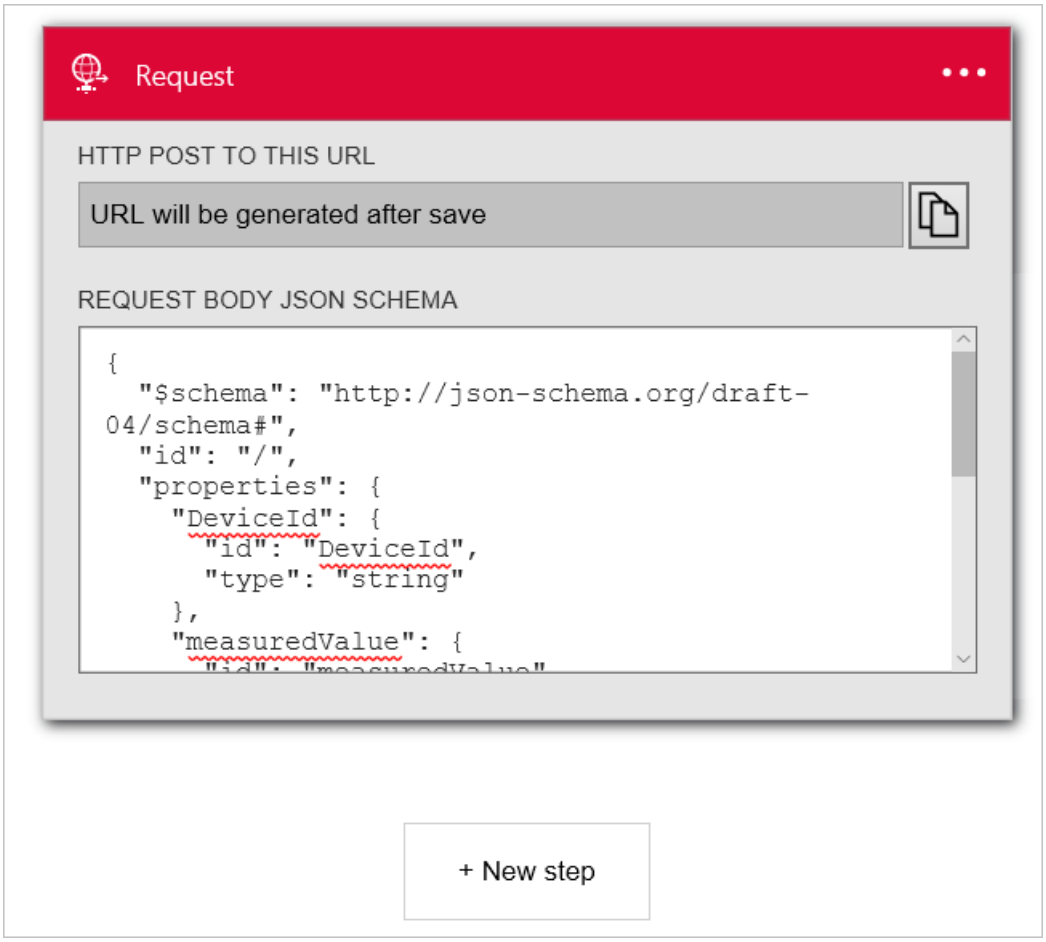

6. Select **Request**. This action specifies that an incoming HTTP request with a specific JSON formatted payload acts as a trigger.
7. Paste the following code into the Request Body JSON Schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "/",
  "properties": {
    "DeviceId": {
      "id": "DeviceId",
      "type": "string"
    },
    "measuredValue": {
      "id": "measuredValue",
      "type": "integer"
    },
    "measurementName": {
      "id": "measurementName",
      "type": "string"
    }
  },
  "required": [
    "DeviceId",
    "measurementName",
    "measuredValue"
  ],
  "type": "object"
}
```

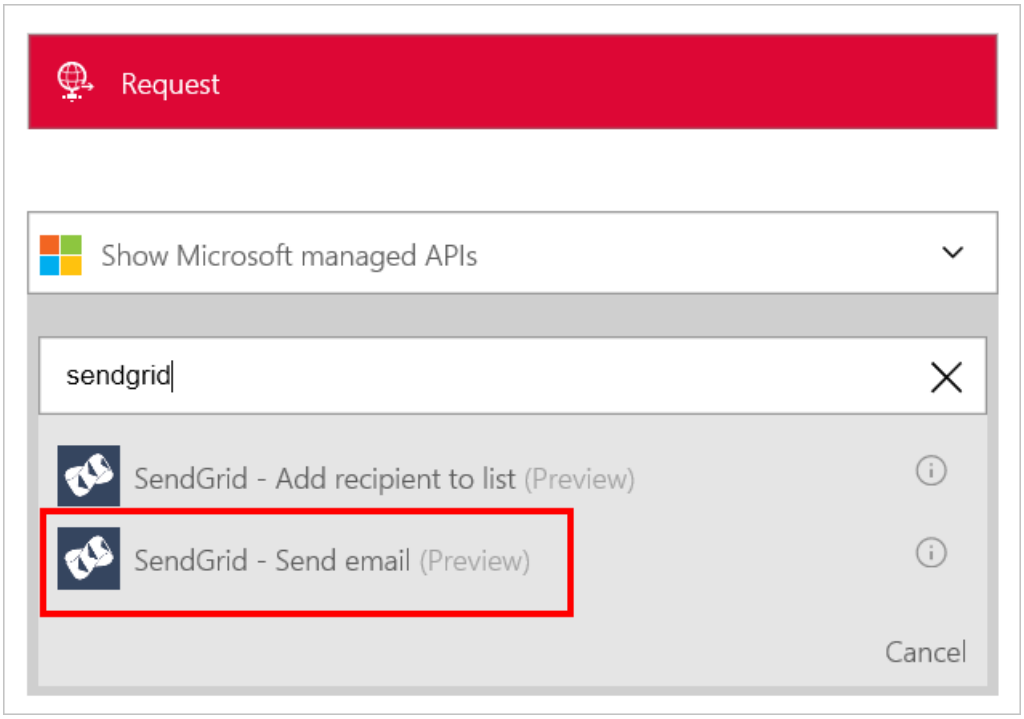
**NOTE**

You can copy the URL for the HTTP post after you save the logic app, but first you must add an action.

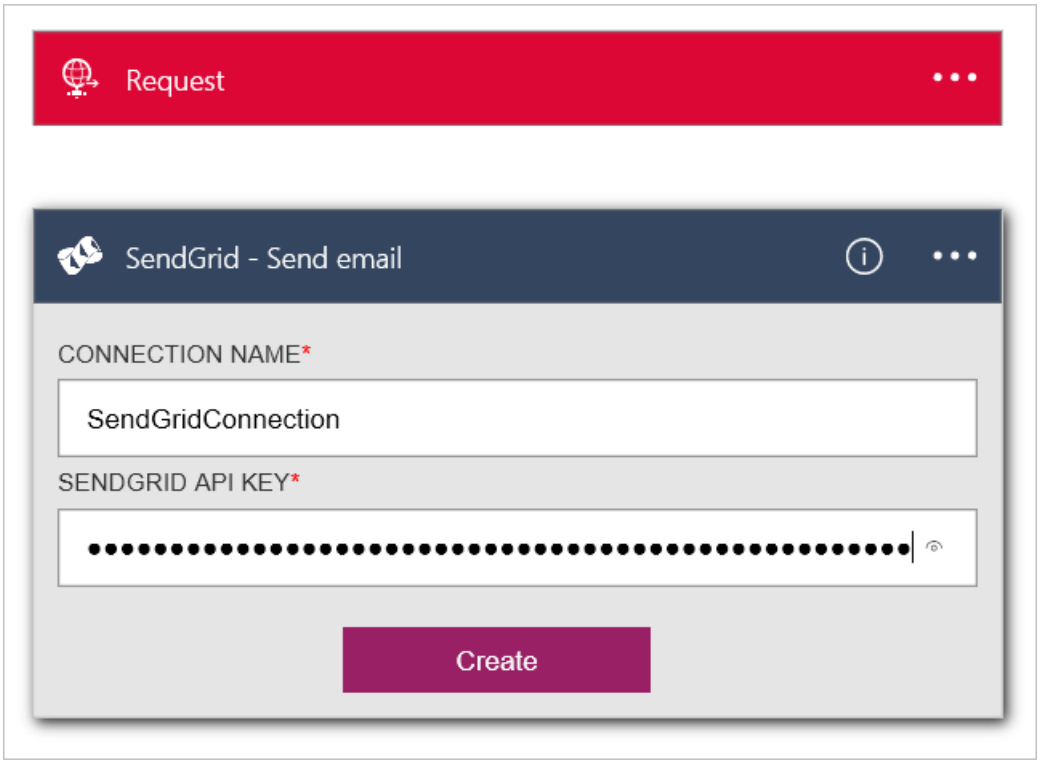
8. Click **+ New step** under your manual trigger. Then click **Add an action**.



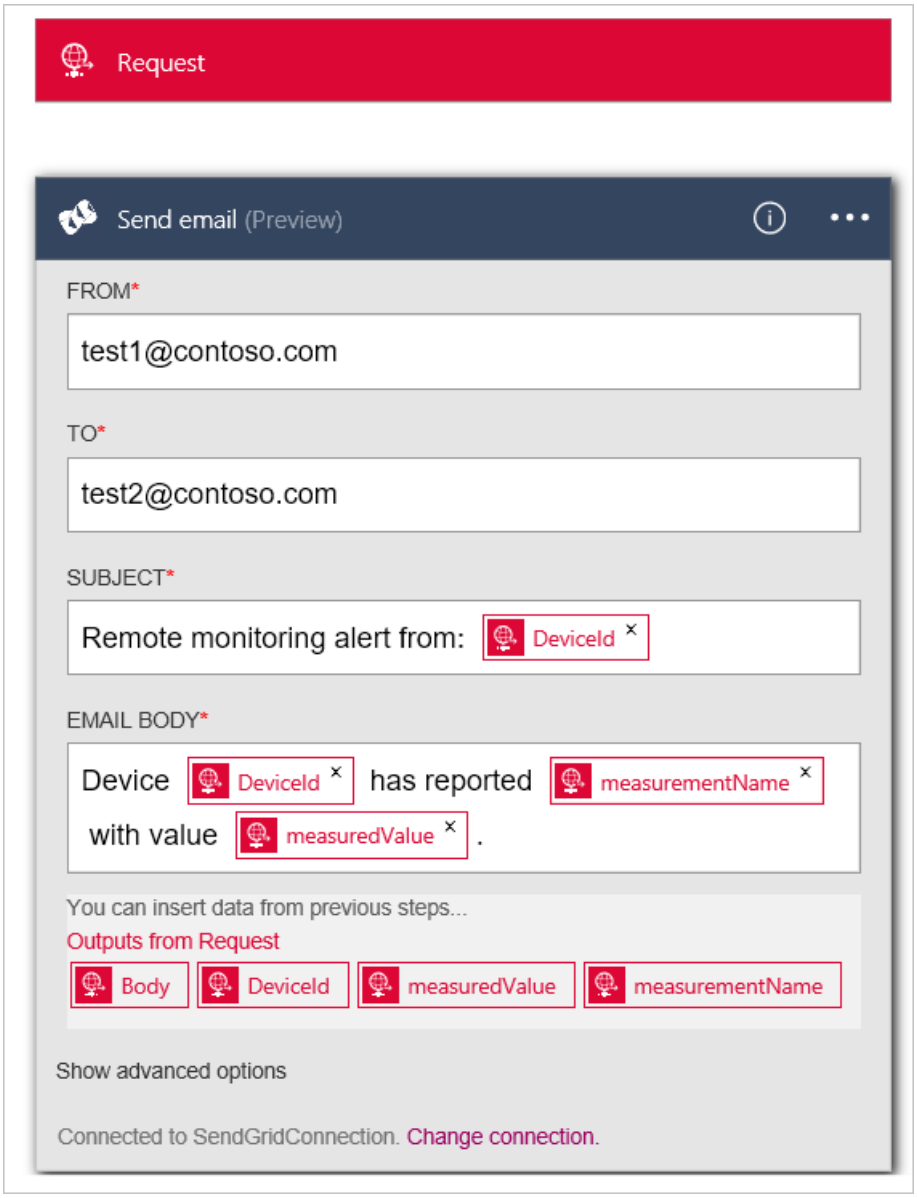

9. Search for **SendGrid - Send email** and click it.




10. Enter a name for the connection, such as **SendGridConnection**, enter the **SendGrid API Key** you created when you set up your SendGrid account, and click **Create**.




11. Add email addresses you own to both the **From** and **To** fields. Add **Remote monitoring alert [DeviceId]** to the **Subject** field. In the **Email Body** field, add **Device [DeviceId] has reported [measurementName] with value [measuredValue].** You can add **[DeviceId]**, **[measurementName]**, and **[measuredValue]** by clicking in the **You can insert data from previous steps** section.




12. Click **Save** in the top menu.
13. Click the **Request** trigger and copy the **Http Post to this URL** value. You need this URL later in this tutorial.

> **NOTE**
> Logic Apps enable you to run many different types of action including actions in Office 365.

## Set up the EventProcessor Web Job

In this section, you connect your preconfigured solution to the Logic App you created. To complete this task, you add the URL to trigger the Logic App to the action that fires when a device sensor value exceeds a threshold.

1. Use your git client to clone the latest version of the azure-iot-remote-monitoring github repository. For example:

```
git clone https://github.com/Azure/azure-iot-remote-monitoring.git
```

2. In Visual Studio, open the **RemoteMonitoring.sln** from the local copy of the repository.
3. Open the **ActionRepository.cs** file in the **Infrastructure\Repository** folder.
4. Update the **actionIds** dictionary with the **Http Post to this URL** you noted from your Logic App as follows:

```
private Dictionary<string,string> actionIds = new Dictionary<string, string>()
{
    { "Send Message", "<Http Post to this URL>" },
    { "Raise Alarm", "<Http Post to this URL>" }
};
```

5. Save the changes in solution and exit Visual Studio.

## Deploy from the command line

In this section, you deploy your updated version of the remote monitoring solution to replace the version currently running in Azure.

1. Following the dev set-up instructions to set up your environment for deployment.
2. To deploy locally, follow the local deployment instructions.
3. To deploy to the cloud and update your existing cloud deployment, follow the cloud deployment instructions. Use the name of your original deployment as the deployment name. For example if the original deployment was called **demologicapp**, use the following command:

   ```
   build.cmd cloud release demologicapp
   ```

   When the build script runs, be sure to use the same Azure account, subscription, region, and Active Directory instance you used when you provisioned the solution.

## See your Logic App in action

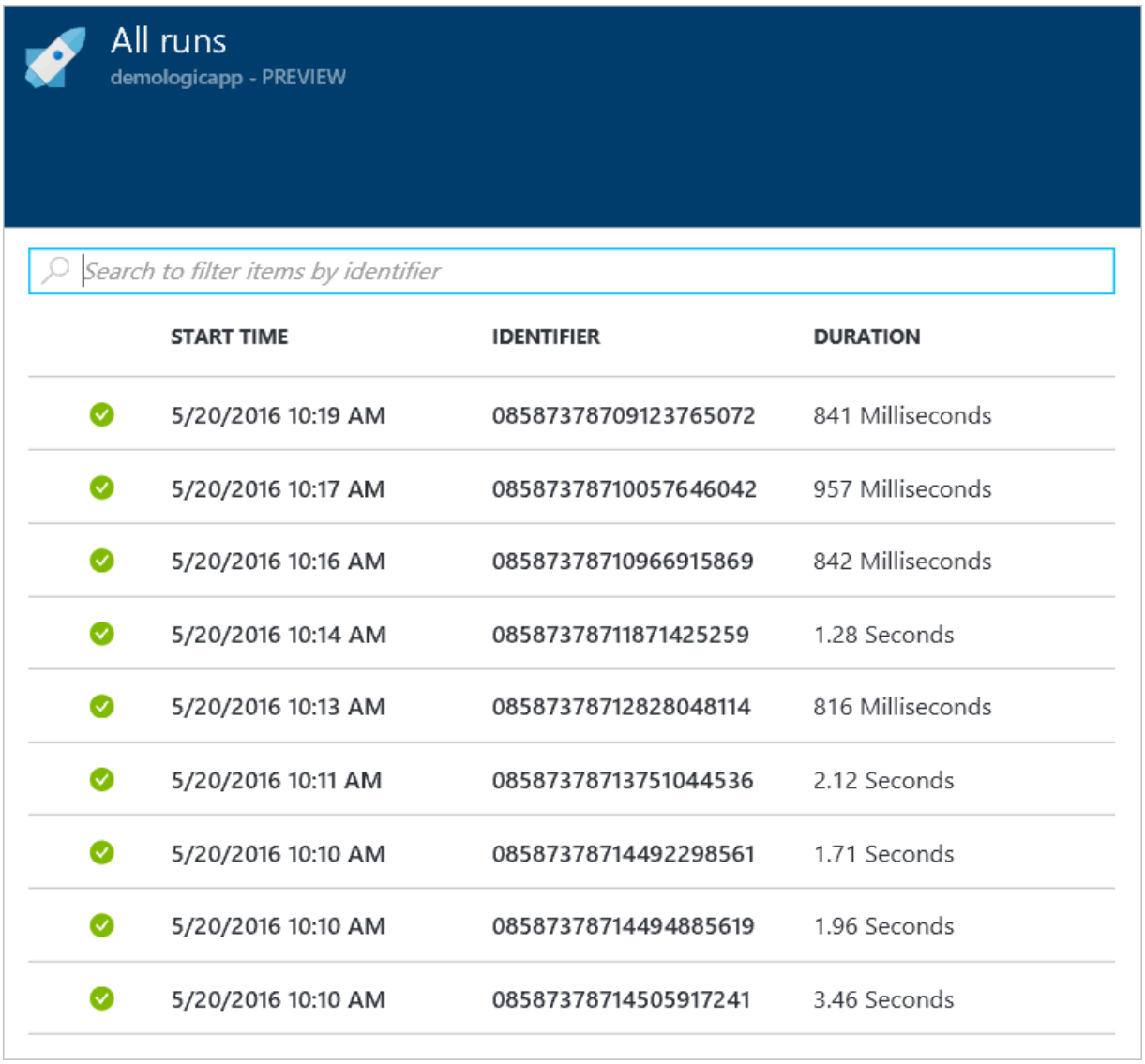
The remote monitoring preconfigured solution has two rules set up by default when you provision a solution. Both rules are on the **SampleDevice001** device:

- Temperature > 38.00
- Humidity > 48.00

The temperature rule triggers the **Raise Alarm** action and the Humidity rule triggers the **SendMessage** action. Assuming you used the same URL for both actions the **ActionRepository** class, your logic app triggers for either rule. Both rules use SendGrid to send an email to the **To** address with details of the alert.

> **NOTE**
> The Logic App continues to trigger every time the threshold is met. To avoid unnecessary emails, you can either disable the rules in your solution portal or disable the Logic App in the Azure portal.

In addition to receiving emails, you can also see when the Logic App runs in the portal:




## Next steps

Now that you've used a Logic App to connect the preconfigured solution to a business process, you can learn more about the options for customizing the preconfigured solutions:

- Use dynamic telemetry with the remote monitoring preconfigured solution
- Device information metadata in the remote monitoring preconfigured solution

# Customize a preconfigured solution

6/27/2017 • 9 min to read • Edit Online

The preconfigured solutions provided with the Azure IoT Suite demonstrate the services within the suite working together to deliver an end-to-end solution. From this starting point, there are various places in which you can extend and customize the solution for specific scenarios. The following sections describe these common customization points.

## Find the source code

The source code for the preconfigured solutions is available on GitHub in the following repositories:

- Remote Monitoring: https://www.github.com/Azure/azure-iot-remote-monitoring
- Predictive Maintenance: https://github.com/Azure/azure-iot-predictive-maintenance
- Connected factory: https://github.com/Azure/azure-iot-connected-factory

The source code for the preconfigured solutions is provided to demonstrate the patterns and practices used to implement the end-to-end functionality of an IoT solution using Azure IoT Suite. You can find more information about how to build and deploy the solutions in the GitHub repositories.

## Change the preconfigured rules

The remote monitoring solution includes three Azure Stream Analytics jobs to handle device information, telemetry, and rules logic in the solution.

The three stream analytics jobs and their syntax are described in depth in the Remote monitoring preconfigured solution walkthrough.

You can edit these jobs directly to alter the logic, or add logic specific to your scenario. You can find the Stream Analytics jobs as follows:

1. Go to Azure portal.
2. Navigate to the resource group with the same name as your IoT solution.
3. Select the Azure Stream Analytics job you'd like to modify.
4. Stop the job by selecting **Stop** in the set of commands.
5. Edit the inputs, query, and outputs.

   A simple modification is to change the query for the **Rules** job to use a **"<"** instead of a **">"**. The solution portal still shows **">"** when you edit a rule, but notice how the behavior is flipped due to the change in the underlying job.

6. Start the job

> **NOTE**
> The remote monitoring dashboard depends on specific data, so altering the jobs can cause the dashboard to fail.

## Add your own rules

In addition to changing the preconfigured Azure Stream Analytics jobs, you can use the Azure portal to add new jobs or add new queries to existing jobs.

# Customize devices

One of the most common extension activities is working with devices specific to your scenario. There are several methods for working with devices. These methods include altering a simulated device to match your scenario, or using the IoT Device SDK to connect your physical device to the solution.

For a step-by-step guide to adding devices, see the Iot Suite Connecting Devices article and the remote monitoring C SDK Sample. This sample is designed to work with the remote monitoring preconfigured solution.

### Create your own simulated device

Included in the remote monitoring solution source code, is a .NET simulator. This simulator is the one provisioned as part of the solution and you can alter it to send different metadata, telemetry, and respond to different commands and methods.

The preconfigured simulator in the remote monitoring preconfigured solution simulates a cooler device that emits temperature and humidity telemetry. You can modify the simulator in the Simulator.WebJob project when you've forked the GitHub repository.

### Available locations for simulated devices

The default set of locations is in Seattle/Redmond, Washington, United States of America. You can change these locations in SampleDeviceFactory.cs.

### Add a desired property update handler to the simulator

You can set a value for a desired property for a device in the solution portal. It is the responsibility of the device to handle the property change request when the device retrieves the desired property value. To add support for a property value change through a desired property, you need to add a handler to the simulator.

The simulator contains handlers for the **SetPointTemp** and **TelemetryInterval** properties that you can update by setting desired values in the solution portal.

The following example shows the handler for the **SetPointTemp** desired property in the **CoolerDevice** class:

```
protected async Task OnSetPointTempUpdate(object value)
{
    var telemetry = _telemetryController as ITelemetryWithSetPointTemperature;
    telemetry.SetPointTemperature = Convert.ToDouble(value);

    await SetReportedPropertyAsync(SetPointTempPropertyName, telemetry.SetPointTemperature);
}
```

This method updates the telemetry point temperature and then reports the change back to IoT Hub by setting a reported property.

You can add your own handlers for your own properties by following the pattern in the preceding example.

You must also bind the desired property to the handler as shown in the following example from the **CoolerDevice** constructor:

```
_desiredPropertyUpdateHandlers.Add(SetPointTempPropertyName, OnSetPointTempUpdate);
```

Note that **SetPointTempPropertyName** is a constant defined as "Config.SetPointTemp".

### Add support for a new method to the simulator

You can customize the simulator to add support for a new method (direct method). There are two key steps required:

- The simulator must notify the IoT hub in the preconfigured solution with details of the method.

- The simulator must include code to handle the method call when you invoke it from the **Device details** panel in the solution explorer or through a job.

The remote monitoring preconfigured solution uses *reported properties* to send details of supported methods to IoT hub. The solution back end maintains a list of all the methods supported by each device along with a history of method invocations. You can view this information about devices and invoke methods in the solution portal.

To notify the IoT hub that a device supports a method, the device must add details of the method to the **SupportedMethods** node in the reported properties:

```
"SupportedMethods": {
  "<method signature>": "<method description>",
  "<method signature>": "<method description>"
}
```

The method signature has the following format:
`<method name>--<parameter #0 name>-<parameter #1 type>-...-<parameter #n name>-<parameter #n type>`. For example, to specify the **InitiateFirmwareUpdate** method expects a string parameter named **FwPackageURI**, use the following method signature:

```
InitiateFirmwareUpate--FwPackageURI-string: "description of method"
```

For a list of supported parameter types, see the **CommandTypes** class in the Infrastructure project.

To delete a method, set the method signature to `null` in the reported properties.

> **NOTE**
> The solution back end only updates information about supported methods when it receives a *device information* message from the device.

The following code sample from the **SampleDeviceFactory** class in the Common project shows how to add a method to the list of **SupportedMethods** in the reported properties sent by the device:

```
device.Commands.Add(new Command(
    "InitiateFirmwareUpdate",
    DeliveryType.Method,
    "Updates device Firmware. Use parameter 'FwPackageUri' to specifiy the URI of the firmware file, e.g.
https://iotrmassets.blob.core.windows.net/firmwares/FW20.bin",
    new[] { new Parameter("FwPackageUri", "string") }
));
```

This code snippet adds details of the **InitiateFirmwareUpdate** method including text to display in the solution portal and details of the required method parameters.

The simulator sends reported properties, including the list of supported methods, to IoT Hub when the simulator starts.

Add a handler to the simulator code for each method it supports. You can see the existing handlers in the **CoolerDevice** class in the Simulator.WebJob project. The following example shows the handler for **InitiateFirmwareUpdate** method:

```
public async Task<MethodResponse> OnInitiateFirmwareUpdate(MethodRequest methodRequest, object userContext)
{
    if (_deviceManagementTask != null && !_deviceManagementTask.IsCompleted)
    {
        return await Task.FromResult(BuildMethodRespose(new
        {
            Message = "Device is busy"
        }, 409));
    }

    try
    {
        var operation = new FirmwareUpdate(methodRequest);
        _deviceManagementTask = operation.Run(Transport).ContinueWith(async task =>
        {
            // after firmware completed, we reset telemetry
            var telemetry = _telemetryController as ITelemetryWithTemperatureMeanValue;
            if (telemetry != null)
            {
                telemetry.TemperatureMeanValue = 34.5;
            }

            await UpdateReportedTemperatureMeanValue();
        });

        return await Task.FromResult(BuildMethodRespose(new
        {
            Message = "FirmwareUpdate accepted",
            Uri = operation.Uri
        }));
    }
    catch (Exception ex)
    {
        return await Task.FromResult(BuildMethodRespose(new
        {
            Message = ex.Message
        }, 400));
    }
}
```

Method handler names must start with `On` followed by the name of the method. The **methodRequest** parameter contains any parameters passed with the method invocation from the solution back end. The return value must be of type **Task<MethodResponse>**. The **BuildMethodResponse** utility method helps you create the return value.

Inside the method handler, you could:

- Start an asynchronous task.
- Retrieve desired properties from the *device twin* in IoT Hub.
- Update a single reported property using the **SetReportedPropertyAsync** method in the **CoolerDevice** class.
- Update multiple reported properties by creating a **TwinCollection** instance and calling the **Transport.UpdateReportedPropertiesAsync** method.

The preceding firmware update example performs the following steps:

- Checks the device is able to accept the firmware update request.
- Asynchronously initiates the firmware update operation and resets the telemetry when the operation is complete.
- Immediately returns the "FirmwareUpdate accepted" message to indicate the request was accepted by the device.

**Build and use your own (physical) device**

The Azure IoT SDKs provide libraries for connecting numerous device types (languages and operating systems) into IoT solutions.

# Modify dashboard limits

**Number of devices displayed in dashboard dropdown**

The default is 200. You can change this number in DashboardController.cs.

**Number of pins to display in Bing Map control**

The default is 200. You can change this number in TelemetryApiController.cs.

**Time period of telemetry graph**

The default is 10 minutes. You can change this value in TelmetryApiController.cs.

# Manually set up application roles

The following procedure describes how to add **Admin** and **ReadOnly** application roles to a preconfigured solution. Note that preconfigured solutions provisioned from the azureiotsuite.com site already include the **Admin** and **ReadOnly** roles.

Members of the **ReadOnly** role can see the dashboard and the device list, but are not allowed to add devices, change device attributes, or send commands. Members of the **Admin** role have full access to all the functionality in the solution.

1. Go to the Azure classic portal.
2. Select **Active Directory**.
3. Click the name of the AAD tenant you used when you provisioned your solution.
4. Click **Applications**.
5. Click the name of the application that matches your preconfigured solution name. If you don't see your application in the list, select **Applications my company owns** in the **Show** dropdown and click the check mark.
6. At the bottom of the page, click **Manage Manifest** and then **Download Manifest**.
7. This procedure downloads a .json file to your local machine. Open this file for editing in a text editor of your choice.
8. On the third line of the .json file, you can see:

   ```
   "appRoles" : [],
   ```

   Replace this line with the following code:

```
"appRoles": [
{
"allowedMemberTypes": [
"User"
],
"description": "Administrator access to the application",
"displayName": "Admin",
"id": "a400a00b-f67c-42b7-ba9a-f73d8c67e433",
"isEnabled": true,
"value": "Admin"
},
{
"allowedMemberTypes": [
"User"
],
"description": "Read only access to device information",
"displayName": "Read Only",
"id": "e5bbd0f5-128e-4362-9dd1-8f253c6082d7",
"isEnabled": true,
"value": "ReadOnly"
} ],
```

9. Save the updated .json file (you can overwrite the existing file).
10. In the Azure classic portal, at the bottom of the page, select **Manage Manifest** then **Upload Manifest** to upload the .json file you saved in the previous step.
11. You have now added the **Admin** and **ReadOnly** roles to your application.
12. To assign one of these roles to a user in your directory, see Permissions on the azureiotsuite.com site.

## Feedback

Do you have a customization you'd like to see covered in this document? Add feature suggestions to User Voice, or comment on this article.

## Next steps

To learn more about the options for customizing the preconfigured solutions, see:

- Connect Logic App to your Azure IoT Suite Remote Monitoring preconfigured solution
- Use dynamic telemetry with the remote monitoring preconfigured solution
- Device information metadata in the remote monitoring preconfigured solution
- Customize how the connected factory solution displays data from your OPC UA servers

# Use dynamic telemetry with the remote monitoring preconfigured solution

8/24/2017 • 6 min to read • Edit Online

Dynamic telemetry enables you to visualize any telemetry sent to the remote monitoring preconfigured solution. The simulated devices that deploy with the preconfigured solution send temperature and humidity telemetry, which you can visualize on the dashboard. If you customize existing simulated devices, create new simulated devices, or connect physical devices to the preconfigured solution you can send other telemetry values such as the external temperature, RPM, or windspeed. You can then visualize this additional telemetry on the dashboard.

This tutorial uses a simple Node.js simulated device that you can easily modify to experiment with dynamic telemetry.

To complete this tutorial, you'll need:

- An active Azure subscription. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.
- Node.js version 0.12.x or later.

You can complete this tutorial on any operating system, such as Windows or Linux, where you can install Node.js.

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

## Configure the Node.js simulated device

1. On the remote monitoring dashboard, click **+ Add a device** and then add a *custom device*. Make a note of the IoT Hub hostname, device id, and device key. You need them later in this tutorial when you prepare the remote_monitoring.js device client application.

2. Ensure that Node.js version 0.12.x or later is installed on your development machine. Run `node --version` at a command prompt or in a shell to check the version. For information about using a package manager to install Node.js on Linux, see Installing Node.js via package manager.
3. When you have installed Node.js, clone the latest version of the azure-iot-sdk-node repository to your development machine. Always use the **master** branch for the latest version of the libraries and samples.
4. From your local copy of the azure-iot-sdk-node repository, copy the following two files from the node/device/samples folder to an empty folder on your development machine:

   - packages.json
   - remote_monitoring.js
5. Open the remote_monitoring.js file and look for the following variable definition:

   ```
   var connectionString = "[IoT Hub device connection string]";
   ```
6. Replace **[IoT Hub device connection string]** with your device connection string. Use the values for your IoT Hub hostname, device id, and device key that you made a note of in step 1. A device connection string has the following format:

   ```
   HostName={your IoT Hub hostname};DeviceId={your device id};SharedAccessKey={your device key}
   ```

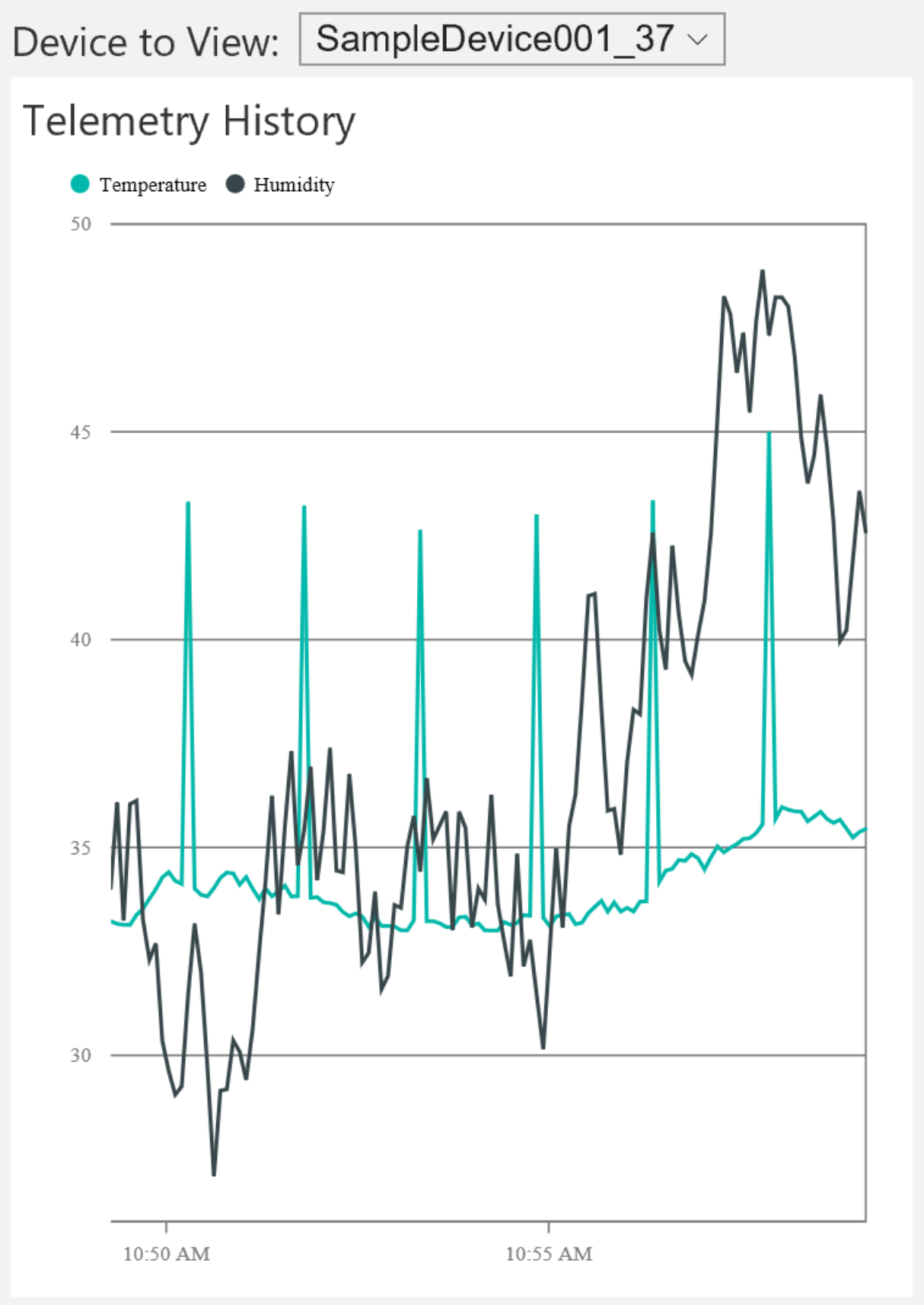
   If your IoT Hub hostname is **contoso** and your device id is **mydevice**, your connection string looks like the following snippet:

   ```
   var connectionString = "HostName=contoso.azure-devices.net;DeviceId=mydevice;SharedAccessKey=2s ... =="
   ```
7. Save the file. Run the following commands in a shell or command prompt in the folder that contains these files to install the necessary packages and then run the sample application:

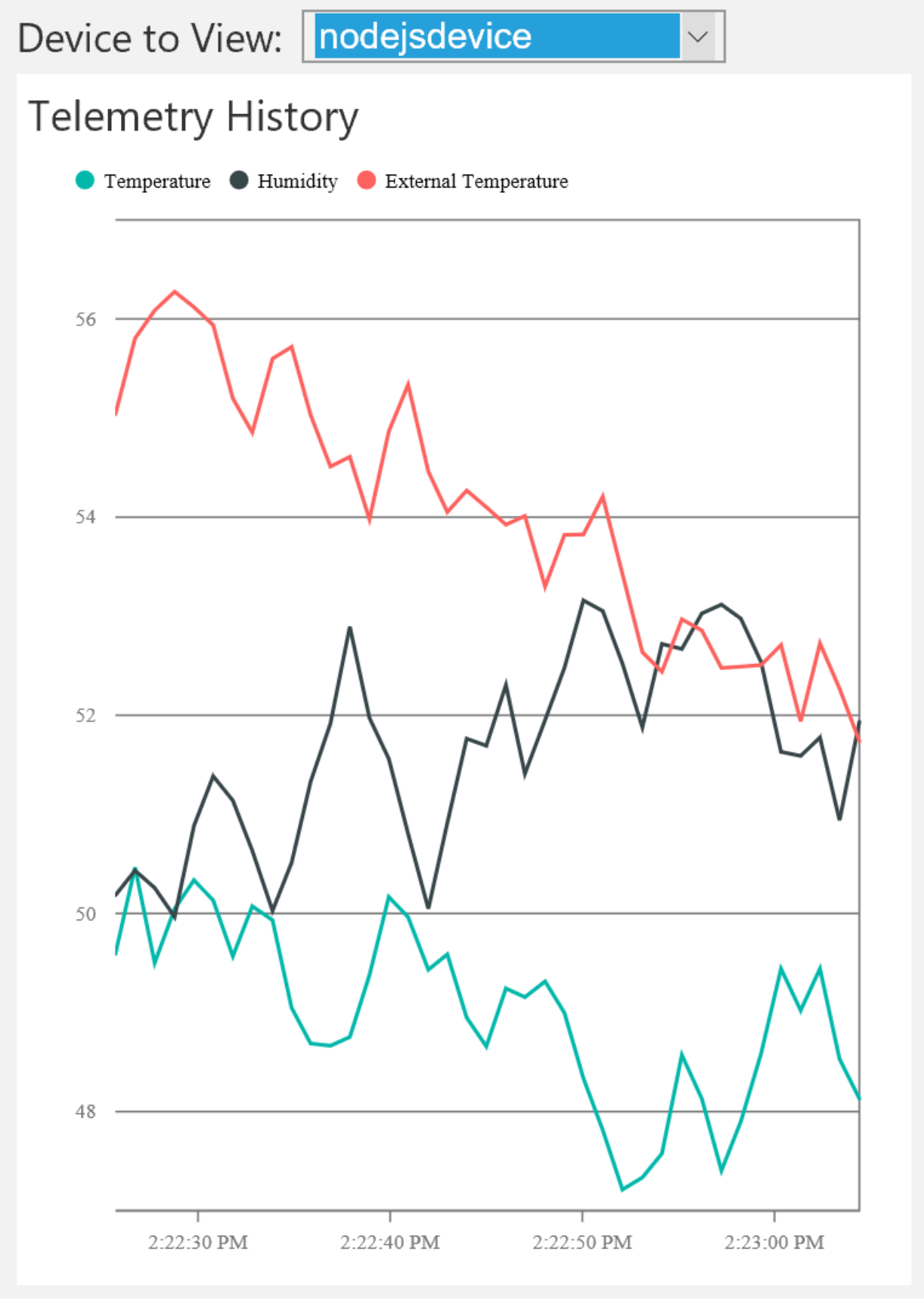
   ```
   npm install
   node remote_monitoring.js
   ```

## Observe dynamic telemetry in action

The dashboard shows the temperature and humidity telemetry from the existing simulated devices:




If you select the Node.js simulated device you ran in the previous section, you see temperature, humidity, and external temperature telemetry:




The remote monitoring solution automatically detects the additional external temperature telemetry type and adds it to the chart on the dashboard.

## Add a telemetry type

The next step is to replace the telemetry generated by the Node.js simulated device with a new set of values:

1. Stop the Node.js simulated device by typing **Ctrl+C** in your command prompt or shell.
2. In the remote_monitoring.js file, you can see the base data values for the existing temperature, humidity,

and external temperature telemetry. Add a base data value for **rpm** as follows:

```
// Sensors data
var temperature = 50;
var humidity = 50;
var externalTemperature = 55;
var rpm = 200;
```

3. The Node.js simulated device uses the **generateRandomIncrement** function in the remote_monitoring.js file to add a random increment to the base data values. Randomize the **rpm** value by adding a line of code after the existing randomizations as follows:

```
temperature += generateRandomIncrement();
externalTemperature += generateRandomIncrement();
humidity += generateRandomIncrement();
rpm += generateRandomIncrement();
```

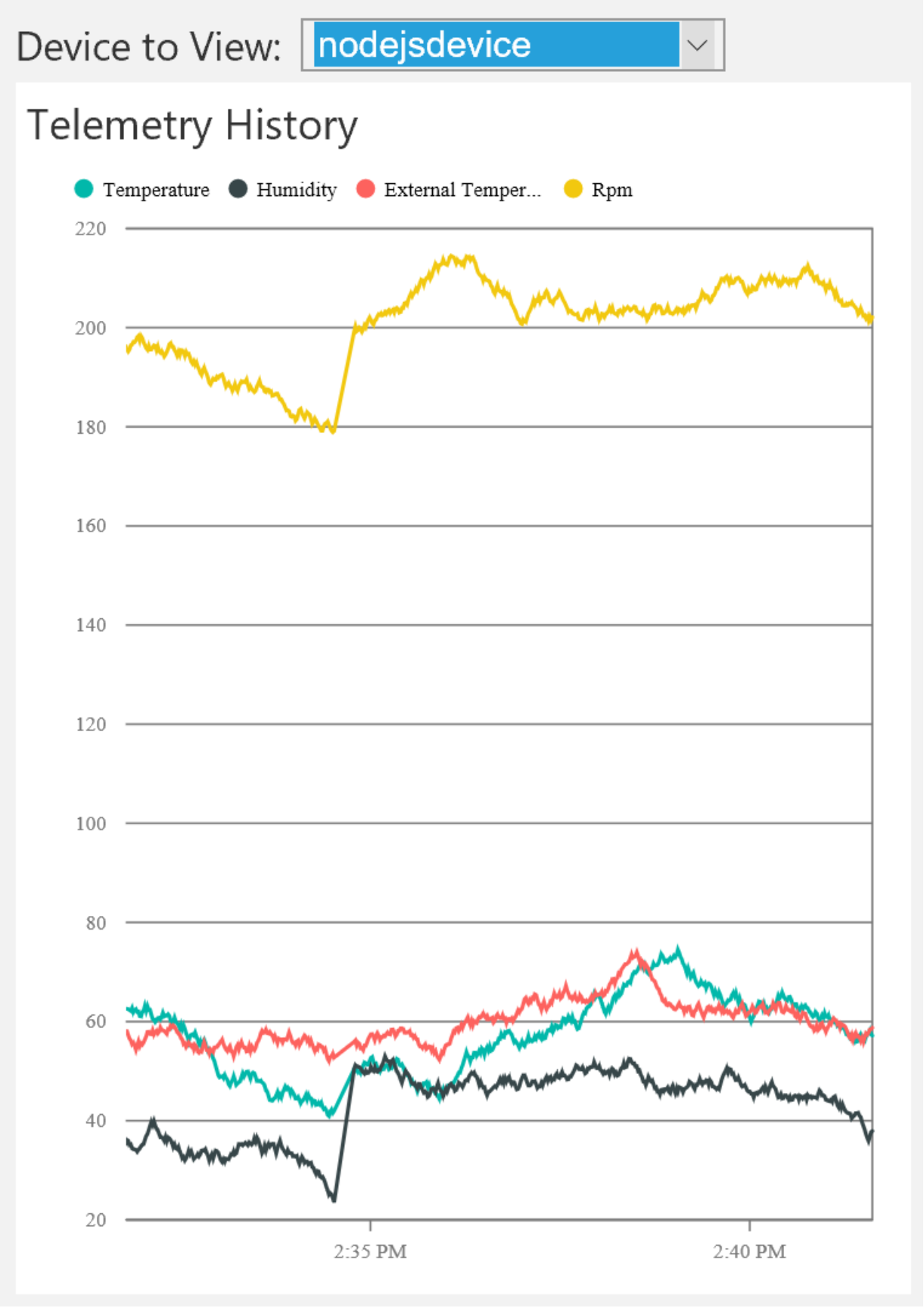
4. Add the new rpm value to the JSON payload the device sends to IoT Hub:

```
var data = JSON.stringify({
  'DeviceID': deviceId,
  'Temperature': temperature,
  'Humidity': humidity,
  'ExternalTemperature': externalTemperature,
  'RPM': rpm
});
```

5. Run the Node.js simulated device using the following command:

```
node remote_monitoring.js
```

6. Observe the new RPM telemetry type that displays on the chart in the dashboard:




**NOTE**

You may need to disable and then enable the Node.js device on the **Devices** page in the dashboard to see the change immediately.

## Customize the dashboard display

The **Device-Info** message can include metadata about the telemetry the device can send to IoT Hub. This metadata can specify the telemetry types the device sends. Modify the **deviceMetaData** value in the remote_monitoring.js file to include a **Telemetry** definition following the **Commands** definition. The following code snippet shows the **Commands** definition (be sure to add a `,` after the **Commands** definition):

```
'Commands': [{
  'Name': 'SetTemperature',
  'Parameters': [{
    'Name': 'Temperature',
    'Type': 'double'
  }]
},
{
  'Name': 'SetHumidity',
  'Parameters': [{
    'Name': 'Humidity',
    'Type': 'double'
  }]
}],
'Telemetry': [{
  'Name': 'Temperature',
  'Type': 'double'
},
{
  'Name': 'Humidity',
  'Type': 'double'
},
{
  'Name': 'ExternalTemperature',
  'Type': 'double'
}]
```

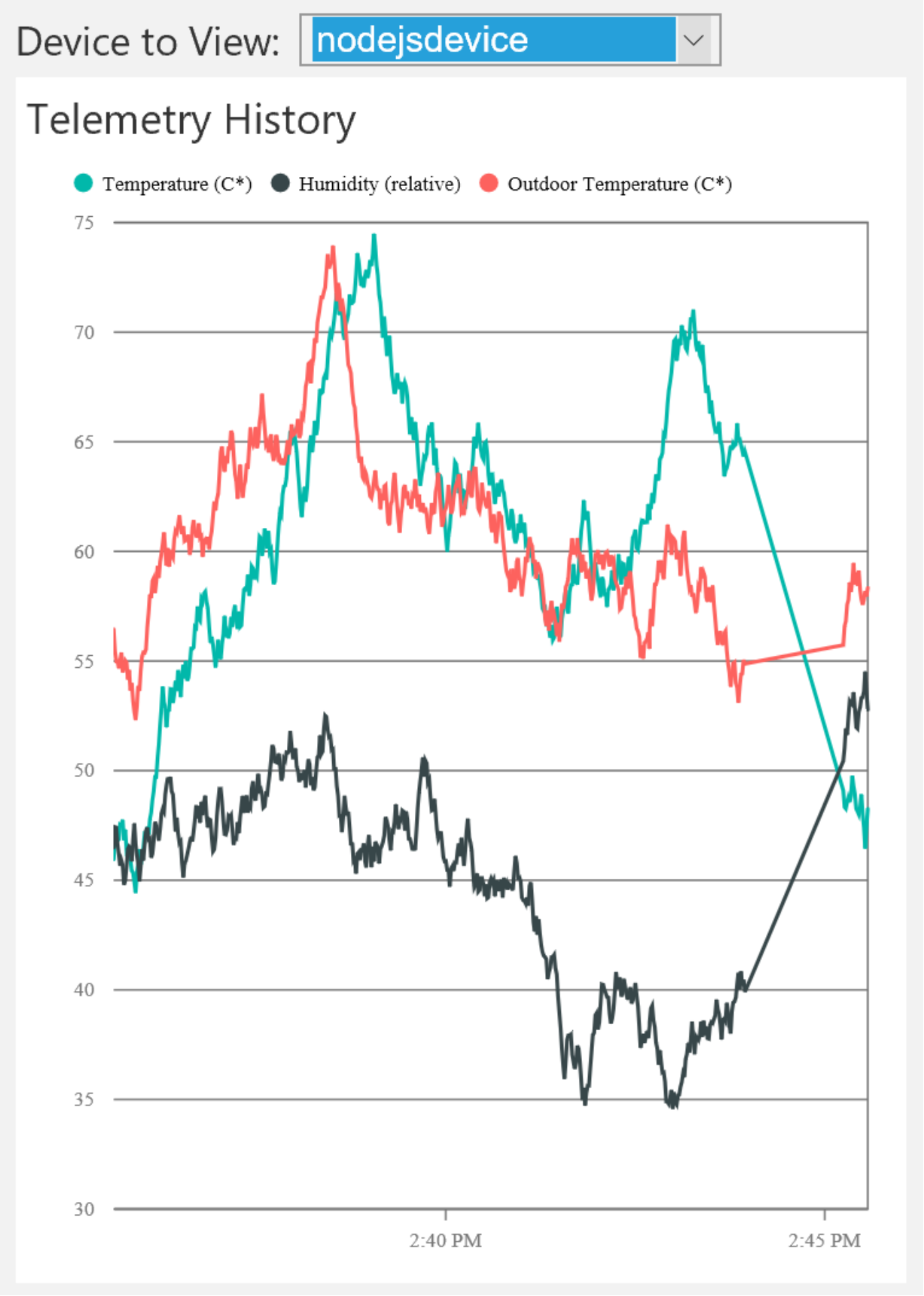
**NOTE**

The remote monitoring solution uses a case-insensitive match to compare the metadata definition with data in the telemetry stream.

Adding a **Telemetry** definition as shown in the preceding code snippet does not change the behavior of the dashboard. However, the metadata can also include a **DisplayName** attribute to customize the display in the dashboard. Update the **Telemetry** metadata definition as shown in the following snippet:

```
'Telemetry': [
{
  'Name': 'Temperature',
  'Type': 'double',
  'DisplayName': 'Temperature (C*)'
},
{
  'Name': 'Humidity',
  'Type': 'double',
  'DisplayName': 'Humidity (relative)'
},
{
  'Name': 'ExternalTemperature',
  'Type': 'double',
  'DisplayName': 'Outdoor Temperature (C*)'
}
]
```

The following screenshot shows how this change modifies the chart legend on the dashboard:




**NOTE**

You may need to disable and then enable the Node.js device on the **Devices** page in the dashboard to see the change immediately.

## Filter the telemetry types

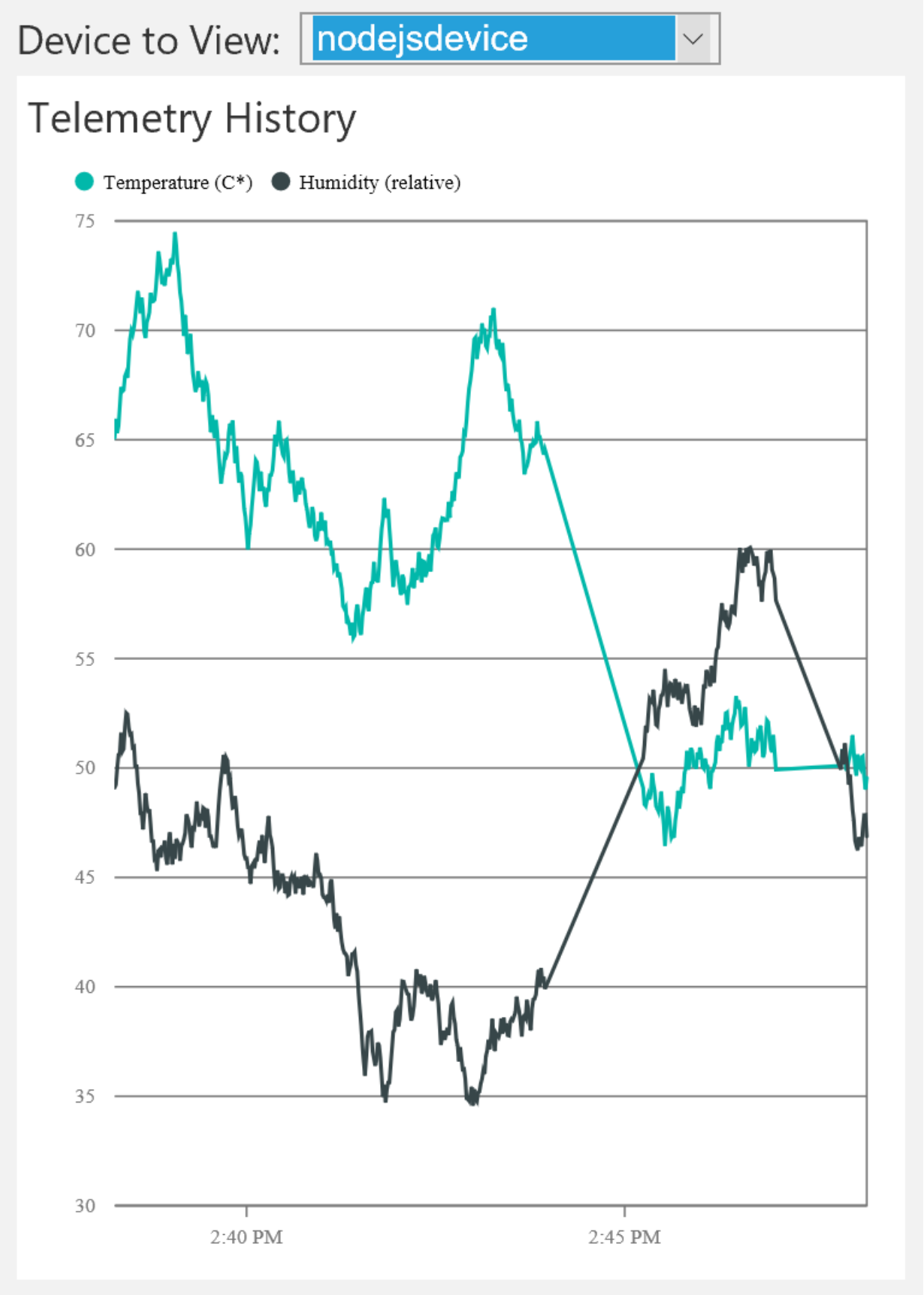
By default, the chart on the dashboard shows every data series in the telemetry stream. You can use the **Device-**

**Info** metadata to suppress the display of specific telemetry types on the chart.

To make the chart show only Temperature and Humidity telemetry, omit **ExternalTemperature** from the **Device-Info Telemetry** metadata as follows:

```
'Telemetry': [
{
  'Name': 'Temperature',
  'Type': 'double',
  'DisplayName': 'Temperature (C*)'
},
{
  'Name': 'Humidity',
  'Type': 'double',
  'DisplayName': 'Humidity (relative)'
},
//{
//  'Name': 'ExternalTemperature',
//  'Type': 'double',
//  'DisplayName': 'Outdoor Temperature (C*)'
//}
]
```

The **Outdoor Temperature** no longer displays on the chart:




This change only affects the chart display. The **ExternalTemperature** data values are still stored and made available for any backend processing.

> **NOTE**
>
> You may need to disable and then enable the Node.js device on the **Devices** page in the dashboard to see the change immediately.

## Handle errors

For a data stream to display on the chart, its **Type** in the **Device-Info** metadata must match the data type of the telemetry values. For example, if the metadata specifies that the **Type** of humidity data is **int** and a **double** is found in the telemetry stream then the humidity telemetry does not display on the chart. However, the **Humidity** values are still stored and made available for any back-end processing.

## Next steps

Now that you've seen how to use dynamic telemetry, you can learn more about how the preconfigured solutions use device information: Device information metadata in the remote monitoring preconfigured solution.

# Create a custom rule in the remote monitoring preconfigured solution

8/24/2017 • 8 min to read • Edit Online

## Introduction

In the preconfigured solutions, you can configure rules that trigger when a telemetry value for a device reaches a specific threshold. Use dynamic telemetry with the remote monitoring preconfigured solution describes how you can add custom telemetry values, such as *ExternalTemperature* to your solution. This article shows you how to create custom rule for dynamic telemetry types in your solution.

This tutorial uses a simple Node.js simulated device to generate dynamic telemetry to send to the preconfigured solution back end. You then add custom rules in the **RemoteMonitoring** Visual Studio solution and deploy this customized back end to your Azure subscription.

To complete this tutorial, you need:

- An active Azure subscription. If you don't have an account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.
- Node.js version 0.12.x or later to create a simulated device.
- Visual Studio 2015 or Visual Studio 2017 to modify the preconfigured solution back end with your new rules.

## Provision the solution

If you haven't already provisioned the remote monitoring preconfigured solution in your account:

1. Log on to azureiotsuite.com using your Azure account credentials, and click **+** to create a solution.
2. Click **Select** on the **Remote monitoring** tile.
3. Enter a **Solution name** for your remote monitoring preconfigured solution.
4. Select the **Region** and **Subscription** you want to use to provision the solution.
5. Click **Create Solution** to begin the provisioning process. This process typically takes several minutes to run.

**Wait for the provisioning process to complete**

1. Click the tile for your solution with **Provisioning** status.
2. Notice the **Provisioning states** as Azure services are deployed in your Azure subscription.
3. Once provisioning completes, the status changes to **Ready**.
4. Click the tile to see the details of your solution in the right-hand pane.

> **NOTE**
> If you are encountering issues deploying the pre-configured solution, review Permissions on the azureiotsuite.com site and the FAQ. If the issues persist, create a service ticket on the portal.

Are there details you'd expect to see that aren't listed for your solution? Give us feature suggestions on User Voice.

Make a note of the solution name you chose for your deployment. You need this solution name later in this tutorial.

## Configure the Node.js simulated device

1. On the remote monitoring dashboard, click **+ Add a device** and then add a *custom device*. Make a note of the IoT Hub hostname, device id, and device key. You need them later in this tutorial when you prepare the remote_monitoring.js device client application.
2. Ensure that Node.js version 0.12.x or later is installed on your development machine. Run `node --version` at a command prompt or in a shell to check the version. For information about using a package manager to install Node.js on Linux, see Installing Node.js via package manager.
3. When you have installed Node.js, clone the latest version of the azure-iot-sdk-node repository to your development machine. Always use the **master** branch for the latest version of the libraries and samples.
4. From your local copy of the azure-iot-sdk-node repository, copy the following two files from the node/device/samples folder to an empty folder on your development machine:

   - packages.json
   - remote_monitoring.js
5. Open the remote_monitoring.js file and look for the following variable definition:

   ```
   var connectionString = "[IoT Hub device connection string]";
   ```
6. Replace **[IoT Hub device connection string]** with your device connection string. Use the values for your IoT Hub hostname, device id, and device key that you made a note of in step 1. A device connection string has the following format:

   ```
   HostName={your IoT Hub hostname};DeviceId={your device id};SharedAccessKey={your device key}
   ```

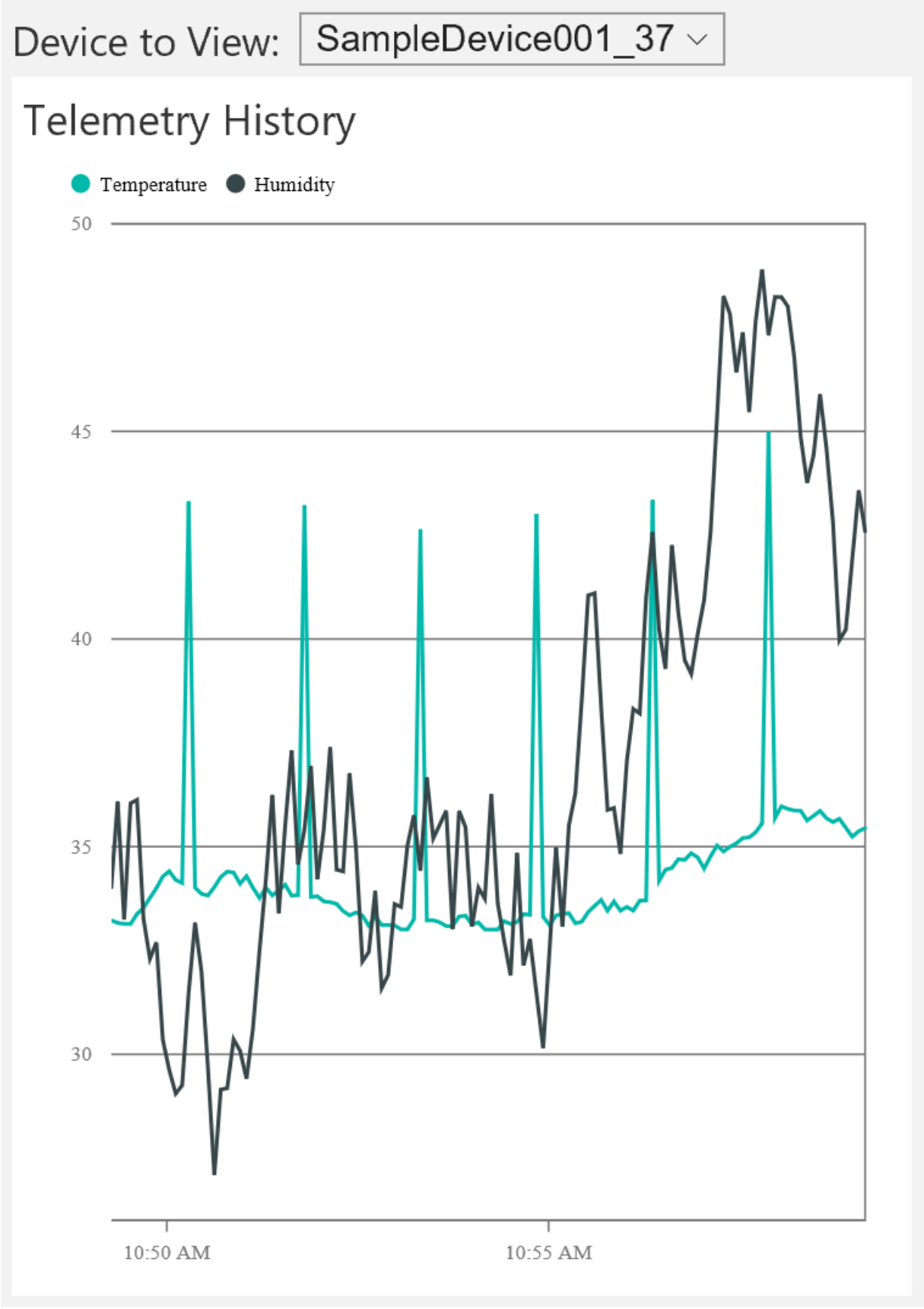
   If your IoT Hub hostname is **contoso** and your device id is **mydevice**, your connection string looks like the following snippet:

   ```
   var connectionString = "HostName=contoso.azure-devices.net;DeviceId=mydevice;SharedAccessKey=2s ... =="
   ```
7. Save the file. Run the following commands in a shell or command prompt in the folder that contains these files to install the necessary packages and then run the sample application:

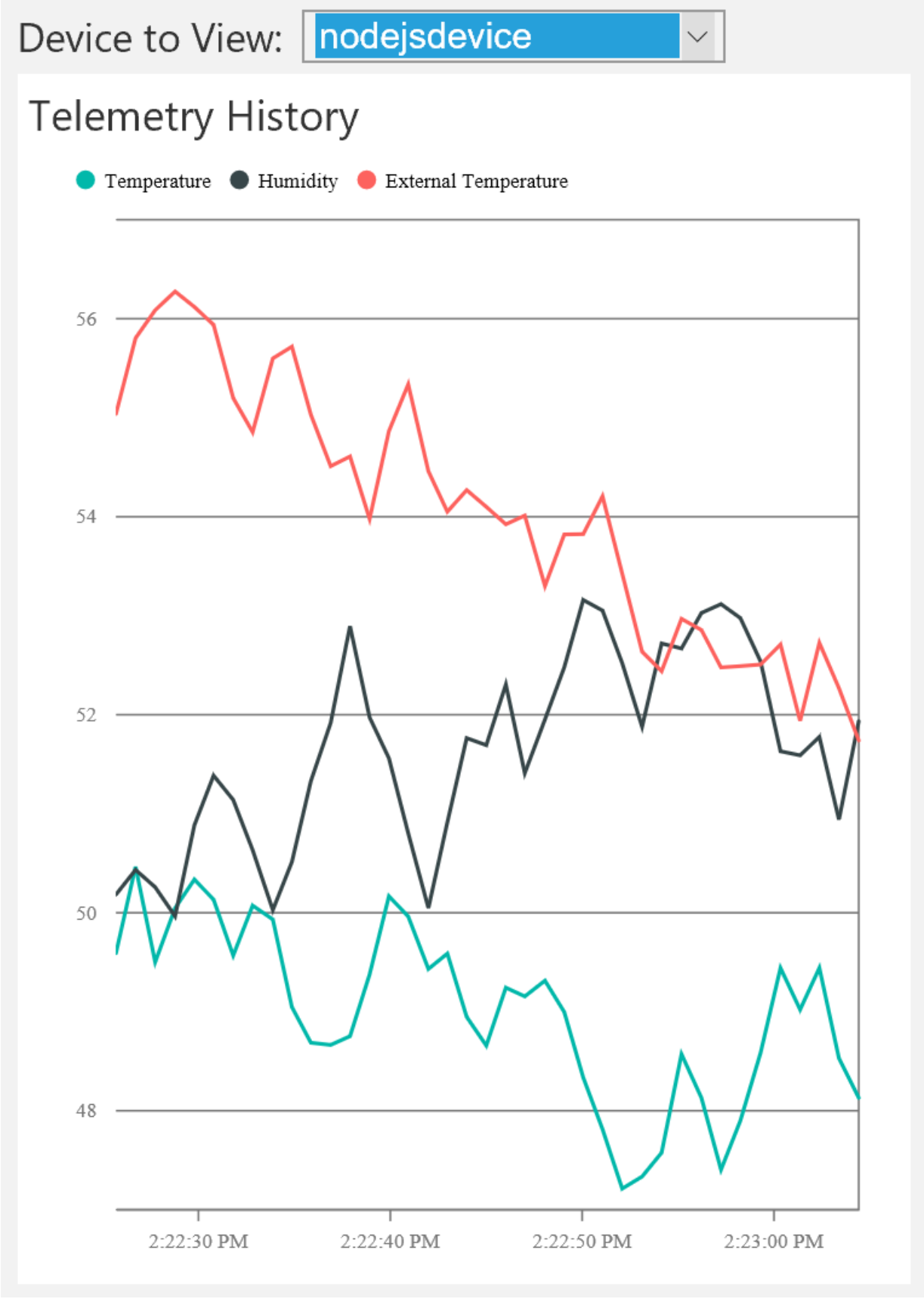
   ```
   npm install
   node remote_monitoring.js
   ```

## Observe dynamic telemetry in action

The dashboard shows the temperature and humidity telemetry from the existing simulated devices:




If you select the Node.js simulated device you ran in the previous section, you see temperature, humidity, and external temperature telemetry:




The remote monitoring solution automatically detects the additional external temperature telemetry type and adds it to the chart on the dashboard.

You can stop the Node.js console app when you have verified that it is sending **ExternalTemperature** telemetry to the preconfigured solution. Keep the console window open because you run this Node.js console app again after you add the custom rule to the solution.

## Rule storage locations

Information about rules is persisted in two locations:

- **DeviceRulesNormalizedTable** table – This table stores a normalized reference to the rules defined by the solution portal. When the solution portal displays device rules, it queries this table for the rule definitions.
- **DeviceRules** blob – This blob stores all the rules defined for all registered devices and is defined as a reference input to the Azure Stream Analytics jobs.

When you update an existing rule or define a new rule in the solution portal, both the table and blob are updated to reflect the changes. The rule definition displayed in the portal comes from the table store, and the rule definition referenced by the Stream Analytics jobs comes from the blob.

## Update the RemoteMonitoring Visual Studio solution

The following steps show you how to modify the RemoteMonitoring Visual Studio solution to include a new rule that uses the **ExternalTemperature** telemetry sent from the simulated device:

1. If you have not already done so, clone the **azure-iot-remote-monitoring** repository to a suitable location on your local machine using the following Git command:

   ```
   git clone https://github.com/Azure/azure-iot-remote-monitoring.git
   ```

2. In Visual Studio, open the RemoteMonitoring.sln file from your local copy of the **azure-iot-remote-monitoring** repository.

3. Open the file Infrastructure\Models\DeviceRuleBlobEntity.cs and add an **ExternalTemperature** property as follows:

   ```
   public double? Temperature { get; set; }
   public double? Humidity { get; set; }
   public double? ExternalTemperature { get; set; }
   ```

4. In the same file, add an **ExternalTemperatureRuleOutput** property as follows:

   ```
   public string TemperatureRuleOutput { get; set; }
   public string HumidityRuleOutput { get; set; }
   public string ExternalTemperatureRuleOutput { get; set; }
   ```

5. Open the file Infrastructure\Models\DeviceRuleDataFields.cs and add the following **ExternalTemperature** property after the existing **Humidity** property:

   ```
   public static string ExternalTemperature
   {
       get { return "ExternalTemperature"; }
   }
   ```

6. In the same file, update the **_availableDataFields** method to include **ExternalTemperature** as follows:

   ```
   private static List<string> _availableDataFields = new List<string>
   {
       Temperature, Humidity, ExternalTemperature
   };
   ```

7. Open the file Infrastructure\Repository\DeviceRulesRepository.cs and modify the **BuildBlobEntityListFromTableRows** method as follows:

```
else if (rule.DataField == DeviceRuleDataFields.Humidity)
{
    entity.Humidity = rule.Threshold;
    entity.HumidityRuleOutput = rule.RuleOutput;
}
else if (rule.DataField == DeviceRuleDataFields.ExternalTemperature)
{
  entity.ExternalTemperature = rule.Threshold;
  entity.ExternalTemperatureRuleOutput = rule.RuleOutput;
}
```

## Rebuild and redeploy the solution.

You can now deploy the updated solution to your Azure subscription.

1. Open an elevated command prompt and navigate to the root of your local copy of the azure-iot-remote-monitoring repository.
2. To deploy your updated solution, run the following command substituting **{deployment name}** with the name of your preconfigured solution deployment that you noted previously:

```
build.cmd cloud release {deployment name}
```

## Update the Stream Analytics job

When the deployment is complete, you can update the Stream Analytics job to use the new rule definitions.

1. In the Azure portal, navigate to the resource group that contains your preconfigured solution resources. This resource group has the same name you specified for the solution during the deployment.
2. Navigate to the {deployment name}-Rules Stream Analytics job.
3. Click **Stop** to stop the Stream Analytics job from running. (You must wait for the streaming job to stop before you can edit the query).
4. Click **Query**. Edit the query to include the **SELECT** statement for **ExternalTemperature**. The following sample shows the complete query with the new **SELECT** statement:

```
WITH AlarmsData AS
(
SELECT
     Stream.IoTHub.ConnectionDeviceId AS DeviceId,
     'Temperature' as ReadingType,
     Stream.Temperature as Reading,
     Ref.Temperature as Threshold,
     Ref.TemperatureRuleOutput as RuleOutput,
     Stream.EventEnqueuedUtcTime AS [Time]
FROM IoTTelemetryStream Stream
JOIN DeviceRulesBlob Ref ON Stream.IoTHub.ConnectionDeviceId = Ref.DeviceID
WHERE
     Ref.Temperature IS NOT null AND Stream.Temperature > Ref.Temperature

UNION ALL

SELECT
     Stream.IoTHub.ConnectionDeviceId AS DeviceId,
     'Humidity' as ReadingType,
     Stream.Humidity as Reading,
     Ref.Humidity as Threshold,
     Ref.HumidityRuleOutput as RuleOutput,
     Stream.EventEnqueuedUtcTime AS [Time]
FROM IoTTelemetryStream Stream
JOIN DeviceRulesBlob Ref ON Stream.IoTHub.ConnectionDeviceId = Ref.DeviceID
WHERE
     Ref.Humidity IS NOT null AND Stream.Humidity > Ref.Humidity

UNION ALL

SELECT
     Stream.IoTHub.ConnectionDeviceId AS DeviceId,
     'ExternalTemperature' as ReadingType,
     Stream.ExternalTemperature as Reading,
     Ref.ExternalTemperature as Threshold,
     Ref.ExternalTemperatureRuleOutput as RuleOutput,
     Stream.EventEnqueuedUtcTime AS [Time]
FROM IoTTelemetryStream Stream
JOIN DeviceRulesBlob Ref ON Stream.IoTHub.ConnectionDeviceId = Ref.DeviceID
WHERE
     Ref.ExternalTemperature IS NOT null AND Stream.ExternalTemperature > Ref.ExternalTemperature
)

SELECT *
INTO DeviceRulesMonitoring
FROM AlarmsData

SELECT *
INTO DeviceRulesHub
FROM AlarmsData
```

5. Click **Save** to change the updated rule query.
6. Click **Start** to start the Stream Analytics job running again.

## Add your new rule in the dashboard

You can now add the **ExternalTemperature** rule to a device in the solution dashboard.

1. Navigate to the solution portal.
2. Navigate to the **Devices** panel.
3. Locate the custom device you created that sends **ExternalTemperature** telemetry and on the **Device Details** panel, click **Add Rule**.

4. Select **ExternalTemperature** in **Data Field**.
5. Set **Threshold** to 56. Then click **Save and view rules**.
6. Return to the dashboard to view the alarm history.
7. In the console window you left open, start the Node.js console app to begin sending **ExternalTemperature** telemetry data.
8. Notice that the **Alarm History** table shows new alarms when the new rule is triggered.

## Additional information

Changing the operator **>** is more complex and goes beyond the steps outlined in this tutorial. While you can change the Stream Analytics job to use whatever operator you like, reflecting that operator in the solution portal is a more complex task.

## Next steps

Now that you've seen how to create custom rules, you can learn more about the preconfigured solutions:

- Connect Logic App to your Azure IoT Suite Remote Monitoring preconfigured solution
- Device information metadata in the remote monitoring preconfigured solution.

# Device information metadata in the remote monitoring preconfigured solution

8/24/2017 • 5 min to read • Edit Online

The Azure IoT Suite remote monitoring preconfigured solution demonstrates an approach for managing device metadata. This article outlines the approach this solution takes to enable you to understand:

- What device metadata the solution stores.
- How the solution manages the device metadata.

## Context

The remote monitoring preconfigured solution uses Azure IoT Hub to enable your devices to send data to the cloud. The solution stores information about devices in three different locations:

| LOCATION | INFORMATION STORED | IMPLEMENTATION |
| --- | --- | --- |
| Identity registry | Device id, authentication keys, enabled state | Built in to IoT Hub |
| Device twins | Metadata: reported properties, desired properties, tags | Built in to IoT Hub |
| Cosmos DB | Command and method history | Custom for solution |

IoT Hub includes a device identity registry to manage access to an IoT hub and uses device twins to manage device metadata. There is also a remote monitoring solution-specific *device registry* that stores command and method history. The remote monitoring solution uses a Cosmos DB database to implement a custom store for command and method history.

> **NOTE**
> The remote monitoring preconfigured solution keeps the device identity registry in sync with the information in the Cosmos DB database. Both use the same device id to uniquely identify each device connected to your IoT hub.

## Device metadata

IoT Hub maintains a device twin for each simulated and physical device connected to a remote monitoring solution. The solution uses device twins to manage the metadata associated with devices. A device twin is a JSON document maintained by IoT Hub, and the solution uses the IoT Hub API to interact with device twins.

A device twin stores three types of metadata:

- *Reported properties* are sent to an IoT hub by a device. In the remote monitoring solution, simulated devices send reported properties at start-up and in response to **Change device state** commands and methods. You can view reported properties in the **Device list** and **Device details** in the solution portal. Reported properties are read only.
- *Desired properties* are retrieved from the IoT hub by devices. It is the responsibility of the device to make any necessary configuration change on the device. It is also the responsibility of the device to report the change

back to the hub as a reported property. You can set a desired property value through the solution portal.

- *Tags* only exist in the device twin and are never synchronized with a device. You can set tag values in the solution portal and use them when you filter the list of devices. The solution also uses a tag to identify the icon to display for a device in the solution portal.

Example reported properties from the simulated devices include manufacturer, model number, latitude, and longitude. Simulated devices also return the list of supported methods as a reported property.

> **NOTE**
>
> The simulated device code only uses the **Desired.Config.TemperatureMeanValue** and **Desired.Config.TelemetryInterval** desired properties to update the reported properties sent back to IoT Hub. All other desired property change requests are ignored.

A device information metadata JSON document stored in the device registry Cosmos DB database has the following structure:

```
{
  "DeviceProperties": {
    "DeviceID": "deviceid1",
    "HubEnabledState": null,
    "CreatedTime": "2016-04-25T23:54:01.313802Z",
    "DeviceState": "normal",
    "UpdatedTime": null
    },
  "SystemProperties": {
    "ICCID": null
  },
  "Commands": [],
  "CommandHistory": [],
  "IsSimulatedDevice": false,
  "id": "fe81a81c-bcbc-4970-81f4-7f12f2d8bda8"
}
```

> **NOTE**
>
> Device information can also include metadata to describe the telemetry the device sends to IoT Hub. The remote monitoring solution uses this telemetry metadata to customize how the dashboard displays dynamic telemetry.

# Lifecycle

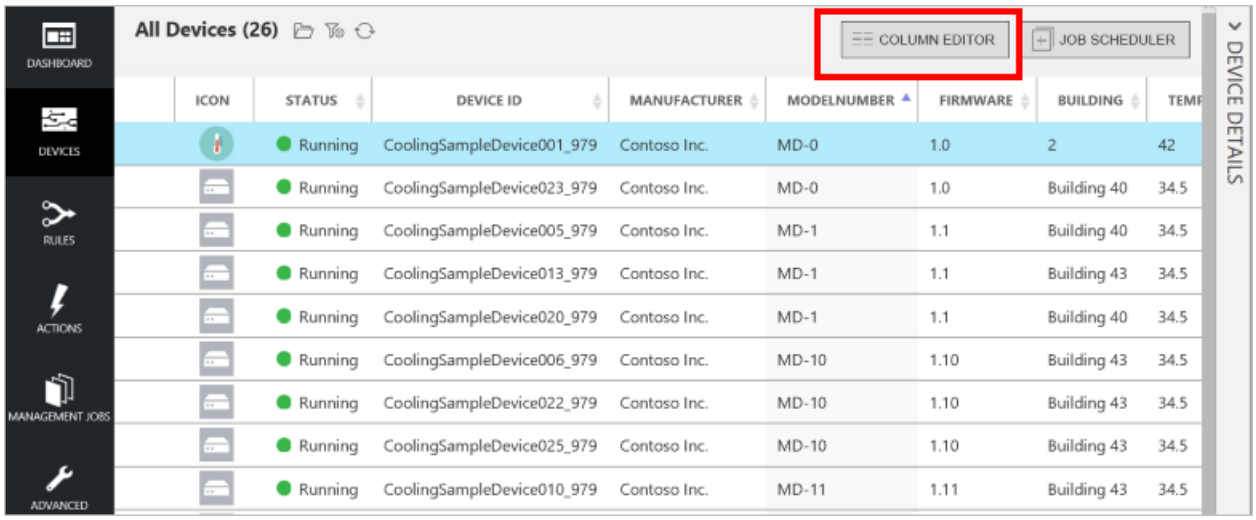
When you first create a device in the solution portal, the solution creates an entry in the Cosmos DB database to store command and method history. At this point, the solution also creates an entry for the device in the device identity registry, which generates the keys the device uses to authenticate with IoT Hub. It also creates a device twin.

When a device first connects to the solution, it sends reported properties and a device information message. The reported property values are automatically saved in the device twin. The reported properties include the device manufacturer, model number, serial number, and a list of supported methods. The device information message includes the list of the commands the device supports including information about any command parameters. When the solution receives this message, it updates the device information in the Cosmos DB database.

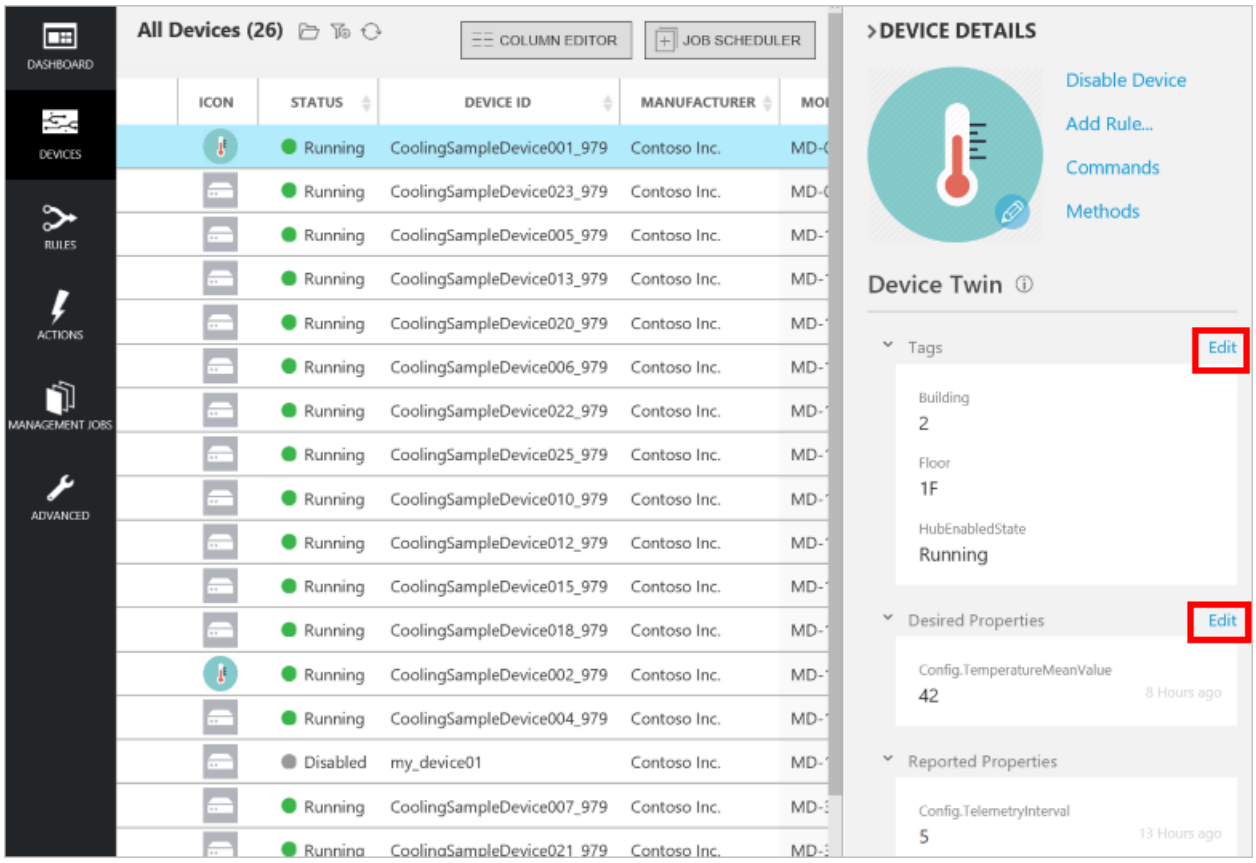
### View and edit device information in the solution portal

The device list in the solution portal displays the following device properties as columns by default: **Status**, **DeviceId**, **Manufacturer**, **Model Number**, **Serial Number**, **Firmware**, **Platform**, **Processor**, and **Installed**

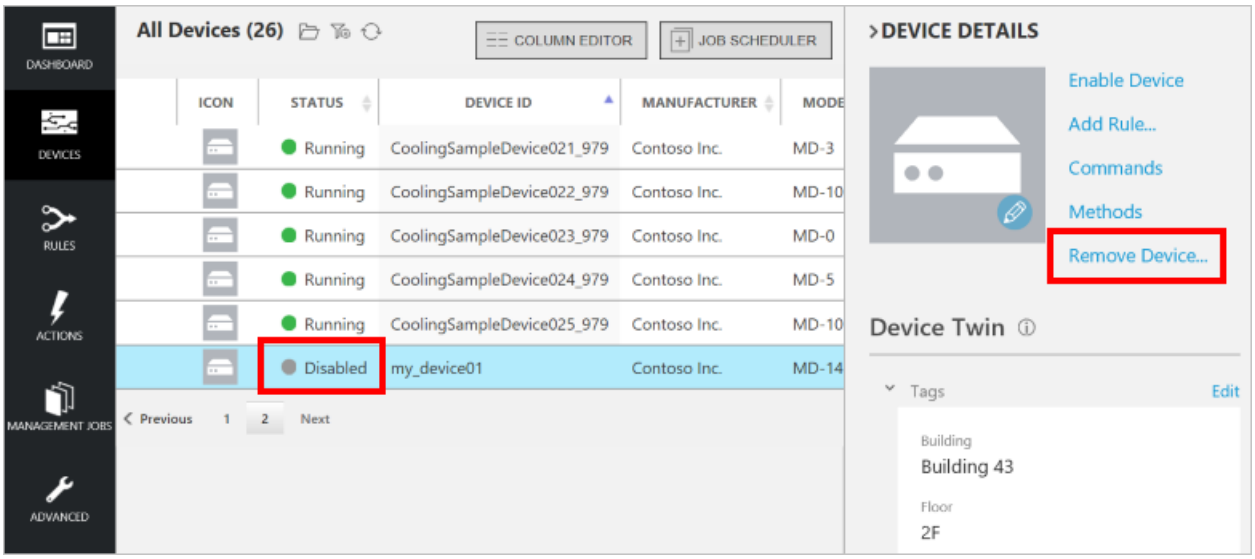
**RAM**. You can customize the columns by clicking **Column editor**. The device properties **Latitude** and **Longitude** drive the location in the Bing Map on the dashboard.




In the **Device Details** pane in the solution portal, you can edit desired properties and tags (reported properties are read only).




You can use the solution portal to remove a device from your solution. When you remove a device, the solution removes the device entry from identity registry and then deletes the device twin. The solution also removes information related to the device from the Cosmos DB database. Before you can remove a device, you must disable it.




## Device information message processing

Device information messages sent by a device are distinct from telemetry messages. Device information messages include the commands a device can respond to, and any command history. IoT Hub itself has no knowledge of the metadata contained in a device information message and processes the message in the same way it processes any device-to-cloud message. In the remote monitoring solution, an Azure Stream Analytics (ASA) job reads the messages from IoT Hub. The **DeviceInfo** stream analytics job filters for messages that contain **"ObjectType": "DeviceInfo"** and forwards them to the **EventProcessorHost** host instance that runs in a web job. Logic in the **EventProcessorHost** instance uses the device id to find the Cosmos DB record for the specific device and update the record.

> **NOTE**
> A device information message is a standard device-to-cloud message. The solution distinguishes between device information messages and telemetry messages by using ASA queries.

## Next steps

Now you've finished learning how you can customize the preconfigured solutions, you can explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Frequently asked questions for IoT Suite
- IoT security from the ground up

# Deploy a gateway on Windows or Linux for the connected factory preconfigured solution

7/24/2017 • 8 min to read • Edit Online

The software required to deploy a gateway for the connected factory preconfigured solution has two components:

- The *OPC Proxy* establishes a connection to IoT Hub. The *OPC Proxy* then waits for command and control messages from the integrated OPC Browser that runs in the connected factory solution portal.
- The *OPC Publisher* connects to existing on-premises OPC UA servers and forwards telemetry messages from them to IoT Hub.

Both components are open-source and are available as source on GitHub and as Docker containers:

| GITHUB | DOCKERHUB |
| --- | --- |
| OPC Publisher | OPC Publisher |
| OPC Proxy | OPC Proxy |

No public-facing IP address or holes in the gateway firewall are required for either component. The OPC Proxy and OPC Publisher use only outbound ports 443, 5671, and 8883.

The steps in this article show you how to deploy a gateway using Docker on either Windows or Linux. The gateway enables connectivity to the connected factory preconfigured solution.

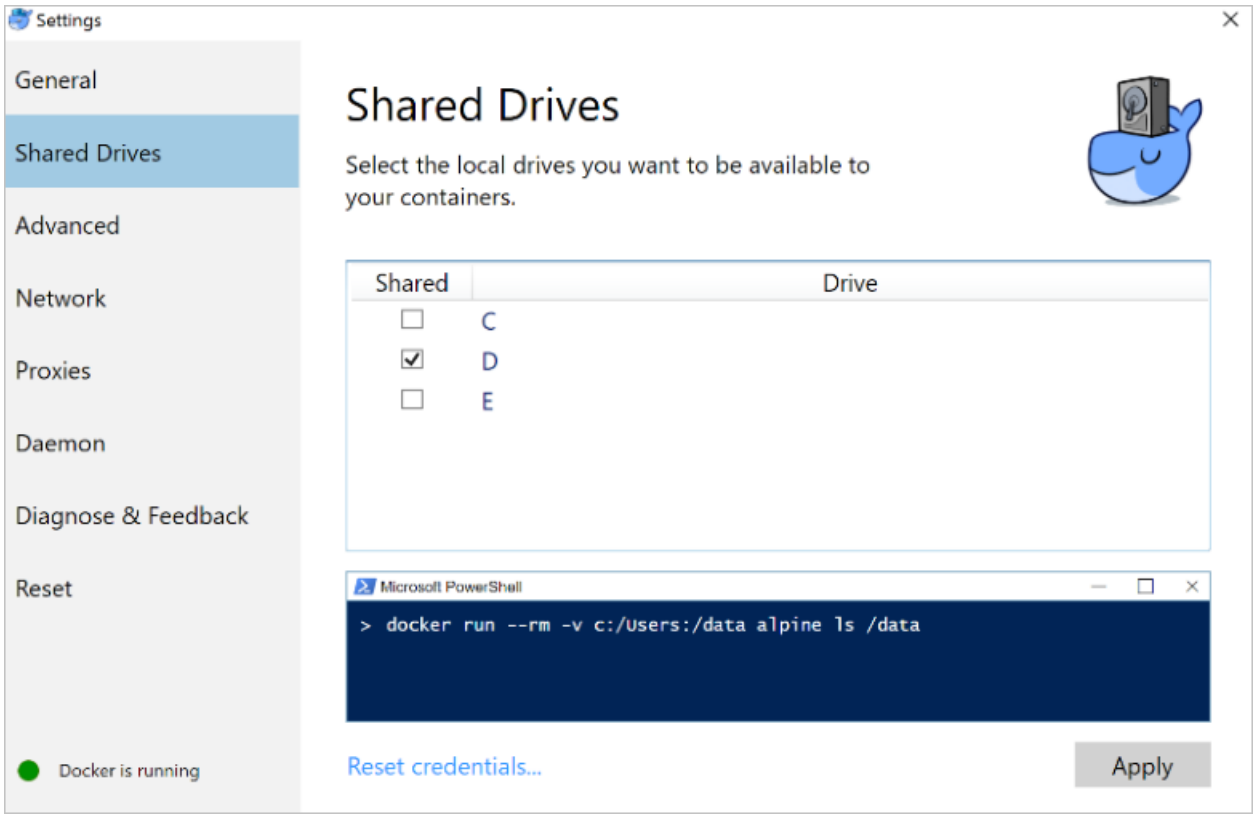
> **NOTE**
> The gateway software that runs in the Docker container is Azure IoT Edge.

## Windows deployment

> **NOTE**
> If you don't yet have a gateway device, Microsoft recommends you buy a commercial gateway from one of our partners. Visit the Azure IoT device catalog for a list of gateway devices compatible with the connected factory solution. Follow the instructions that come with the device to set up the gateway. Alternatively, use the following instructions to manually set up one of your existing gateways.

### Install Docker

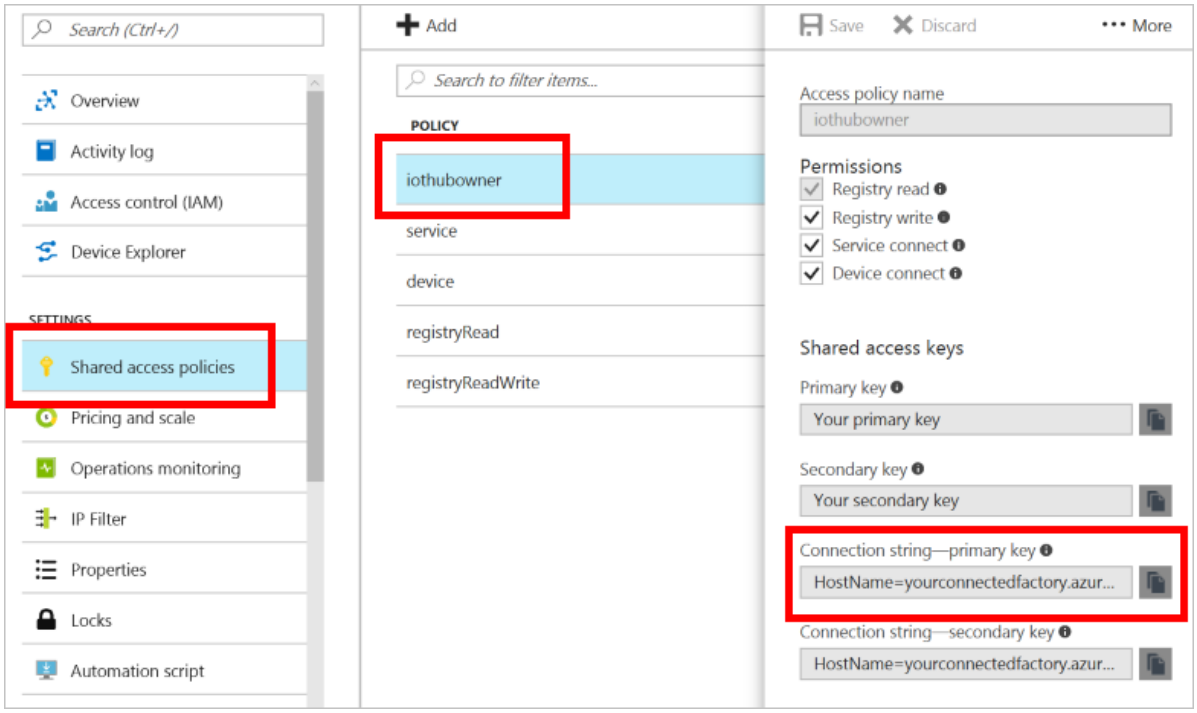
Install Docker for Windows on your Windows-based gateway device. During Windows Docker setup, select a drive on your host machine to share with Docker. The following screenshot shows sharing the D drive on your Windows system:




Then create a folder called **docker** in the root of the shared drive. You can also perform this step after installing docker from the **Settings** menu.

### Configure the gateway

1. You need the **iothubowner** connection string of your Azure IoT Suite connected factory deployment to complete the gateway deployment. In the Azure portal, navigate to your IoT Hub in the resource group created when you deployed the connected factory solution. Click **Shared access policies** to access the **iothubowner** connection string:




   Copy the **Connection string--primary key** value.

2. Configure the gateway for your IoT Hub by running the two gateway modules **once** from a command prompt with:

```
docker run -it --rm -h <ApplicationName> -v
//D/docker:/build/src/GatewayApp.NetCore/bin/Debug/netcoreapp1.0/publish/CertificateStores -v
//D/docker:/root/.dotnet/corefx/cryptography/x509stores microsoft/iot-gateway-opc-ua:1.0.0
<ApplicationName> "<IoTHubOwnerConnectionString>"
```

```
docker run -it --rm -v //D/docker:/mapped microsoft/iot-gateway-opc-ua-proxy:0.1.3 -i -c "
<IoTHubOwnerConnectionString>" -D /mapped/cs.db
```

- **<ApplicationName>** is the name to give to your OPC UA Publisher in the format **publisher.<your fully qualified domain name>**. For example, if your factory network is called **myfactorynetwork.com**, the **ApplicationName** value is **publisher.myfactorynetwork.com**.
- **<IoTHubOwnerConnectionString>** is the **iothubowner** connection string you copied in the previous step. This connection string is only used in this step, you don't need it in the following steps:

  You use the mapped D:\docker folder (the `-v` argument) later to persist the two X.509 certificates that the gateway modules use.

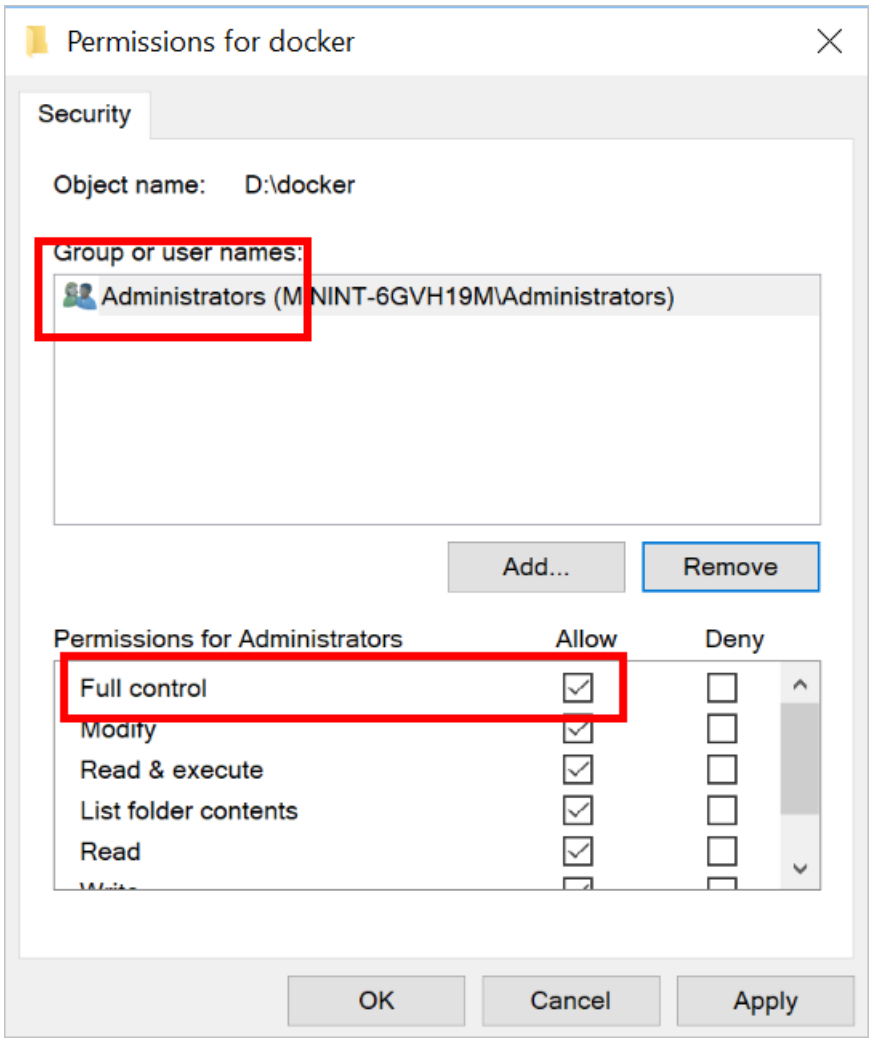
**Run the gateway**

1. Restart the gateway using the following commands:

```
docker run -it --rm -h <ApplicationName> --expose 62222 -p 62222:62222 -v
//D/docker:/build/src/GatewayApp.NetCore/bin/Debug/netcoreapp1.0/publish/Logs -v
//D/docker:/build/src/GatewayApp.NetCore/bin/Debug/netcoreapp1.0/publish/CertificateStores -v
//D/docker:/shared -v //D/docker:/root/.dotnet/corefx/cryptography/x509stores -e
_GW_PNFP="/shared/publishednodes.JSON" microsoft/iot-gateway-opc-ua:1.0.0 <ApplicationName>
```

```
docker run -it --rm -v //D/docker:/mapped microsoft/iot-gateway-opc-ua-proxy:0.1.3 -D /mapped/cs.db
```

2. For security reasons, the two X.509 certificates persisted in the D:\docker folder contain the private key. Limit access to this folder to the credentials (typically **Administrators**) you use to run the Docker container. Right-click the D:\docker folder, choose **Properties**, then **Security**, and then **Edit**. Give **Administrators** full control and remove everyone else:




3. Verify network connectivity. From a command prompt, enter the command `ping publisher.<your fully qualified domain name>` to ping your gateway. If the destination is unreachable, add the IP address and name of your gateway to the hosts file on your gateway. The hosts file is located in the **Windows\System32\drivers\etc** folder.

4. Next, try to connect to the publisher using a local OPC UA client running on the gateway. The OPC UA endpoint URL is `opc.tcp://publisher.<your fully qualified domain name>:62222`. If you don't have an OPC UA client, you can download and use an open-source OPC UA client.

5. When you have successfully completed these local tests, browse to the **Connect your own OPC UA Server** page in the connected factory solution portal. Enter the publisher endpoint URL ( `tcp://publisher.<your fully qualified domain name>:62222` ) and click **Connect**. You get a certificate warning, then click **Proceed.** Next you get an error that the publisher doesn't trust the UA Web Client. To resolve this error, copy the **UA Web Client** certificate from the **D:\docker\Rejected Certificates\certs** folder to the **D:\docker\UA Applications\certs** folder on the gateway. You do not need to restart of the gateway. Repeat this step. You can now connect to the gateway from the cloud, and you are ready to add OPC UA servers to the solution.

**Add your OPC UA servers**

1. Browse to the **Connect your own OPC UA Server** page in the connected factory solution portal. Follow the same steps as in the preceding section to establish trust between the connected factory portal and the OPC UA server. This step establishes a mutual trust of the certificates from the connected factory portal and the OPC UA server and creates a connection.

2. Browse the OPC UA nodes tree of your OPC UA server, right-click the OPC nodes, and select **publish**. For publishing to work this way, the OPC UA server and the publisher must be on the same network. In other words, if the fully qualified domain name of the publisher is **publisher.mydomain.com** then the fully

qualified domain name of the OPC UA server must be, for example, **myopcuaserver.mydomain.com**. If your setup is different, you can manually add nodes to the publishesnodes.json file found in the **D:\docker** folder. The publishesnodes.json file is automatically generated on the first successful publish of an OPC node.

3. Telemetry now flows from the gateway device. You can view the telemetry in the **Factory Locations** view of the connected factory portal under **New Factory**.

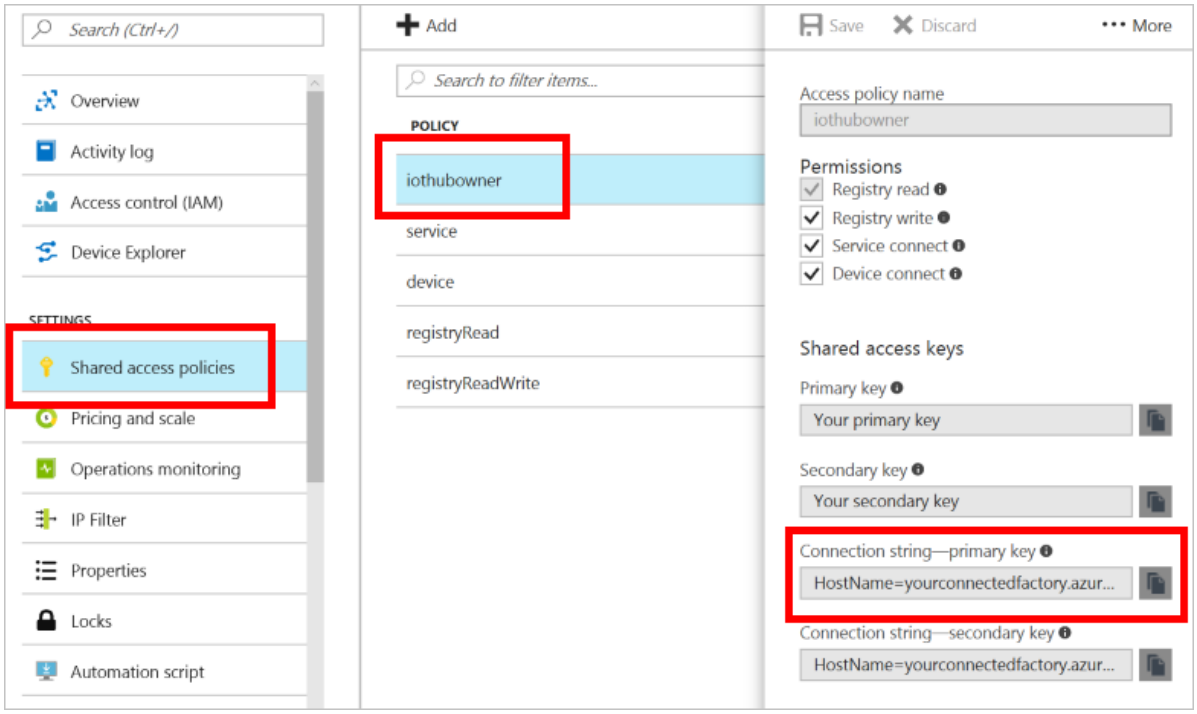
# Linux deployment

> **NOTE**
> If you don't yet have a gateway device, Microsoft recommends you buy a commercial gateway from one of our partners. Visit the Azure IoT device catalog for a list of gateway devices compatible with the connected factory solution. Follow the instructions that come with the device to set up the gateway. Alternatively, use the following instructions to manually set up one of your existing gateways.

Install Docker on your Linux gateway device.

### Configure the gateway

1. You need the **iothubowner** connection string of your Azure IoT Suite connected factory deployment to complete the gateway deployment. In the Azure portal, navigate to your IoT Hub in the resource group created when you deployed the connected factory solution. Click **Shared access policies** to access the **iothubowner** connection string:




   Copy the **Connection string--primary key** value.

2. Configure the gateway for your IoT Hub by running the two gateway modules **once** from a shell with:

```
sudo docker run -it --rm -h <ApplicationName> -v
/shared:/build/src/GatewayApp.NetCore/bin/Debug/netcoreapp1.0/publish/ -v
/shared:/root/.dotnet/corefx/cryptography/x509stores microsoft/iot-gateway-opc-ua:1.0.0
<ApplicationName> "<IoTHubOwnerConnectionString>"
```

```
sudo docker run --rm -it -v /shared:/mapped microsoft/iot-gateway-opc-ua-proxy:0.1.3 -i -c "
<IoTHubOwnerConnectionString>" -D /mapped/cs.db
```

   - **<ApplicationName>** is the name of the OPC UA application the gateway creates in the format

**publisher.<your fully qualified domain name>**. For example, **publisher.microsoft.com**.

- **<IoTHubOwnerConnectionString>** is the **iothubowner** connection string you copied in the previous step. This connection string is only used in this step, you don't need it in the following steps:

  You use the **/shared** folder (the `-v` argument) later to persist the two X.509 certificates that the gateway modules use.

**Run the gateway**

1. Restart the gateway using the following commands:

```
sudo docker run -it -h <ApplicationName> --expose 62222 -p 62222:62222 --rm -v
/shared:/build/src/GatewayApp.NetCore/bin/Debug/netcoreapp1.0/publish/Logs -v
/shared:/build/src/GatewayApp.NetCore/bin/Debug/netcoreapp1.0/publish/CertificateStores -v
/shared:/shared -v /shared:/root/.dotnet/corefx/cryptography/x509stores -e
_GW_PNFP="/shared/publishednodes.JSON" microsoft/iot-gateway-opc-ua:1.0.0 <ApplicationName>
```

```
sudo docker run -it -v /shared:/mapped microsoft/iot-gateway-opc-ua-proxy:0.1.3 -D /mapped/cs.db
```

2. For security reasons, the two X.509 certificates persisted in the **/shared** folder contain the private key. Limit access to this folder to the credentials you use to run the Docker container. To set the permissions for **root** only, use the `chmod` shell command on the folder.

3. Verify network connectivity. From a shell, enter the command `ping publisher.<your fully qualified domain name>` to ping your gateway. If the destination is unreachable, add the IP address and name of your gateway to your hosts file on your gateway. The hosts file is located in the **/etc** folder.

4. Next, try to connect to the publisher using a local OPC UA client running on the gateway. The OPC UA endpoint URL is `opc.tcp://publisher.<your fully qualified domain name>:62222`. If you don't have an OPC UA client, you can download and use an open-source OPC UA client.

5. When you have successfully completed these local tests, browse to the **Connect your own OPC UA Server** page in the connected factory solution portal. Enter the publisher endpoint URL ( `tcp://publisher.<your fully qualified domain name>:62222`) and click **Connect**. You get a certificate warning, then click **Proceed.** Next you get an error that the publisher doesn't trust the UA Web Client. To resolve this error, copy the **UA Web Client** certificate from the **/shared/Rejected Certificates/certs** folder to the **/shared/UA Applications/certs** folder on the gateway. You do not need to restart of the gateway. Repeat this step. You can now connect to the gateway from the cloud, and you are ready to add OPC UA servers to the solution.

**Add your OPC UA servers**

1. Browse to the **Connect your own OPC UA Server** page in the connected factory solution portal. Follow the same steps as in the preceding section to establish trust between the connected factory portal and the OPC UA server. This step establishes a mutual trust of the certificates from the connected factory portal and the OPC UA server and creates a connection.

2. Browse the OPC UA nodes tree of your OPC UA server, right-click the OPC nodes, and select **publish**. For publishing to work this way, the OPC UA server and the publisher must be on the same network. In other words, if the fully qualified domain name of the publisher is **publisher.mydomain.com** then the fully qualified domain name of the OPC UA server must be, for example, **myopcuaserver.mydomain.com**. If your setup is different, you can manually add nodes to the publishesnodes.json file found in the **/shared** folder. The publishesnodes.json is automatically generated on the first successful publish of an OPC node.

3. Telemetry now flows from the gateway device. You can view the telemetry in the **Factory Locations** view of the connected factory portal under **New Factory**.

# Next steps

To learn more about the architecture of the connected factory preconfigured solution, see Connected factory preconfigured solution walkthrough.

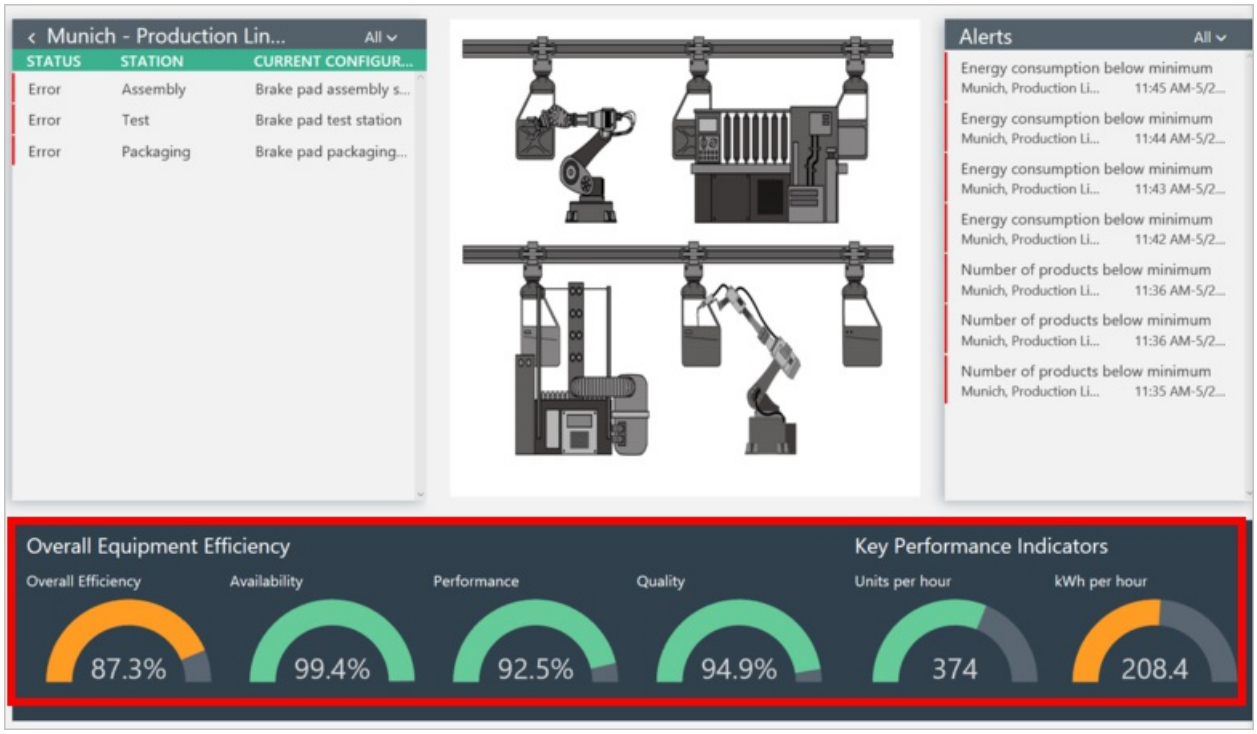
# Customize how the connected factory solution displays data from your OPC UA servers

8/24/2017 • 5 min to read • Edit Online

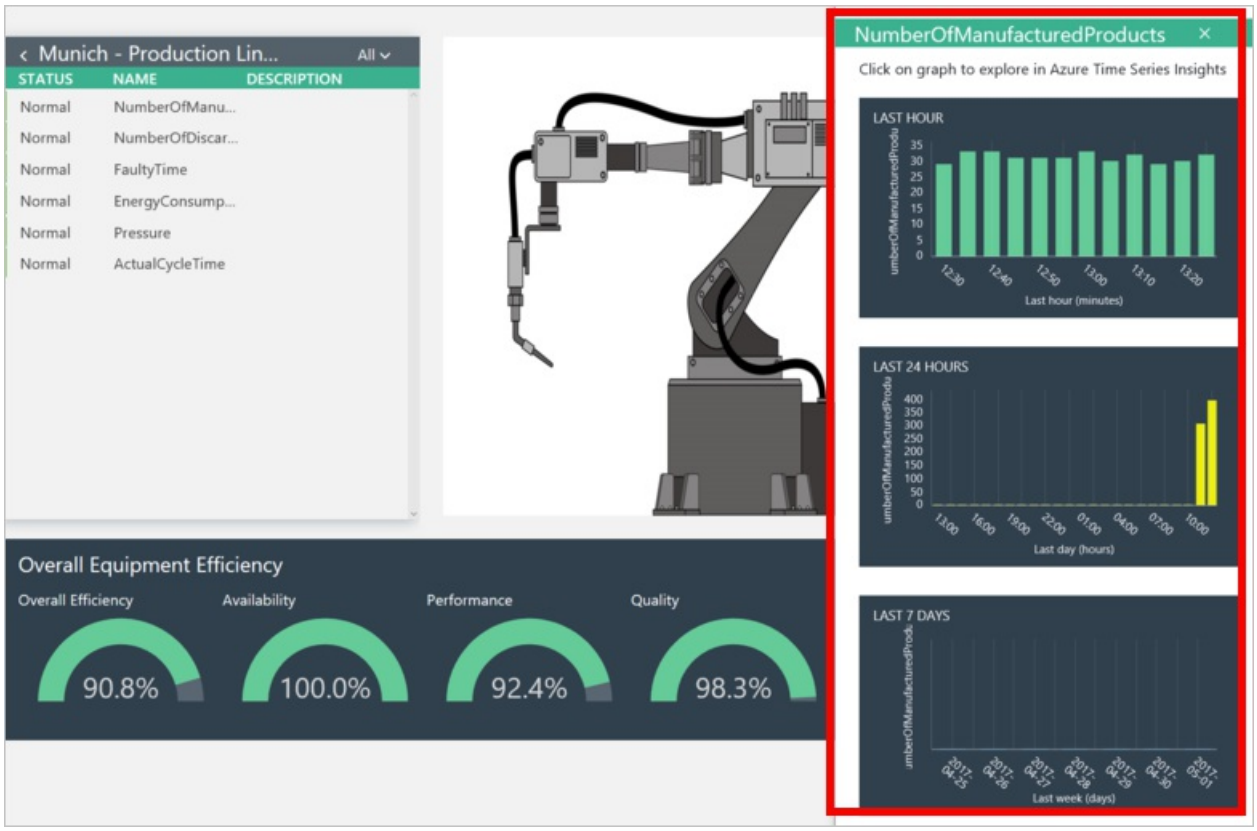
## Introduction

The connected factory solution aggregates and displays data from the OPC UA servers connected to the solution. You can browse and send commands to the OPC UA servers in your solution. For more information about OPC UA, see the Connected factory FAQ.

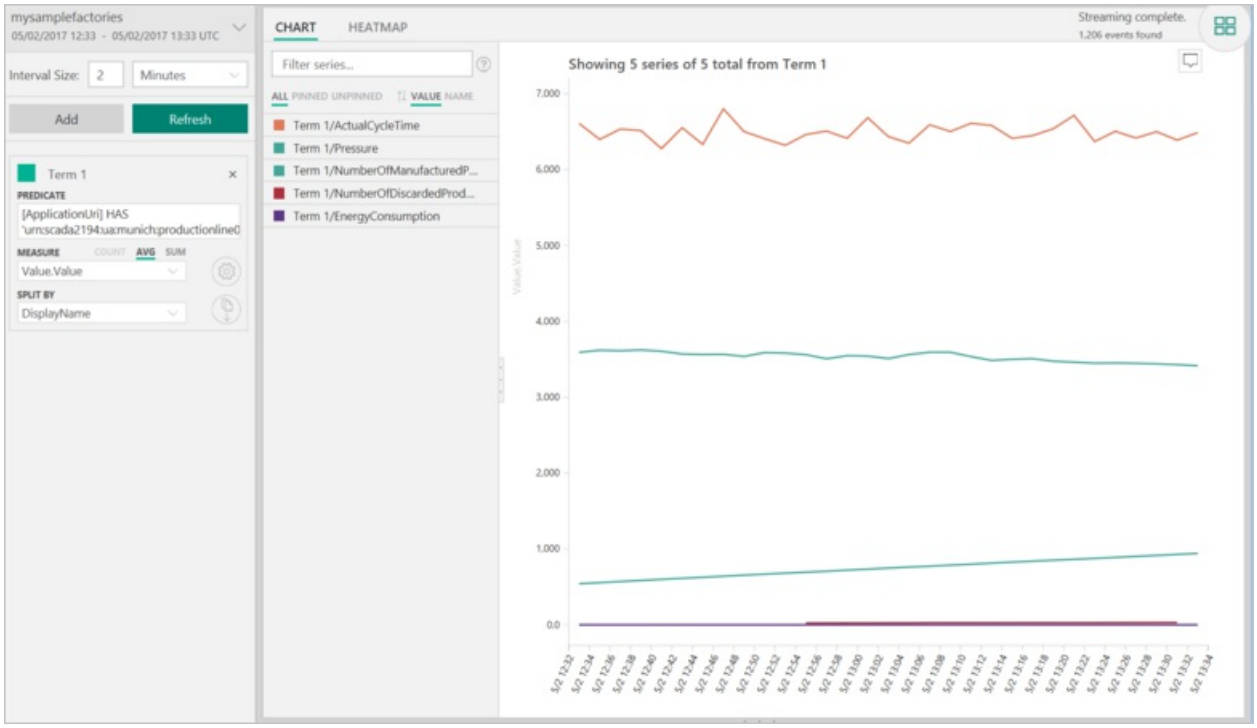
Examples of aggregated data in the solution include the Overall Equipment Efficiency (OEE) and Key Performance Indicators (KPIs) that you can view in the dashboard at the factory, line, and station levels. The following screenshot shows the OEE and KPI values for the **Assembly** station, on **Production line 1**, in the **Munich** factory:




The solution enables you to view detailed information from specific data items from the OPC UA servers, called *stations*. The following screenshot shows plots of the number of manufactured items from a specific station:




If you click one of the graphs, you can explore the data further using Time Series Insights (TSI):




This article describes:

- How the data is made available to the various views in the solution.
- How you can customize the way the solution displays the data.

## Data sources

The connected factory solution displays data from the OPC UA servers connected to the solution. The default installation includes several OPC UA servers running a factory simulation. You can add your own OPC UA servers that connect through a gateway to your solution.

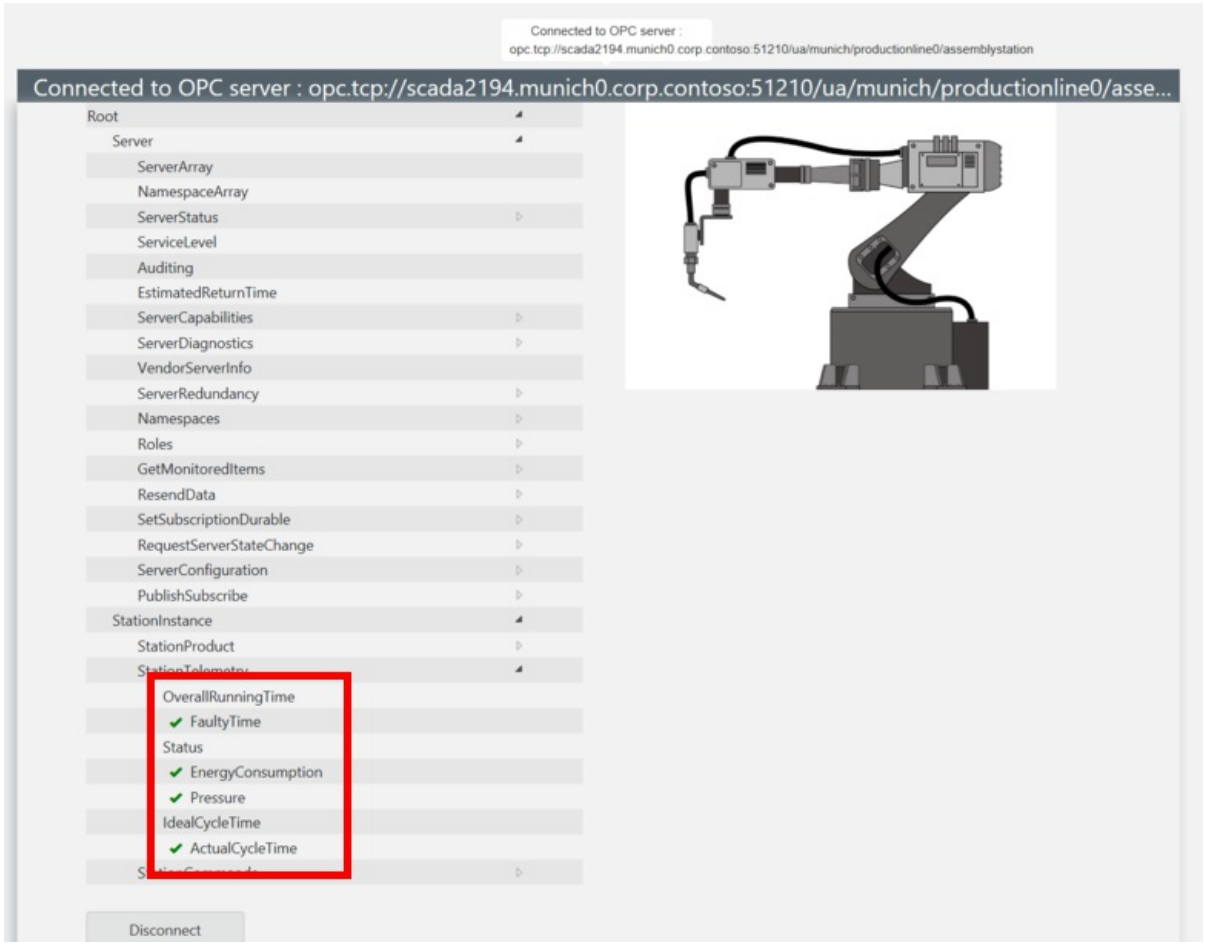
You can browse the data items that a connected OPC UA server can send to your solution in the dashboard:

1. Navigate to the **Select an OPC UA server** view:


2. Select a server and click **Connect**. Click **Proceed** when the security warning appears.

> **NOTE**
> This warning only appears once for each server and establishes a trust relationship between the solution dashboard and the server.

3. You can now browse the data items that the server can send to the solution. Items that are being sent to the solution have a green check mark:




4. If you are an *Administrator* in the solution, you can choose to publish a data item to make it available in the connected factory solution. As an Administrator, you can also change the value of data items and call methods in the OPC UA server.

# Map the data

The connected factory solution maps and aggregates the published data items from the OPC UA server to the various views in the solution. The connected factory solution deploys to your Azure account when you provision the solution. A JSON file in the Visual Studio connected factory solution stores this mapping information. You can view and modify this JSON configuration file in the connected factory Visual Studio solution. You can redeploy the solution after you make a change.

You can use the configuration file to:

- Edit the existing simulated factories, production lines, and stations.
- Map data from real OPC UA servers that you connect to the solution.

To clone a copy of the connected factory Visual Studio solution, use the following git command:

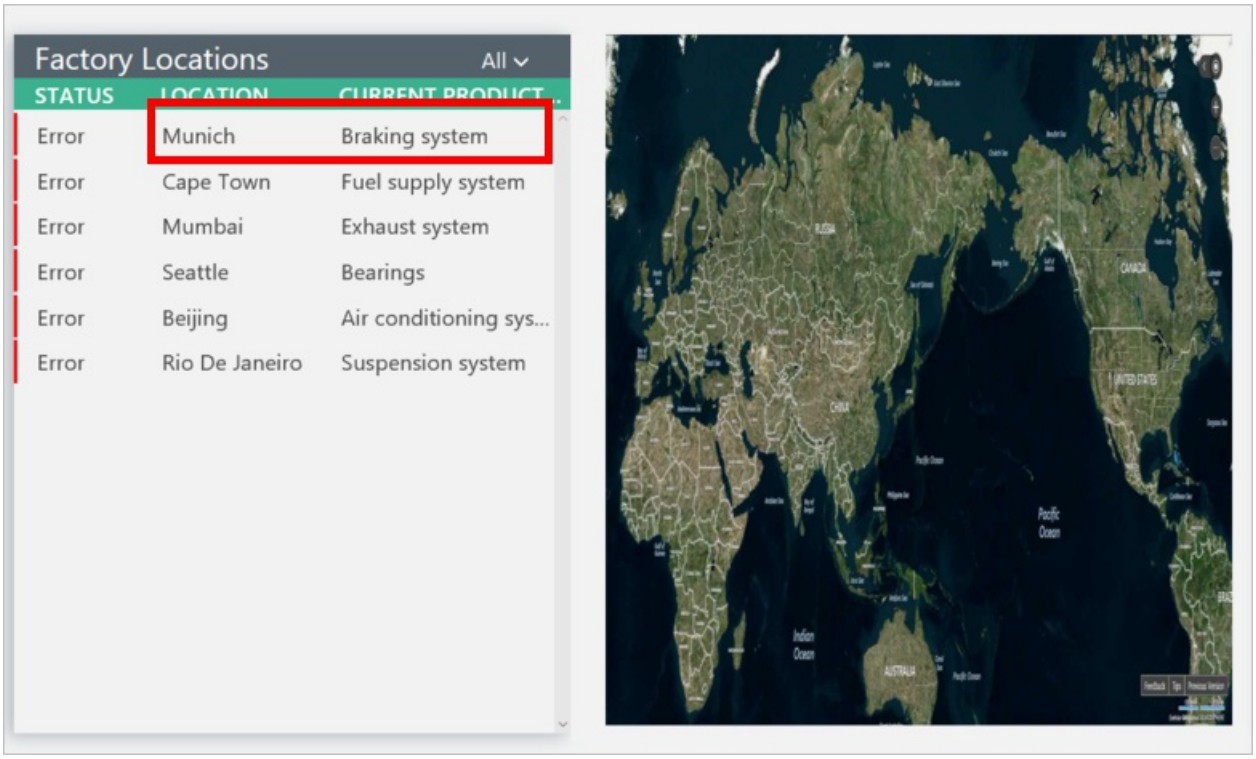
```
git clone https://github.com/Azure/azure-iot-connected-factory.git
```

The file **ContosoTopologyDescription.json** defines the mapping from the OPC UA server data items to the views in the connected factory solution dashboard. You can find this configuration file in the **Contoso\Topology** folder in the **WebApp** project in the Visual Studio solution.

The content of the JSON file is organized as a hierarchy of factory, production line, and station nodes. This hierarchy defines the navigation hierarchy in the connected factory dashboard. Values at each node of the hierarchy determine the information displayed in the dashboard. For example, the JSON file contains the following values for the Munich factory:

```
"Guid": "73B534AE-7C7E-4877-B826-F1C0EA339F65",
"Name": "Munich",
"Description": "Braking system",
"Location": {
    "City": "Munich",
    "Country": "Germany",
    "Latitude": 48.13641,
    "Longitude": 11.57754
},
"Image": "munich.jpg"
```

The name, description, and location appear on this view in the dashboard:



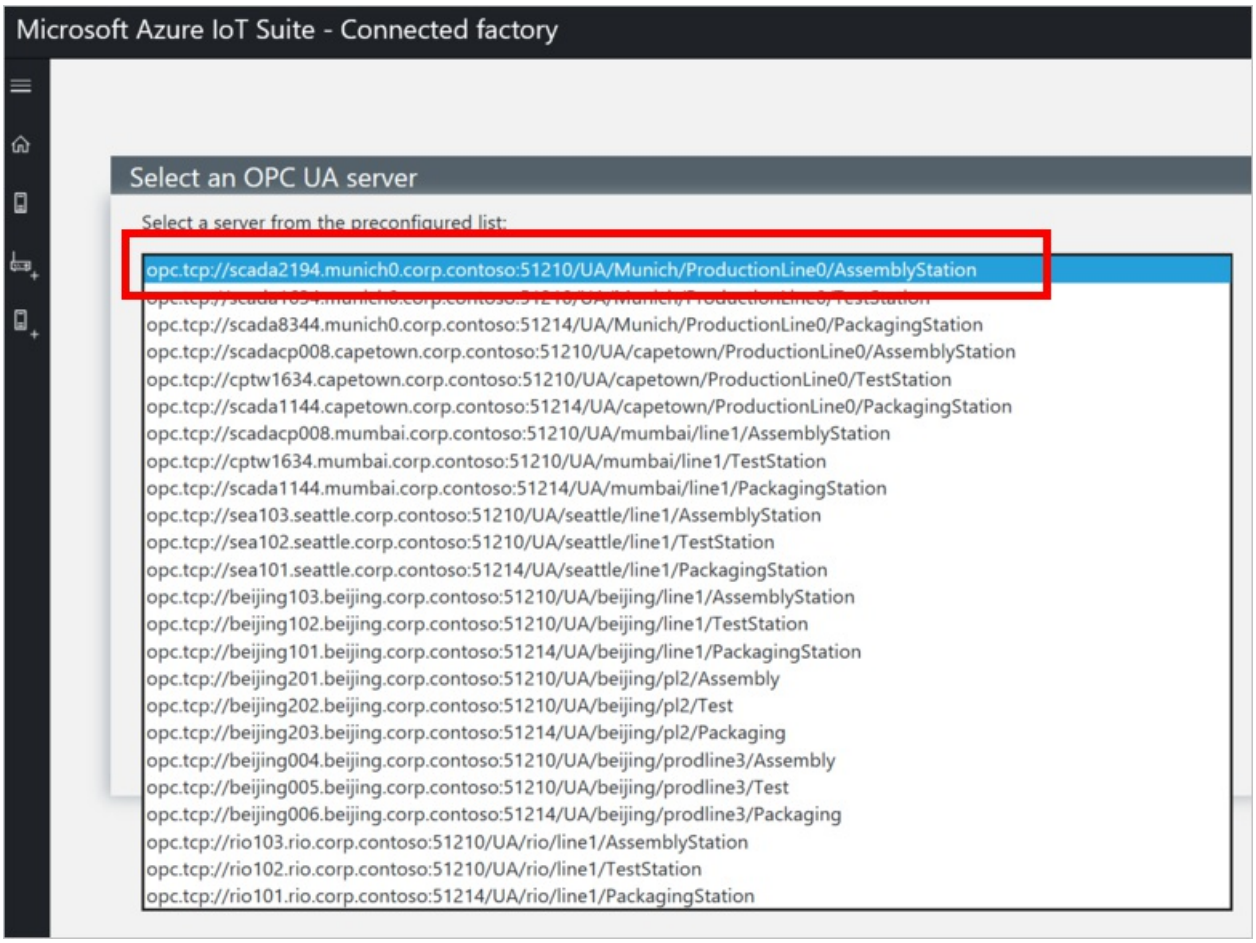

Each factory, production line, and station have an image property. You can find these JPEG files in the **Content\img** folder in the **WebApp** project. These image files display in the connected factory dashboard.

Each station includes several detailed properties that define the mapping from the OPC UA data items. These properties are described in the following sections:

**OpcUri**

The **OpcUri** value is the OPC UA Application URI that uniquely identifies the OPC UA server. For example, the **OpcUri** value for the assembly station on production line 1 in Munich looks like the following: **urn:scada2194:ua:munich:productionline0:assemblystation**.

You can view the URIs of the connected OPC UA servers in the solution dashboard:




**Simulation**

The information in the **Simulation** node is specific to the OPC UA simulation that runs in the OPC UA servers that are provisioned by default. It is not used for a real OPC UA server.

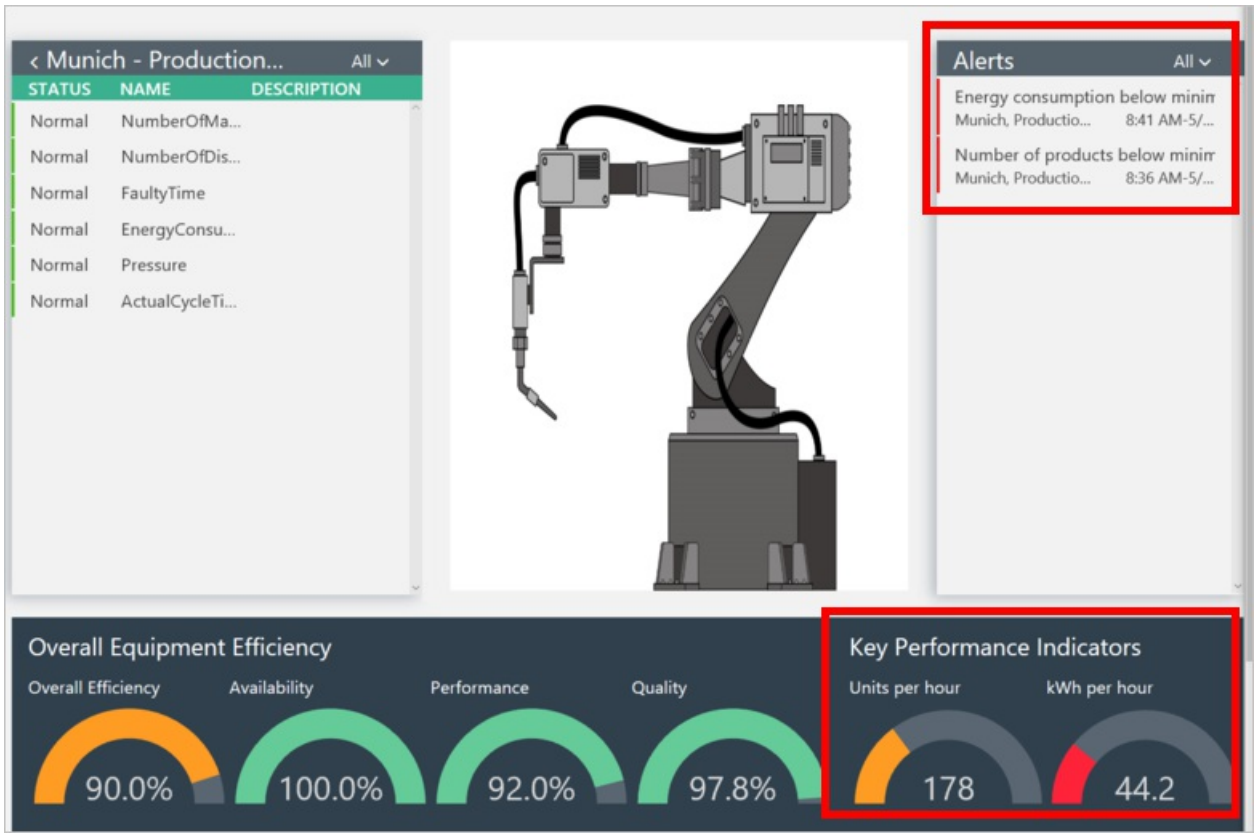
**Kpi1 and Kpi2**

These nodes describe how data from the station contributes to the two KPI values in the dashboard. In a default deployment, these KPI values are units per hour and kWh per hour. The solution calculates KPI vales at the level of a station and aggregates them at the production line and factory levels.

Each KPI has a minimum, maximum, and target value. Each KPI value can also define alert actions for the connected factory solution to perform. The following snippet shows the KPI definitions for the assembly station on production line 1 in Munich:

```
"Kpi1": {
  "Minimum": 150,
  "Target": 300,
  "Maximum": 600
},
"Kpi2": {
  "Minimum": 50,
  "Target": 100,
  "Maximum": 200,
  "MinimumAlertActions": [
    {
      "Type": "None"
    }
  ]
}
```

The following screenshot shows the KPI data in the dashboard.




**OpcNodes**

The **OpcNodes** nodes identify the published data items from the OPC UA server and specify how to process that data.

The **NodeId** value identifies the specific OPC UA NodeID from the OPC UA server. The first node in the assembly station for production line 1 in Munich has a value **ns=2;i=385**. A **NodeId** value specifies the data item to read from the OPC UA server, and the **SymbolicName** provides a user-friendly name to use in the dashboard for that data.

Other values associated with each node are summarized in the following table:

| VALUE | DESCRIPTION |
|---|---|
| Relevance | The KPI and OEE values this data contributes to. |
| OpCode | How the data is aggregated. |
| Units | The units to use in the dashboard. |
| Visible | Whether to display this value in the dashboard. Some values are used in calculations but not displayed. |
| Maximum | The maximum value that triggers an alert in the dashboard. |
| MaximumAlertActions | An action to take in response to an alert. For example, send a command to a station. |
| ConstValue | A constant value used in a calculation. |

## Deploy the changes

When you have finished making changes to the **ContosoTopologyDescription.json** file, you must redeploy the

connected factory solution to your Azure account.

The **azure-iot-connected-factory** repository includes a **build.ps1** PowerShell script you can use to rebuild and deploy the solution.

## Next Steps

Learn more about the connected factory preconfigured solution by reading the following articles:

- Connected factory preconfigured solution walkthrough
- Deploy a gateway for connected factory
- Permissions on the azureiotsuite.com site
- Connected factory FAQ
- FAQ

# Internet of Things security architecture

7/3/2017 • 24 min to read • Edit Online

When designing a system, it is important to understand the potential threats to that system, and add appropriate defenses accordingly, as the system is designed and architected. It is particularly important to design the product from the start with security in mind because understanding how an attacker might be able to compromise a system helps make sure appropriate mitigations are in place from the beginning.

## Security starts with a threat model

Microsoft has long used threat models for its products and has made the company's threat modeling process publically available. The company experience demonstrates that the modeling has unexpected benefits beyond the immediate understanding of what threats are the most concerning. For example, it also creates an avenue for an open discussion with others outside the development team, which can lead to new ideas and improvements in the product.

The objective of threat modeling is to understand how an attacker might be able to compromise a system and then make sure appropriate mitigations are in place. Threat modeling forces the design team to consider mitigations as the system is designed rather than after a system is deployed. This fact is critically important, because retrofitting security defenses to a myriad of devices in the field is infeasible, error prone and will leave customers at risk.

Many development teams do an excellent job capturing the functional requirements for the system that benefit customers. However, identifying non-obvious ways that someone might misuse the system is more challenging. Threat modeling can help development teams understand what an attacker might do and why. Threat modeling is a structured process that creates a discussion about the security design decisions in the system, as well as changes to the design that are made along the way that impact security. While a threat model is simply a document, this documentation also represents an ideal way to ensure continuity of knowledge, retention of lessons learned, and help new team onboard rapidly. Finally, an outcome of threat modeling is to enable you to consider other aspects of security, such as what security commitments you wish to provide to your customers. These commitments in conjunction with threat modeling will inform and drive testing of your Internet of Things (IoT) solution.

**When to threat model**

Threat modeling offers the greatest value if it is incorporated into the design phase. When you are designing, you have the greatest flexibility to make changes to eliminate threats. Eliminating threats by design is the desired outcome. It is much easier than adding mitigations, testing them, and ensuring they remain current and moreover, such elimination is not always possible. It becomes harder to eliminate threats as a product becomes more mature, and in turn will ultimately require more work and a lot harder tradeoffs than threat modeling early on in the development.

**What to threat model**

You should thread model the solution as a whole and also focus in the following areas:

- The security and privacy features
- The features whose failures are security relevant
- The features that touch a trust boundary

**Who threat models**

Threat modeling is a process like any other. It is a good idea to treat the threat model document like any other component of the solution and validate it. Many development teams do an excellent job capturing the functional requirements for the system that benefit customers. However, identifying non-obvious ways that someone might

misuse the system is more challenging. Threat modeling can help development teams understand what an attacker might do and why.

**How to threat model**

The threat modeling process is composed of four steps; the steps are:

- Model the application
- Enumerate Threats
- Mitigate threats
- Validate the mitigations

**The process steps**

Three rules of thumb to keep in mind when building a threat model:

1. Create a diagram out of reference architecture.
2. Start breadth-first. Get an overview, and understand the system as a whole, before deep-diving. This helps ensure that you deep-dive in the right places.
3. Drive the process, don't let the process drive you. If you find an issue in the modeling phase and want to explore it, go for it! Don't feel you need to follow these steps slavishly.

**Threats**

The four core elements of a threat model are:

- Processes (web services, Win32 services, *nix daemons, etc. Note that some complex entities (for example field gateways and sensors) can be abstracted as a process when a technical drill-down in these areas is not possible.
- Data stores (anywhere data is stored, such as a configuration file or database)
- Data flow (where data moves between other elements in the application)
- External Entities (anything that interacts with the system, but is not under the control of the application, examples include users and satellite feeds)

All elements in the architectural diagram are subject to various threats; we will use the STRIDE mnemonic. Read Threat Modeling Again, STRIDE to know more about the STRIDE elements.

Different elements of the application diagram are subject to certain STRIDE threats:

- Processes are subject to STRIDE
- Data flows are subject to TID
- Data stores are subject to TID, and sometimes R, if the data stores are log files.
- External entities are subject to SRD

# Security in IoT

Connected special-purpose devices have a significant number of potential interaction surface areas and interaction patterns, all of which must be considered to provide a framework for securing digital access to those devices. The term "digital access" is used here to distinguish from any operations that are carried out through direct device interaction where access security is provided through physical access control. For example, putting the device into a room with a lock on the door. While physical access cannot be denied using software and hardware, measures can be taken to prevent physical access from leading to system interference.

As we explore the interaction patterns, we will look at "device control" and "device data" with the same level of attention. "Device control" can be classified as any information that is provided to a device by any party with the goal of changing or influencing its behavior towards its state or the state of its environment. "Device data" can be classified as any information that a device emits to any other party about its state and the observed state of its environment.

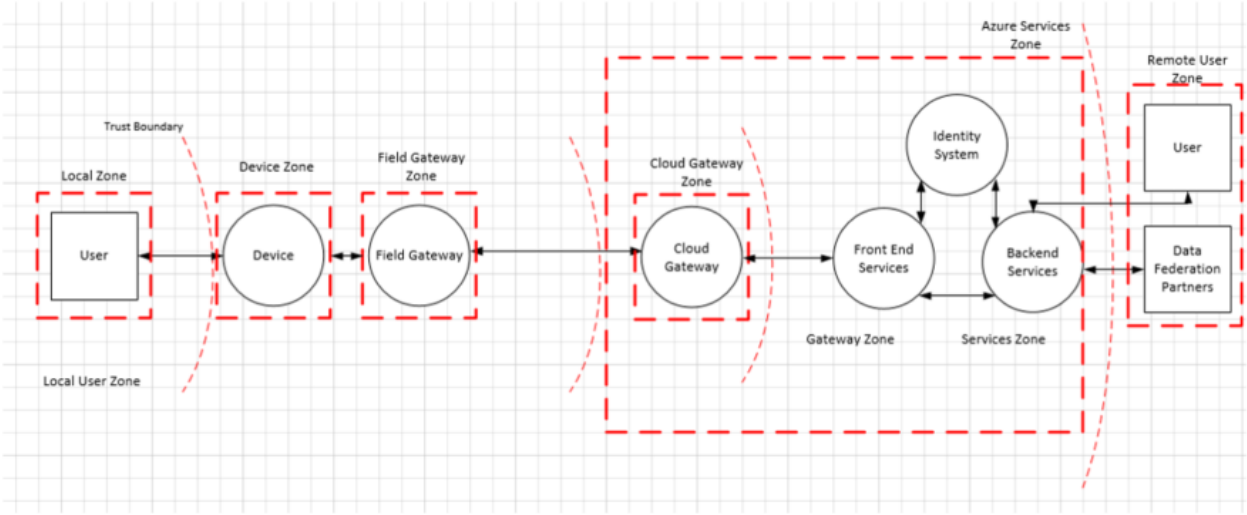
In order to optimize security best practices, it is recommended that a typical IoT architecture be divided into several

component/zones as part of the threat modeling exercise. These zones are described fully throughout this section and include:

- Device,
- Field Gateway,
- Cloud gateways, and
- Services.

Zones are broad way to segment a solution; each zone often has its own data and authentication and authorization requirements. Zones can also be used to isolation damage and restrict the impact of low trust zones on higher trust zones.

Each zone is separated by a Trust Boundary, which is noted as the dotted red line in the diagram below. It represents a transition of data/information from one source to another. During this transition, the data/information could be subject to Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege (STRIDE).




The components depicted within each boundary are also subjected to STRIDE, enabling a full 360 threat modeling view of the solution. The sections below elaborate on each of the components and specific security concerns and solutions that should be put into place.

The sections that follow will discuss standard components typically found in these zones.

**The Device Zone**

The device environment is the immediate physical space around the device where physical access and/or "local network" peer-to-peer digital access to the device is feasible. A "local network" is assumed to be a network that is distinct and insulated from – but potentially bridged to – the public Internet, and includes any short-range wireless radio technology that permits peer-to-peer communication of devices. It does *not* include any network virtualization technology creating the illusion of such a local network and it does also not include public operator networks that require any two devices to communicate across public network space if they were to enter a peer-to-peer communication relationship.

**The Field Gateway Zone**

Field gateway is a device/appliance or some general-purpose server computer software that acts as communication enabler and, potentially, as a device control system and device data processing hub. The field gateway zone includes the field gateway itself and all devices that are attached to it. As the name implies, field gateways act outside dedicated data processing facilities, are usually location bound, are potentially subject to physical intrusion, and will have limited operational redundancy. All to say that a field gateway is commonly a thing one can touch and sabotage while knowing what its function is.

A field gateway is different from a mere traffic router in that it has had an active role in managing access and information flow, meaning it is an application addressed entity and network connection or session terminal. An NAT device or firewall, in contrast, does not qualify as field gateways since they are not explicit connection or session terminals, but rather a route (or block) connections or sessions made through them. The field gateway has two distinct surface areas. One faces the devices that are attached to it and represents the inside of the zone, and the other faces all external parties and is the edge of the zone.

**The cloud gateway zone**

Cloud gateway is a system that enables remote communication from and to devices or field gateways from several different sites across public network space, typically towards a cloud-based control and data analysis system, a federation of such systems. In some cases, a cloud gateway may immediately facilitate access to special-purpose devices from terminals such as tablets or phones. In the context discussed here, "cloud" is meant to refer to a dedicated data processing system that is not bound to the same site as the attached devices or field gateways. Also in a Cloud Zone, operational measures prevent targeted physical access and are not necessarily exposed to a "public cloud" infrastructure.

A cloud gateway may potentially be mapped into a network virtualization overlay to insulate the cloud gateway and all of its attached devices or field gateways from any other network traffic. The cloud gateway itself is neither a device control system nor a processing or storage facility for device data; those facilities interface with the cloud gateway. The cloud gateway zone includes the cloud gateway itself along with all field gateways and devices directly or indirectly attached to it. The edge of the zone is a distinct surface area where all external parties communicate through.

**The services zone**

A "service" is defined for this context as any software component or module that is interfacing with devices through a field- or cloud gateway for data collection and analysis, as well as for command and control. Services are mediators. They act under their identity towards gateways and other subsystems, store and analyze data, autonomously issue commands to devices based on data insights or schedules and expose information and control capabilities to authorized end users.

**Information-devices vs. special-purpose devices**

PCs, phones, and tablets are primarily interactive information devices. Phones and tablets are explicitly optimized around maximizing battery lifetime. They preferably turn off partially when not immediately interacting with a person, or when not providing services like playing music or guiding their owner to a particular location. From a systems perspective, these information technology devices are mainly acting as proxies towards people. They are "people actuators" suggesting actions and "people sensors" collecting input.

Special-purpose devices, from simple temperature sensors to complex factory production lines with thousands of components inside them, are different. These devices are much more scoped in purpose and even if they provide some user interface, they are largely scoped to interfacing with or be integrated into assets in the physical world. They measure and report environmental circumstances, turn valves, control servos, sound alarms, switch lights, and do many other tasks. They help to do work for which an information device is either too generic, too expensive, too big, or too brittle. The concrete purpose immediately dictates their technical design as well the available monetary budget for their production and scheduled lifetime operation. The combination of these two key factors constrains the available operational energy budget, physical footprint, and thus available storage, compute, and security capabilities.

If something "goes wrong" with automated or remote controllable devices, for example, physical defects or control logic defects to willful unauthorized intrusion and manipulation. The production lots may be destroyed, buildings may be looted or burned down, and people may be injured or even die. This is, of course, a whole different class of damage than someone maxing out a stolen credit card's limit. The security bar for devices that make things move, and also for sensor data that eventually results in commands that cause things to move, must be higher than in any e-commerce or banking scenario.

**Device control and device data interactions**

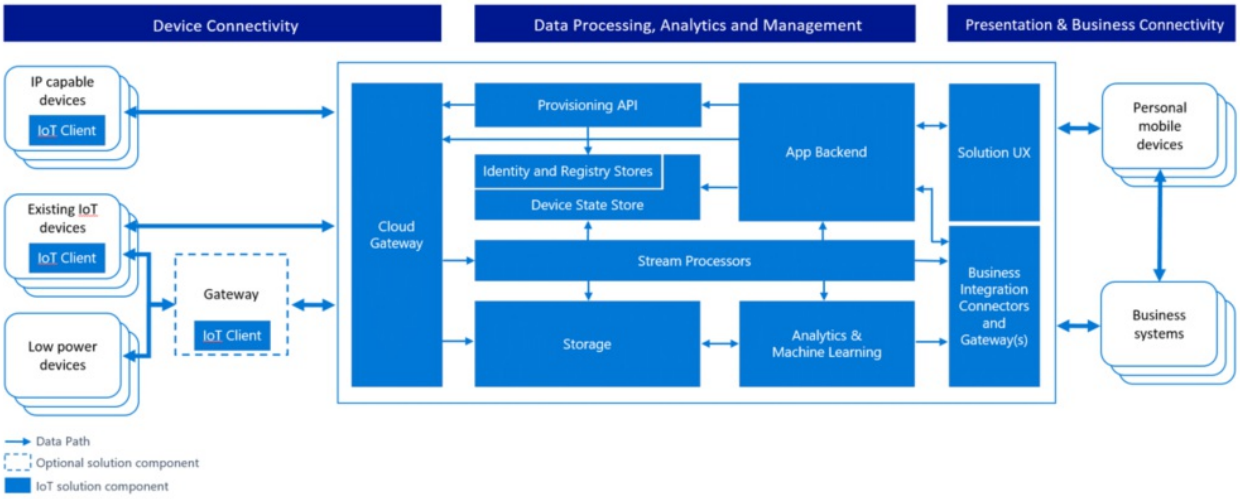
Connected special-purpose devices have a significant number of potential interaction surface areas and interaction patterns, all of which must be considered to provide a framework for securing digital access to those devices. The term "digital access" is used here to distinguish from any operations that are carried out through direct device interaction where access security is provided through physical access control. For example, putting the device into a room with a lock on the door. While physical access cannot be denied using software and hardware, measures can be taken to prevent physical access from leading to system interference.

As we explore the interaction patterns, we will look at "device control" and "device data" with the same level of attention while threat modeling. "Device control" can be classified as any information that is provided to a device by any party with the goal of changing or influencing its behavior towards its state or the state of its environment. "Device data" can be classified as any information that a device emits to any other party about its state and the observed state of its environment.

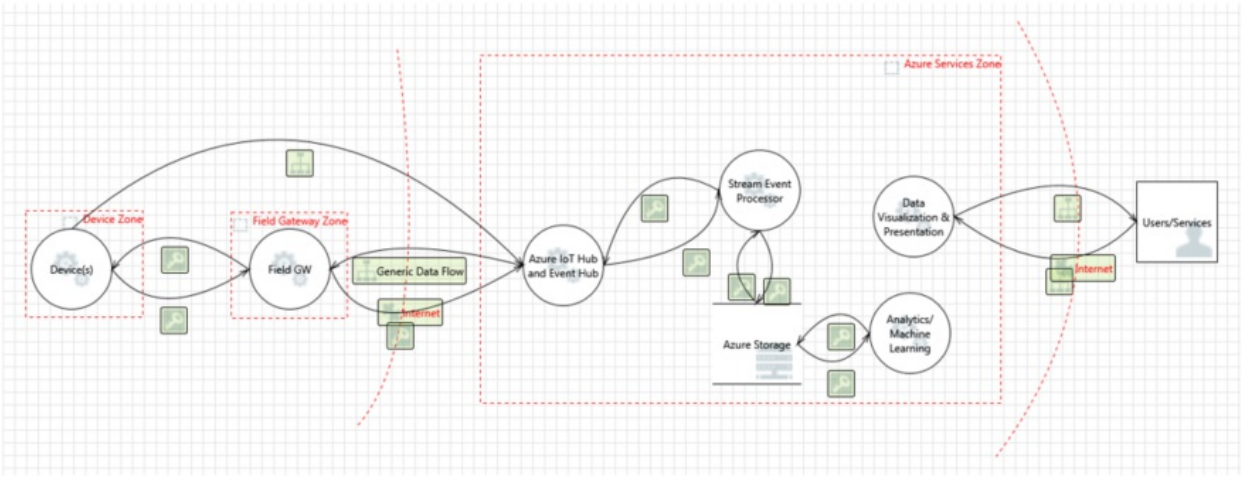
# Threat modeling the Azure IoT reference architecture

Microsoft uses the framework outlined above to do threat modeling for Azure IoT. In the section below we therefore use the concrete example of Azure IoT Reference Architecture to demonstrate how to think about threat modeling for IoT and how to address the threats identified. In our case we identified four main areas of focus:

- Devices and Data Sources,
- Data Transport,
- Device and Event Processing, and
- Presentation




The diagram below provides a simplified view of Microsoft's IoT Architecture using a Data Flow Diagram model that is used by the Microsoft Threat Modeling Tool:




It is important to note that the architecture separates the device and gateway capabilities. This allows the user to leverage gateway devices that are more secure: they are capable of communicating with the cloud gateway using secure protocols, which typically requires greater processing overhead that a native device - such as a thermostat - could provide on its own. In the Azure services zone, we assume that the Cloud Gateway is represented by the Azure IoT Hub service.

**Device and data sources/data transport**

This section explores the architecture outlined above through the lens of threat modeling and gives an overview of how we are addressing some of the inherent concerns. We will focus on the core elements of a threat model:

- Processes (those under our control and external items)
- Communication (also called data flows)
- Storage (also called data stores)

**Processes**

In each of the categories outlined in the Azure IoT architecture, we try to mitigate a number of different threats across the different stages data/information exists in: process, communication, and storage. Below we give an overview of the most common ones for the "process" category, followed by an overview of how these could be best mitigated:

**Spoofing (S)**: An attacker may extract cryptographic key material from a device, either at the software or hardware level, and subsequently access the system with a different physical or virtual device under the identity of the device the key material has been taken from. A good illustration is remote controls that can turn any TV and that are popular prankster tools.

**Denial of Service (D)**: A device can be rendered incapable of functioning or communicating by interfering with radio frequencies or cutting wires. For example, a surveillance camera that had its power or network connection intentionally knocked out will not report data, at all.

**Tampering (T)**: An attacker may partially or wholly replace the software running on the device, potentially allowing the replaced software to leverage the genuine identity of the device if the key material or the cryptographic facilities holding key materials were available to the illicit program. For example, an attacker may leverage extracted key material to intercept and suppress data from the device on the communication path and replace it with false data that is authenticated with the stolen key material.

**Information Disclosure (I)**: If the device is running manipulated software, such manipulated software could potentially leak data to unauthorized parties. For example, an attacker may leverage extracted key material to inject itself into the communication path between the device and a controller or field gateway or cloud gateway to siphon off information.

**Elevation of Privilege (E)**: A device that does specific function can be forced to do something else. For example, a valve that is programmed to open half way can be tricked to open all the way.

| COMPONENT | THREAT | MITIGATION | RISK | IMPLEMENTATION |
|---|---|---|---|---|
| Device | S | Assigning identity to the device and authenticating the device | Replacing device or part of the device with some other device. How do we know we are talking to the right device? | Authenticating the device, using Transport Layer Security (TLS) or IPSec. Infrastructure should support using pre-shared key (PSK) on those devices that cannot handle full asymmetric cryptography. Leverage Azure AD, OAuth |
| TRID | Apply tamperproof mechanisms to the device for example by making it very hard to impossible to extract keys and other cryptographic material from the device. | The risk is if someone is tampering the device (physical interference). How are we sure, that device has not tampered with. | The most effective mitigation is a trusted platform module (TPM) capability that allows storing keys in special on-chip circuitry from which the keys cannot be read, but can only be used for cryptographic operations that use the key but never disclose the key. Memory encryption of the device. Key management for the device. Signing the code. | |
| E | Having access control of the device. Authorization scheme. | If the device allows for individual actions to be performed based on commands from an outside source, or even compromised sensors, it will allow the attack to perform operations not otherwise accessible. | Having authorization scheme for the device | |
| Field Gateway | S | Authenticating the Field gateway to Cloud Gateway (cert based, PSK, Claim based,..) | If someone can spoof Field Gateway, then it can present itself as any device. | TLS RSA/PSK, IPSec, RFC 4279. All the same key storage and attestation concerns of devices in general – best case is use TPM. 6LowPAN extension for IPSec to support Wireless Sensor Networks (WSN). |

| COMPONENT | THREAT | MITIGATION | RISK | IMPLEMENTATION |
|---|---|---|---|---|
| TRID | Protect the Field Gateway against tampering (TPM?) | Spoofing attacks that trick the cloud gateway thinking it is talking to field gateway could result in information disclosure and data tampering | Memory encryption, TPM's, authentication. | |
| E | Access control mechanism for Field Gateway | | | |

Here are some examples of threats in this category:

Spoofing: An attacker may extract cryptographic key material from a device, either at the software or hardware level, and subsequently access the system with a different physical or virtual device under the identity of the device the key material has been taken from.

**Denial of Service**: A device can be rendered incapable of functioning or communicating by interfering with radio frequencies or cutting wires. For example, a surveillance camera that had its power or network connection intentionally knocked out will not report data, at all.

**Tampering**: An attacker may partially or wholly replace the software running on the device, potentially allowing the replaced software to leverage the genuine identity of the device if the key material or the cryptographic facilities holding key materials were available to the illicit program.

**Tampering**: A surveillance camera that's showing a visible-spectrum picture of an empty hallway could be aimed at a photograph of such a hallway. A smoke or fire sensor could be reporting someone holding a lighter under it. In either case, the device may be technically fully trustworthy towards the system, but it will report manipulated information.

**Tampering**: An attacker may leverage extracted key material to intercept and suppress data from the device on the communication path and replace it with false data that is authenticated with the stolen key material.

**Tampering**: An attacker may partially or completely replace the software running on the device, potentially allowing the replaced software to leverage the genuine identity of the device if the key material or the cryptographic facilities holding key materials were available to the illicit program.

**Information Disclosure**: If the device is running manipulated software, such manipulated software could potentially leak data to unauthorized parties.

**Information Disclosure**: An attacker may leverage extracted key material to inject itself into the communication path between the device and a controller or field gateway or cloud gateway to siphon off information.

**Denial of Service**: The device can be turned off or turned into a mode where communication is not possible (which is intentional in many industrial machines).

**Tampering**: The device can be reconfigured to operate in a state unknown to the control system (outside of known calibration parameters) and thus provide data that can be misinterpreted

**Elevation of Privilege**: A device that does specific function can be forced to do something else. For example, a valve that is programmed to open half way can be tricked to open all the way.

**Denial of Service**: The device can be turned into a state where communication is not possible.

**Tampering**: The device can be reconfigured to operate in a state unknown to the control system (outside of known

calibration parameters) and thus provide data that can be misinterpreted.

**Spoofing/Tampering/Repudiation**: If not secured (which is rarely the case with consumer remote controls) an attacker can manipulate the state of a device anonymously. A good illustration is remote controls that can turn any TV and that are popular prankster tools.

#### Communication

Threats around communication path between devices, devices and field gateways and device and cloud gateway. The table below has some guidance around open sockets on the device/VPN:

| COMPONENT | THREAT | MITIGATION | RISK | IMPLEMENTATION |
|---|---|---|---|---|
| Device IoT Hub | TID | (D)TLS (PSK/RSA) to encrypt the traffic | Eavesdropping or interfering the communication between the device and the gateway | Security on the protocol level. With custom protocols, we need to figure out how to protect them. In most cases, the communication takes place from the device to the IoT Hub (device initiates the connection). |
| Device Device | TID | (D)TLS (PSK/RSA) to encrypt the traffic. | Reading data in transit between devices. Tampering with the data. Overloading the device with new connections | Security on the protocol level (MQTT/AMQP/HTTP/CoAP. With custom protocols, we need to figure out how to protect them. The mitigation for the DoS threat is to peer devices through a cloud or field gateway and have them only act as clients towards the network. The peering may result in a direct connection between the peers after having been brokered by the gateway |
| External Entity Device | TID | Strong pairing of the external entity to the device | Eavesdropping the connection to the device. Interfering the communication with the device | Securely pairing the external entity to the device NFC/Bluetooth LE. Controlling the operational panel of the device (Physical) |
| Field Gateway Cloud Gateway | TID | TLS (PSK/RSA) to encrypt the traffic. | Eavesdropping or interfering the communication between the device and the gateway | Security on the protocol level (MQTT/AMQP/HTTP/CoAP). With custom protocols, we need to figure out how to protect them. |

| COMPONENT | THREAT | MITIGATION | RISK | IMPLEMENTATION |
|---|---|---|---|---|
| Device Cloud Gateway | TID | TLS (PSK/RSA) to encrypt the traffic. | Eavesdropping or interfering the communication between the device and the gateway | Security on the protocol level (MQTT/AMQP/HTTP/CoAP). With custom protocols, we need to figure out how to protect them. |

Here are some examples of threats in this category:

**Denial of Service**: Constrained devices are generally under DoS threat when they actively listen for inbound connections or unsolicited datagrams on a network, because an attacker can open many connections in parallel and not service them or service them very slowly, or the device can be flooded with unsolicited traffic. In both cases, the device can effectively be rendered inoperable on the network.

**Spoofing, Information Disclosure**: Constrained devices and special-purpose devices often have one-for-all security facilities like password or PIN protection, or they wholly rely on trusting the network, meaning they will grant access to information when a device is on the same network, and that network is often only protected by a shared key. That means that when the shared secret to device or network is disclosed, it is possible to control the device or observe data emitted from the device.

**Spoofing**: an attacker may intercept or partially override the broadcast and spoof the originator (man in the middle)

**Tampering**: an attacker may intercept or partially override the broadcast and send false information

**Information Disclosure:** an attacker may eavesdrop on a broadcast and obtain information without authorization
**Denial of Service:** an attacker may jam the broadcast signal and deny information distribution

#### Storage

Every device and field gateway has some form of storage (temporary for queuing the data, operating system (OS) image storage).

| COMPONENT | THREAT | MITIGATION | RISK | IMPLEMENTATION |
|---|---|---|---|---|
| Device storage | TRID | Storage encryption, signing the logs | Reading data from the storage (PII data), tampering with telemetry data. Tampering with queued or cached command control data. Tampering with configuration or firmware update packages while cached or queued locally can lead to OS and/or system components being compromised | Encryption, message authentication code (MAC) or digital signature. Where possible, strong access control through resource access control lists (ACLs) or permissions. |
| Device OS image | TRID | | Tampering with OS /replacing the OS components | Read-only OS partition, signed OS image, Encryption |

| COMPONENT | THREAT | MITIGATION | RISK | IMPLEMENTATION |
|---|---|---|---|---|
| Field Gateway storage (queuing the data) | TRID | Storage encryption, signing the logs | Reading data from the storage (PII data), tampering with telemetry data, tampering with queued or cached command control data. Tampering with configuration or firmware update packages (destined for devices or field gateway) while cached or queued locally can lead to OS and/or system components being compromised | BitLocker |
| Field Gateway OS image | TRID | | Tampering with OS /replacing the OS components | Read-only OS partition, signed OS image, Encryption |

**Device and event processing/cloud gateway zone**

A cloud gateway is system that enables remote communication from and to devices or field gateways from several different sites across public network space, typically towards a cloud-based control and data analysis system, a federation of such systems. In some cases, a cloud gateway may immediately facilitate access to special-purpose devices from terminals such as tablets or phones. In the context discussed here, "cloud" is meant to refer to a dedicated data processing system that is not bound to the same site as the attached devices or field gateways, and where operational measures prevent targeted physical access but is not necessarily to a "public cloud" infrastructure. A cloud gateway may potentially be mapped into a network virtualization overlay to insulate the cloud gateway and all of its attached devices or field gateways from any other network traffic. The cloud gateway itself is neither a device control system nor a processing or storage facility for device data; those facilities interface with the cloud gateway. The cloud gateway zone includes the cloud gateway itself along with all field gateways and devices directly or indirectly attached to it.

Cloud gateway is mostly custom built piece of software running as a service with exposed endpoints to which field gateway and devices connect. As such it must be designed with security in mind. Please follow SDL process for designing and building this service.

**Services zone**

A control system (or controller) is a software solution that interfaces with a device, or a field gateway, or cloud gateway for the purpose of controlling one or multiple devices and/or to collect and/or store and/or analyze device data for presentation, or subsequent control purposes. Control systems are the only entities in the scope of this discussion that may immediately facilitate interaction with people. The exception is intermediate physical control surfaces on devices, like a switch that allows a person to turn the device off or change other properties, and for which there is no functional equivalent that can be accessed digitally.

Intermediate physical control surfaces are those where any sort of governing logic constrains the function of the physical control surface such that an equivalent function can be initiated remotely or input conflicts with remote input can be avoided – such intermediated control surfaces are conceptually attached to a local control system that leverages the same underlying functionality as any other remote control system that the device may be attached to in parallel. Top threats to the cloud computing can be read at Cloud Security Alliance (CSA) page.

# Additional resources

Refer to the following articles for additional information:

- SDL Threat Modeling Tool
- Microsoft Azure IoT reference architecture

## See also

To learn more about securing your IoT solution, see Secure your IoT deployment.

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Frequently asked questions for IoT Suite

You can read about IoT Hub security in Control access to IoT Hub in the IoT Hub developer guide.

# Internet of Things security best practices

7/3/2017 • 6 min to read • Edit Online

To secure an Internet of Things (IoT) infrastructure requires a rigorous security-in-depth strategy. This strategy requires you to secure data in the cloud, protect data integrity while in transit over the public internet, and securely provision devices. Each layer builds greater security assurance in the overall infrastructure.

## Secure an IoT infrastructure

This security-in-depth strategy can be developed and executed with active participation of various players involved with the manufacturing, development, and deployment of IoT devices and infrastructure. Following is a high-level description of these players.

- **IoT hardware manufacturer/integrator**: Typically, these are the manufacturers of IoT hardware being deployed, integrators assembling hardware from various manufacturers, or suppliers providing hardware for an IoT deployment manufactured or integrated by other suppliers.
- **IoT solution developer**: The development of an IoT solution is typically done by a solution developer. This developer may part of an in-house team or a system integrator (SI) specializing in this activity. The IoT solution developer can develop various components of the IoT solution from scratch, integrate various off-the-shelf or open-source components, or adopt preconfigured solutions with minor adaptation.
- **IoT solution deployer**: After an IoT solution is developed, it needs to be deployed in the field. This involves deployment of hardware, interconnection of devices, and deployment of solutions in hardware devices or the cloud.
- **IoT solution operator**: After the IoT solution is deployed, it requires long-term operations, monitoring, upgrades, and maintenance. This can be done by an in-house team that comprises information technology specialists, hardware operations and maintenance teams, and domain specialists who monitor the correct behavior of overall IoT infrastructure.

The sections that follow provide best practices for each of these players to help develop, deploy, and operate a secure IoT infrastructure.

## IoT hardware manufacturer/integrator

The following are the best practices for IoT hardware manufacturers and hardware integrators.

- **Scope hardware to minimum requirements**: The hardware design should include the minimum features required for operation of the hardware, and nothing more. An example is to include USB ports only if necessary for the operation of the device. These additional features open the device for unwanted attack vectors that should be avoided.
- **Make hardware tamper proof**: Build in mechanisms to detect physical tampering, such as opening of the device cover or removing a part of the device. These tamper signals may be part of the data stream uploaded to the cloud, which could alert operators of these events.
- **Build around secure hardware**: If COGS permits, build security features such as secure and encrypted storage, or boot functionality based on Trusted Platform Module (TPM). These features make devices more secure and help protect the overall IoT infrastructure.
- **Make upgrades secure**: Firmware upgrades during the lifetime of the device are inevitable. Building devices with secure paths for upgrades and cryptographic assurance of firmware versions will allow the device to be secure during and after upgrades.

## IoT solution developer

The following are the best practices for IoT solution developers:

- **Follow secure software development methodology**: Development of secure software requires ground-up thinking about security, from the inception of the project all the way to its implementation, testing, and deployment. The choices of platforms, languages, and tools are all influenced with this methodology. The Microsoft Security Development Lifecycle provides a step-by-step approach to building secure software.
- **Choose open-source software with care**: Open-source software provides an opportunity to quickly develop solutions. When you're choosing open-source software, consider the activity level of the community for each open-source component. An active community ensures that software is supported and that issues are discovered and addressed. Alternatively, an obscure and inactive open-source software might not be supported and issues will probably not be discovered.
- **Integrate with care**: Many software security flaws exist at the boundary of libraries and APIs. Functionality that may not be required for the current deployment might still be available via an API layer. To ensure overall security, make sure to check all interfaces of components being integrated for security flaws.

## IoT solution deployer

The following are best practices for IoT solution deployers:

- **Deploy hardware securely**: IoT deployments may require hardware to be deployed in unsecure locations, such as in public spaces or unsupervised locales. In such situations, ensure that hardware deployment is tamper-proof to the maximum extent. If USB or other ports are available on the hardware, ensure that they are covered securely. Many attack vectors can use these as entry points.
- **Keep authentication keys safe**: During deployment, each device requires device IDs and associated authentication keys generated by the cloud service. Keep these keys physically safe even after the deployment. Any compromised key can be used by a malicious device to masquerade as an existing device.

## IoT solution operator

The following are the best practices for IoT solution operators:

- **Keep the system up to date**: Ensure that device operating systems and all device drivers are upgraded to the latest versions. If you turn on automatic updates in Windows 10 (IoT or other SKUs), Microsoft keeps it up to date, providing a secure operating system for IoT devices. Keeping other operating systems (such as Linux) up to date helps ensure that they are also protected against malicious attacks.
- **Protect against malicious activity**: If the operating system permits, install the latest antivirus and antimalware capabilities on each device operating system. This can help mitigate most external threats. You can protect most modern operating systems against threats by taking appropriate steps.
- **Audit frequently**: Auditing IoT infrastructure for security-related issues is key when responding to security incidents. Most operating systems provide built-in event logging that should be reviewed frequently to make sure no security breach has occurred. Audit information can be sent as a separate telemetry stream to the cloud service where it can be analyzed.
- **Physically protect the IoT infrastructure**: The worst security attacks against IoT infrastructure are launched using physical access to devices. One important safety practice is to protect against malicious use of USB ports and other physical access. One key to uncovering breaches that might have occurred is logging of physical access, such as USB port use. Again, Windows 10 (IoT and other SKUs) enables detailed logging of these events.
- **Protect cloud credentials**: Cloud authentication credentials used for configuring and operating an IoT deployment are possibly the easiest way to gain access and compromise an IoT system. Protect the credentials by changing the password frequently, and refrain from using these credentials on public machines.

Capabilities of different IoT devices vary. Some devices might be computers running common desktop operating

systems, and some devices might be running very light-weight operating systems. The security best practices described previously might be applicable to these devices in varying degrees. If provided, additional security and deployment best practices from the manufacturers of these devices should be followed.

Some legacy and constrained devices might not have been designed specifically for IoT deployment. These devices might lack the capability to encrypt data, connect with the Internet, or provide advanced auditing. In these cases, a modern and secure field gateway can aggregate data from legacy devices and provide the security required for connecting these devices over the Internet. Field gateways can provide secure authentication, negotiation of encrypted sessions, receipt of commands from the cloud, and many other security features.

## See also

To learn more about securing your IoT solution, see:

- IoT security architecture
- Secure your IoT deployment

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Frequently asked questions for Azure IoT Suite

You can read about IoT Hub security in Control access to IoT Hub in the IoT Hub developer guide.

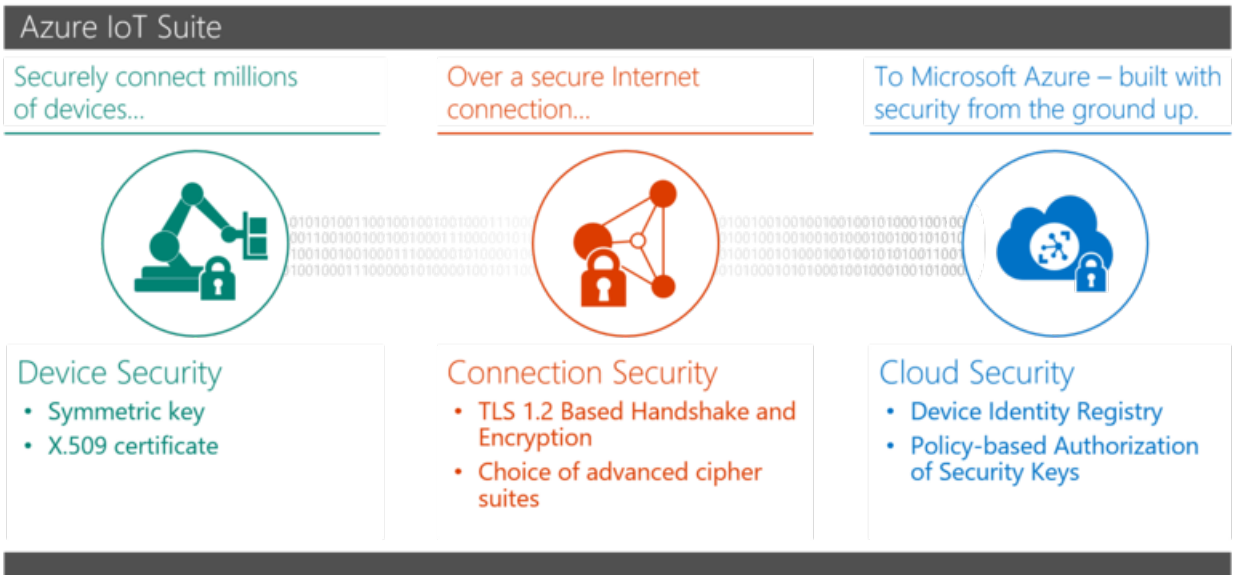
# Secure your IoT deployment

8/24/2017 • 7 min to read • Edit Online

This article provides the next level of detail for securing the Azure IoT-based Internet of Things (IoT) infrastructure. It links to implementation level details for configuring and deploying each component. It also provides comparisons and choices between various competing methods.

Securing the Azure IoT deployment can be divided into the following three security areas:

- **Device Security**: Securing the IoT device while it is deployed in the wild.
- **Connection Security**: Ensuring all data transmitted between the IoT device and IoT Hub is confidential and tamper-proof.
- **Cloud Security**: Providing a means to secure data while it moves through, and is stored in the cloud.




## Secure device provisioning and authentication

The Azure IoT Suite secures IoT devices by the following two methods:

- By providing a unique identity key (security tokens) for each device, which can be used by the device to communicate with the IoT Hub.
- By using an on-device X.509 certificate and private key as a means to authenticate the device to the IoT Hub. This authentication method ensures that the private key on the device is not known outside the device at any time, providing a higher level of security.

The security token method provides authentication for each call made by the device to IoT Hub by associating the symmetric key to each call. X.509-based authentication allows authentication of an IoT device at the physical layer as part of the TLS connection establishment. The security-token-based method can be used without the X.509 authentication which is a less secure pattern. The choice between the two methods is primarily dictated by how secure the device authentication needs to be, and availability of secure storage on the device (to store the private key securely).

## IoT Hub security tokens

IoT Hub uses security tokens to authenticate devices and services to avoid sending keys on the network. Additionally, security tokens are limited in time validity and scope. Azure IoT SDKs automatically generate tokens

without requiring any special configuration. Some scenarios, however, require the user to generate and use security tokens directly. These include the direct use of the MQTT, AMQP, or HTTP surfaces, or the implementation of the token service pattern.

More details on the structure of the security token and its usage can be found in the following articles:

- Security token structure
- Using SAS tokens as a device

Each IoT Hub has an identity registry that can be used to create per-device resources in the service, such as a queue that contains in-flight cloud-to-device messages, and to allow access to the device-facing endpoints. The IoT Hub identity registry provides secure storage of device identities and security keys for a solution. Individual or groups of device identities can be added to an allow list, or a block list, enabling complete control over device access. The following articles provide more details on the structure of the identity registry and supported operations.

IoT Hub supports protocols such as MQTT, AMQP, and HTTP. Each of these protocols use security tokens from the IoT device to IoT Hub differently:

- AMQP: SASL PLAIN and AMQP Claims-based security ({policyName}@sas.root.{iothubName} in the case of IoT hub-level tokens; {deviceId} in case of device-scoped tokens).
- MQTT: CONNECT packet uses {deviceId} as the {ClientId}, {IoThubhostname}/{deviceId} in the **Username** field and a SAS token in the **Password** field.
- HTTP: Valid token is in the authorization request header.

IoT Hub identity registry can be used to configure per-device security credentials and access control. However, if an IoT solution already has a significant investment in a custom device identity registry and/or authentication scheme, it can be integrated into an existing infrastructure with IoT Hub by creating a token service.

**X.509 certificate-based device authentication**

The use of a device-based X.509 certificate and its associated private and public key pair allows additional authentication at the physical layer. The private key is stored securely in the device and is not discoverable outside the device. The X.509 certificate contains information about the device, such as device ID, and other organizational details. A signature of the certificate is generated by using the private key.

High-level device provisioning flow:

- Associate an identifier to a physical device – device identity and/or X.509 certificate associated to the device during device manufacturing or commissioning.
- Create a corresponding identity entry in IoT Hub – device identity and associated device information in the IoT Hub identity registry.
- Securely store X.509 certificate thumbprint in IoT Hub identity registry.

**Root certificate on device**

While establishing a secure TLS connection with IoT Hub, the IoT device authenticates IoT Hub using a root certificate which is part of the device SDK. For the C client SDK the certificate is located under the folder "\c\certs" under the root of the repo. Though these root certificates are long-lived, they still may expire or be revoked. If there is no way of updating the certificate on the device, the device may not be able to subsequently connect to the IoT Hub (or any other cloud service). Having a means to update the root certificate once the IoT device is deployed will effectively mitigate this risk.

## Securing the connection

Internet connection between the IoT device and IoT Hub is secured using the Transport Layer Security (TLS) standard. Azure IoT supports TLS 1.2, TLS 1.1 and TLS 1.0, in this order. Support for TLS 1.0 is provided for

backward compatibility only. It is recommended to use TLS 1.2 since it provides the most security.

Azure IoT Suite supports the following Cipher Suites, in this order.

| CIPHER SUITE | LENGTH |
| --- | --- |
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028) ECDH secp384r1 (eq. 7680 bits RSA) FS | 256 |
| TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027) ECDH secp256r1 (eq. 3072 bits RSA) FS | 128 |
| TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) ECDH secp384r1 (eq. 7680 bits RSA) FS | 256 |
| TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) ECDH secp256r1 (eq. 3072 bits RSA) FS | 128 |
| TLS_RSA_WITH_AES_256_GCM_SHA384 (0x9d) | 256 |
| TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c) | 128 |
| TLS_RSA_WITH_AES_256_CBC_SHA256 (0x3d) | 256 |
| TLS_RSA_WITH_AES_128_CBC_SHA256 (0x3c) | 128 |
| TLS_RSA_WITH_AES_256_CBC_SHA (0x35) | 256 |
| TLS_RSA_WITH_AES_128_CBC_SHA (0x2f) | 128 |
| TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa) | 112 |

## Securing the cloud

Azure IoT Hub allows definition of access control policies for each security key. It uses the following set of permissions to grant access to each of IoT Hub's endpoints. Permissions limit the access to an IoT Hub based on functionality.

- **RegistryRead**. Grants read access to the identity registry. For more information, see identity registry.
- **RegistryReadWrite**. Grants read and write access to the identity registry. For more information, see identity registry.
- **ServiceConnect**. Grants access to cloud service-facing communication and monitoring endpoints. For example, it grants permission to back-end cloud services to receive device-to-cloud messages, send cloud-to-device messages, and retrieve the corresponding delivery acknowledgments.
- **DeviceConnect**. Grants access to device-facing endpoints. For example, it grants permission to send device-to-cloud messages and receive cloud-to-device messages. This permission is used by devices.

There are two ways to obtain **DeviceConnect** permissions with IoT Hub with security tokens: using a device identity key, or a shared access key. Moreover, it is important to note that all functionality accessible from devices is exposed by design on endpoints with prefix `/devices/{deviceId}`.

Service components can only generate security tokens using shared access policies granting the appropriate permissions.

Azure IoT Hub and other services which may be part of the solution allow management of users using the Azure

Active Directory.

Data ingested by Azure IoT Hub can be consumed by a variety of services such as Azure Stream Analytics and Azure blob storage. These services allow management access. Read more about these services and available options below:

- Azure Cosmos DB: A scalable, fully-indexed database service for semi-structured data that manages metadata for the devices you provision, such as attributes, configuration, and security properties. Cosmos DB offers high-performance and high-throughput processing, schema-agnostic indexing of data, and a rich SQL query interface.
- Azure Stream Analytics: Real-time stream processing in the cloud that enables you to rapidly develop and deploy a low-cost analytics solution to uncover real-time insights from devices, sensors, infrastructure, and applications. The data from this fully-managed service can scale to any volume while still achieving high throughput, low latency, and resiliency.
- Azure App Services: A cloud platform to build powerful web and mobile apps that connect to data anywhere; in the cloud or on-premises. Build engaging mobile apps for iOS, Android, and Windows. Integrate with your Software as a Service (SaaS) and enterprise applications with out-of-the-box connectivity to dozens of cloud-based services and enterprise applications. Code in your favorite language and IDE (.NET, Node.js, PHP, Python, or Java) to build web apps and APIs faster than ever.
- Logic Apps: The Logic Apps feature of Azure App Service helps integrate your IoT solution to your existing line-of-business systems and automate workflow processes. Logic Apps enables developers to design workflows that start from a trigger and then execute a series of steps—rules and actions that use powerful connectors to integrate with your business processes. Logic Apps offers out-of-the-box connectivity to a vast ecosystem of SaaS, cloud-based, and on-premises applications.
- Azure blob storage: Reliable, economical cloud storage for the data that your devices send to the cloud.

## Conclusion

This article provides overview of implementation level details for designing and deploying an IoT infrastructure using Azure IoT. Configuring each component to be secure is key in securing the overall IoT infrastructure. The design choices available in Azure IoT provide some level of flexibility and choice; however, each choice may have security implications. It is recommended that each of these choices be evaluated through a risk/cost assessment.

## See also

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Frequently asked questions for IoT Suite

You can read about IoT Hub security in Control access to IoT Hub in the IoT Hub developer guide.

# Internet of Things security from the ground up

8/24/2017 • 12 min to read • Edit Online

The Internet of Things (IoT) poses unique security, privacy, and compliance challenges to businesses worldwide. Unlike traditional cyber technology where these issues revolve around software and how it is implemented, IoT concerns what happens when the cyber and the physical worlds converge. Protecting IoT solutions requires ensuring secure provisioning of devices, secure connectivity between these devices and the cloud, and secure data protection in the cloud during processing and storage. Working against such functionality, however, are resource-constrained devices, geographic distribution of deployments, and a large number of devices within a solution.

This article explores how the Microsoft Azure IoT Suite provides a secure and private Internet of Things cloud solution. The Azure IoT Suite delivers a complete end-to-end solution, with security built into every stage from the ground up. At Microsoft, developing secure software is part of the software engineering practice, rooted in our decades long experience of developing secure software. To ensure this, Security Development Lifecycle (SDL) is the foundational development methodology, coupled with a host of infrastructure-level security services such as Operational Security Assurance (OSA) and the Microsoft Digital Crimes Unit, Microsoft Security Response Center, and Microsoft Malware Protection Center.

The Azure IoT Suite offers unique features, which make provisioning, connecting to, and storing data from IoT devices easy and transparent and, most of all, secure. In this article, we examine the Azure IoT Suite security features and deployment strategies to ensure security, privacy, and compliance challenges are addressed.

## Introduction

The Internet of Things (IoT) is the wave of the future, offering businesses immediate and real-world opportunities to reduce costs, increase revenue, and transform their business. Many businesses, however, are hesitant to deploy IoT in their organizations due to concerns about security, privacy, and compliance. A major point of concern comes from the uniqueness of the IoT infrastructure, which merges the cyber and physical worlds together, compounding individual risks inherent in these two worlds. Security of IoT pertains to ensuring the integrity of code running on devices, providing device and user authentication, defining clear ownership of devices (as well as data generated by those devices), and being resilient to cyber and physical attacks.

Then, there's the issue of privacy. Companies want transparency concerning data collection, as in what's being collected and why, who can see it, who controls access, and so on. Finally, there are general safety issues of the equipment along with the people operating them, and issues of maintaining industry standards of compliance.

Given the security, privacy, transparency, and compliance concerns, choosing the right IoT solution provider remains a challenge. Stitching together individual pieces of IoT software and services provided by a variety of vendors introduces gaps in security, privacy, transparency, and compliance which may be hard to detect, let alone fix. The choice of the right IoT software and service provider is based on finding providers which have extensive experience running services which span across verticals and geographies, but are also able to scale in a secure and transparent fashion. Similarly, it helps for the selected provider to have decades of experience with developing secure software running on billions of machines worldwide, and have the ability to appreciate the threat landscape posed by this new world of the Internet of Things.

## Secure infrastructure from the ground up

The Microsoft Cloud infrastructure supports more than one billion customers in 127 countries. Drawing on our decades-long experience building enterprise software and running some of the largest online services in the world, we provide higher levels of enhanced security, privacy, compliance, and threat mitigation practices than most customers could achieve on their own.

Our Security Development Lifecycle (SDL) provides a mandatory company-wide development process that embeds security requirements into the entire software lifecycle. To help ensure that operational activities follow the same level of security practices, we use rigorous security guidelines laid out in our Operational Security Assurance (OSA) process. We also work with third-party audit firms for ongoing verification that we meet our compliance obligations, and we engage in broad security efforts through the creation of centers of excellence, including the Microsoft Digital Crimes Unit, Microsoft Security Response Center, and Microsoft Malware Protection Center.

## Microsoft Azure - secure IoT infrastructure for your business

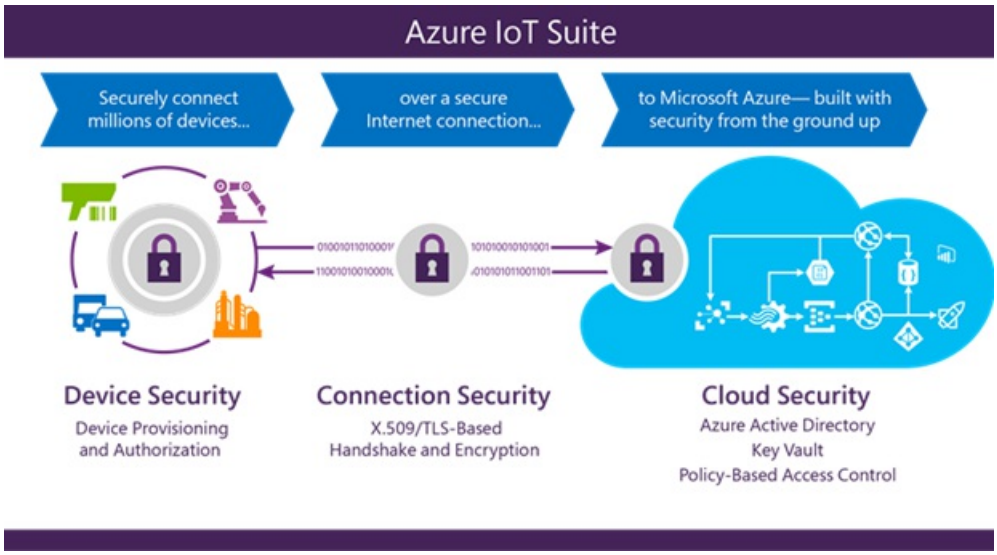
Microsoft Azure offers a complete cloud solution, one that combines a constantly growing collection of integrated cloud services—analytics, machine learning, storage, security, networking, and web—with an industry-leading commitment to the protection and privacy of your data. Our assume breach strategy uses a dedicated "red team" of software security experts who simulate attacks, testing the ability of Azure to detect, protect against emerging threats, and recover from breaches. Our global incident response team works around the clock to mitigate the effects of attacks and malicious activity. The team follows established procedures for incident management, communication, and recovery, and uses discoverable and predictable interfaces with internal and external partners.

Our systems provide continuous intrusion detection and prevention, service attack prevention, regular penetration testing, and forensic tools that help identify and mitigate threats. Multi-factor authentication provides an extra layer of security for end users to access the network. And for the application and the host provider, we offer access control, monitoring, anti-malware, vulnerability scanning, patches, and configuration management.

The Microsoft Azure IoT Suite takes advantage of the security and privacy built into the Azure platform along with our SDL and OSA processes for secure development and operation of all Microsoft software. These procedures provide infrastructure protection, network protection, and identity and management features fundamental to the security of any solution.

The Azure IoT Hub within the IoT Suite offers a fully-managed service that enables reliable and secure bi-directional communication between IoT devices and Azure services such as Azure Machine Learning and Azure Stream Analytics by using per-device security credentials and access control.

To best communicate security and privacy features built into the Azure IoT Suite, we've broken down the suite into the three primary security areas.




**Secure device provisioning and authentication**

The Azure IoT Suite secures devices while they are out in the field by providing a unique identity key for each device, which can be used by the IoT infrastructure to communicate with the device while it is in operation. The

process is quick and easy to set up. The generated key with a user-selected device ID forms the basis of a token used in all communication between the device and the Azure IoT Hub.

Device IDs can be associated with a device during manufacturing (i.e. flashed in a hardware trust module) or can use an existing fixed identity as a proxy (for example CPU serial numbers). Since changing this identifying information in the device is not simple, it is important to introduce logical device IDs in case the underlying device hardware changes but the logical device remains the same. In some cases, the association of a device identity can happen at device deployment time (i.e. an authenticated field engineer physically configures a new device while communicating with the solution backend). The Azure IoT Hub identity registry provides secure storage of device identities and security keys for a solution. Individual or groups of device identities can be added to an allow list, or a block list, enabling complete control over device access.

Azure IoT Hub access control policies in the cloud enable activation and disabling any device identity, providing a way to disassociate a device from an IoT deployment when required. This association and disassociation of devices is based on each device identity.

Additional device security features include the following:

- Devices do not accept unsolicited network connections. They establish all connections and routes in an outbound-only fashion. For a device to receive a command from the backend, the device must initiate a connection to check for any pending commands to process. Once a connection between the device and IoT Hub is securely established, messaging from the cloud to the device and device to the cloud can be sent transparently.
- Devices only connect to or establish routes to well-known services with which they are peered, such as an Azure IoT Hub.
- System-level authorization and authentication use per-device identities, making access credentials and permissions near-instantly revocable.

**Secure connectivity**

Durability of messaging is an important feature of any IoT solution. The need to durably deliver commands and/or receive data from devices is underlined by the fact that IoT devices are connected over the Internet, or other similar networks which can be unreliable. Azure IoT Hub offers durability of messaging between cloud and devices through a system of acknowledgments in response to messages. Additional durability for messaging is achieved by caching messages in the IoT Hub for up to seven days for telemetry and two days for commands.

Efficiency is important to ensure conservation of resources and operation in a resource-constrained environment. HTTPS (HTTP Secure), the industry-standard secure version of the popular http protocol, is supported by Azure IoT Hub, enabling efficient communication. Advanced Message Queuing Protocol (AMQP) and Message Queuing Telemetry Transport (MQTT), supported by Azure IoT Hub, are designed not only for efficiency in terms of resource use but also reliable message delivery.

Scalability requires the ability to securely interoperate with a wide range of devices. Azure IoT hub enables secure connection to both IP-enabled and non-IP-enabled devices. IP-enabled devices are able to directly connect and communicate with the IoT Hub over a secure connection. Non-IP-enabled devices are resource-constrained and connect only over short distance communication protocols, such as Zwave, ZigBee, and Bluetooth. A field gateway is used to aggregate these devices and performs protocol translation to enable secure bi-directional communication with the cloud.

Additional connection security features include the following:

- The communication path between devices and Azure IoT Hub, or between gateways and Azure IoT Hub, is secured using industry-standard Transport Layer Security (TLS) with Azure IoT Hub authenticated using X.509 protocol.
- In order to protect devices from unsolicited inbound connections, Azure IoT Hub does not open any connection to the device. The device initiates all connections.

- Azure IoT Hub durably stores messages for devices and waits for the device to connect. These commands are stored for two days, enabling devices connecting sporadically, due to power or connectivity concerns, to receive these commands. Azure IoT Hub maintains a per-device queue for each device.

**Secure processing and storage in the cloud**

From encrypted communications to processing data in the cloud, the Azure IoT Suite helps keep data secure. It provides flexibility to implement additional encryption and management of security keys. Using Azure Active Directory (AAD) for user authentication and authorization, Azure IoT Suite can provide a policy-based authorization model for data in the cloud, enabling easy access management that can be audited and reviewed. This model also enables near-instant revocation of access to data in the cloud, and of devices connected to the Azure IoT Suite.

Once data is in the cloud, it can be processed and stored in any user-defined workflow. Access to each part of the data is controlled with Azure Active Directory, depending on the storage service used.

All keys used by the IoT infrastructure are stored in the cloud in secure storage, with the ability to roll over in case keys need to be re-provisioned. Data can be stored in Azure Cosmos DB or in SQL databases, enabling definition of the level of security desired. Additionally, Azure provides a way to monitor and audit all access to your data to alert you of any intrusion or unauthorized access.

# Conclusion

The Internet of Things starts with your things—the things that matter most to businesses. IoT can deliver amazing value to a business by reducing costs, increasing revenue, and transforming business. Success of this transformation largely depends on choosing the right IoT software and service provider. That means finding a provider that not only catalyzes this transformation by understanding business needs and requirements, but also provides services and software built with security, privacy, transparency, and compliance as major design considerations. Microsoft has extensive experience with developing and deploying secure software and services and continues to be a leader in this new age of Internet of Things.

The Microsoft Azure IoT Suite builds in security measures by design, enabling secure monitoring of assets to improve efficiencies, drive operational performance to enable innovation, and employ advanced data analytics to transform businesses. With its layered approach towards security, multiple security features, and design patterns, Azure IoT Suite helps deploy an infrastructure which can be trusted to transform any business.

# Additional information

Each Azure IoT Suite pre-configured solution creates instances of Azure services, such as the following:

- **Azure IoT Hub**: Your gateway that connects the cloud to "things". You can scale to millions of connections per hub and process massive volumes of data with per-device authentication support helping you secure your solution.
- **Azure Cosmos DB**: A scalable, fully-indexed database service for semi-structured data that manages metadata for the devices you provision, such as attributes, configuration, and security properties. Cosmos DB offers high-performance and high-throughput processing, schema-agnostic indexing of data, and a rich SQL query interface.
- **Azure Stream Analytics**: Real-time stream processing in the cloud that enables you to rapidly develop and deploy a low-cost analytics solution to uncover real-time insights from devices, sensors, infrastructure, and applications. The data from this fully-managed service can scale to any volume while still achieving high throughput, low latency, and resiliency.
- **Azure App Services**: A cloud platform to build powerful web and mobile apps that connect to data anywhere; in the cloud or on-premises. Build engaging mobile apps for iOS, Android, and Windows. Integrate with your Software as a Service (SaaS) and enterprise applications with out-of-the-box connectivity to dozens of cloud-based services and enterprise applications. Code in your favorite language and IDE—.NET, Node.js, PHP,

Python, or Java—to build web apps and APIs faster than ever.

- **Logic Apps**: The Logic Apps feature of Azure App Service helps integrate your IoT solution to your existing line-of-business systems and automate workflow processes. Logic Apps enables developers to design workflows that start from a trigger and then execute a series of steps—rules and actions that use powerful connectors to integrate with your business processes. Logic Apps offers out-of-the-box connectivity to a vast ecosystem of SaaS, cloud-based, and on-premises applications.
- **Azure blob storage**: Reliable, economical cloud storage for the data that your devices send to the cloud.

## Next steps

To learn more about securing your IoT solution, see:

- IoT Security Best Practices
- IoT Security Architecture
- Secure your IoT deployment

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Frequently asked questions for IoT Suite

You can read about IoT Hub security in Control access to IoT Hub in the IoT Hub developer guide.

# Frequently asked questions for IoT Suite

8/24/2017 • 4 min to read • Edit Online

See also, the connected factory specific FAQ.

**Where can I find the source code for the preconfigured solutions?**

The source code is stored in the following GitHub repositories:

- Remote monitoring preconfigured solution
- Predictive maintenance preconfigured solution
- Connected factory preconfigured solution

**How do I update to the latest version of the remote monitoring preconfigured solution that uses the IoT Hub device management features?**

- If you deploy a preconfigured solution from the https://www.azureiotsuite.com/ site, it always deploys a new instance of the latest version of the solution.
- If you deploy a preconfigured solution using the command line, you can update an existing deployment with new code. See Cloud deployment in the GitHub repository.

**How can I add support for a new device method to the remote monitoring preconfigured solution?**

See the section Add support for a new method to the simulator in the Customize a preconfigured solution article.

**The simulated device is ignoring my desired property changes, why?**

In the remote monitoring preconfigured solution, the simulated device code only uses the **Desired.Config.TemperatureMeanValue** and **Desired.Config.TelemetryInterval** desired properties to update the reported properties. All other desired property change requests are ignored.

**My device does not appear in the list of devices in the solution dashboard, why?**

The list of devices in the solution dashboard uses a query to return the list of devices. Currently, a query cannot return more than 10K devices. Try making the search criteria for your query more restrictive.

**What's the difference between deleting a resource group in the Azure portal and clicking delete on a preconfigured solution in azureiotsuite.com?**

- If you delete the preconfigured solution in azureiotsuite.com, you delete all the resources that were provisioned when you created the preconfigured solution. If you added additional resources to the resource group, these resources are also deleted.
- If you delete the resource group in the Azure portal, you only delete the resources in that resource group. You also need to delete the Azure Active Directory application associated with the preconfigured solution in the Azure classic portal.

**How many IoT Hub instances can I provision in a subscription?**

By default you can provision 10 IoT hubs per subscription. You can create an Azure support ticket to raise this limit. As a result, since every preconfigured solution provisions a new IoT Hub, you can only provision up to 10 preconfigured solutions in a given subscription.

**How many Azure Cosmos DB instances can I provision in a subscription?**

Fifty. You can create an Azure support ticket to raise this limit, but by default, you can only provision 50 Cosmos DB instances per subscription.

**How many Free Bing Maps APIs can I provision in a subscription?**

Two. You can create only two Internal Transactions Level 1 Bing Maps for Enterprise plans in an Azure subscription. The remote monitoring solution is provisioned by default with the Internal Transactions Level 1 plan. As a result, you can only provision up to two remote monitoring solutions in a subscription with no modifications.

**I have a remote monitoring solution deployment with a static map, how do I add an interactive Bing map?**

1. Get your Bing Maps API for Enterprise QueryKey from Azure portal:

   a. Navigate to the Resource Group where your Bing Maps API for Enterprise is in the Azure portal.
   b. Click **All Settings**, then **Key Management**.
   c. You can see two keys: **MasterKey** and **QueryKey**. Copy the value for **QueryKey**.

   > **NOTE**
   > Don't have a Bing Maps API for Enterprise account? Create one in the Azure portal by clicking + New, searching for Bing Maps API for Enterprise and follow prompts to create.

2. Pull down the latest code from the Azure-IoT-Remote-Monitoring.
3. Run a local or cloud deployment following the command-line deployment guidance in the /docs/ folder in the repository.
4. After you've run a local or cloud deployment, look in your root folder for the *.user.config file created during deployment. Open this file in a text editor.
5. Change the following line to include the value you copied from your **QueryKey**:

```
<setting name="MapApiQueryKey" value="" />
```

**Can I create a preconfigured solution if I have Microsoft Azure for DreamSpark?**

Currently, you cannot create a preconfigured solution with a Microsoft Azure for DreamSpark account. However, you can create a free trial account for Azure in just a couple of minutes that enables you create a preconfigured solution.

**Can I create a preconfigured solution if I have Cloud Solution Provider (CSP) subscription?**

Currently, you cannot create a preconfigured solution with a Cloud Solution Provider (CSP) subscription. However, you can create a free trial account for Azure in just a couple of minutes that enables you create a preconfigured solution.

**How do I delete an AAD tenant?**

See Eric Golpe's blog post Walkthrough of Deleting an Azure AD Tenant.

**Next steps**

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Connected factory preconfigured solution overview
- IoT security from the ground up

# Frequently asked questions for IoT Suite connected factory preconfigured solution

8/24/2017 • 6 min to read • Edit Online

See also, the general FAQ for IoT Suite.

**Where can I find the source code for the preconfigured solution?**

The source code is stored in the following GitHub repository:

- Connected factory preconfigured solution

**What is OPC UA?**

OPC Unified Architecture (UA), released in 2008, is a platform-independent, service-oriented interoperability standard. OPC UA is used by various industrial systems and devices such as industry PCs, PLCs, and sensors. OPC UA integrates the functionality of the OPC Classic specifications into one extensible framework with built-in security. It is a standard that is driven by the OPC Foundation. The OPC Foundation is a not-for-profit organization with more than 440 members. The goal of the organization is to use OPC specifications to facilitate multi-vendor, multi-platform, secure and reliable interoperability through:

- Infrastructure
- Specifications
- Technology
- Processes

**Why did Microsoft choose OPC UA for the connected factory preconfigured solution?**

Microsoft chose OPC UA because it is an open, non-proprietary, platform independent, industry-recognized, and proven standard. It is a requirement for Industrie 4.0 (RAMI4.0) reference architecture solutions ensuring interoperability between a broad set of manufacturing processes and equipment. Microsoft sees demand from our customers to build Industrie 4.0 solutions. Support for OPC UA helps lower the barrier for customers to achieve their goals and provides immediate business value to them.

**How do I add a public IP address to the simulation VM?**

You have two options to add the IP address:

- Use the PowerShell script `Simulation/Factory/Add-SimulationPublicIp.ps1` in the repository. Pass in your deployment name as a parameter. For a local deployment, use `<your username>ConnFactoryLocal`. The script prints out the IP address of the VM.
- In the Azure portal, locate the resource group of your deployment. Except for a local deployment, the resource group has the name you specified as solution or deployment name. For a local deployment using the build script, the name of the resource group is `<your username>ConnFactoryLocal`. Now add a new **Public IP address** resource to the resource group.

> **NOTE**
>
> In either case, ensure you install the latest patches by following the instructions on the Ubuntu website. Keep the installation up to date for as long as your VM is accessible through a public IP address.

**How do I remove the public IP address to the simulation VM?**

You have two options to remove the IP address:

- Use the PowerShell script Simulation/Factory/Remove-SimulationPublicIp.ps1 of the repository. Pass in your deployment name as a parameter. For a local deployment, use `<your username>ConnFactoryLocal`. The script prints out the IP address of the VM.

- In the Azure portal, locate the resource group of your deployment. Except for a local deployment, the resource group has the name you specified as solution or deployment name. For a local deployment using the build script, the name of the resource group is `<your username>ConnFactoryLocal`. Now remove the **Public IP address** resource from the resource group.

**How do I sign in to the simulation VM?**

Signing in to the simulation VM is only supported if you have deployed your solution using the PowerShell script `build.ps1` in the repository.

If you deployed the solution from www.azureiotsuite.com, you cannot sign in to the VM. You cannot sign in, because the password is generated randomly and you cannot reset it.

1. Add a public IP address to the VM. See How do I add a public IP address to the simulation VM?
2. Create an SSH session to your VM using the IP address of the VM.
3. The username to use is: `docker`.
4. The password to use depends on the version you used to deploy:
   - For solutions deployed using the build.ps1 script before 1 June 2017, the password is: `Passw0rd`.
   - For solutions deployed using the build.ps1 script after 1 June 2017, you can find the password in the `<name of your deployment>.config.user` file. The password is stored in the **VmAdminPassword** setting. The password is generated randomly at deployment time unless you specify it using the `build.ps1` script parameter `-VmAdminPassword`

**How do I stop and start all docker processes in the simulation VM?**

1. Sign in to the simulation VM. See How do I sign in to the simulation VM?
2. To check which containers are active, run: `docker ps`.
3. To stop all simulation containers, run: `./stopsimulation`.
4. To start all simulation containers:

   - Export a shell variable with the name **IOTHUB_CONNECTIONSTRING**. Use the value of the **IotHubOwnerConnectionString** setting in the `<name of your deployment>.config.user` file. For example:

     ```
     export IOTHUB_CONNECTIONSTRING="HostName={yourdeployment}.azure-
     devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey={your key}"
     ```

   - Run `./startsimulation`.

**How do I update the simulation in the VM?**

If you have made any changes to the simulation, you can use the PowerShell script `build.ps1` in the repository using the `updatedimulation` command. This script builds all the simulation components, stops the simulation in the VM, uploads, installs, and starts them.

**How do I find out the connection string of the IoT hub used by my solution?**

If you deployed your solution with the `build.ps1` script in the repository, the connection string is the value of **IotHubOwnerConnectionString** in the `<name of your deployment>.config.user` file.

You can also find the connection string using the Azure portal. In the IoT Hub resource in the resource group of your deployment, locate the connection string settings.

### Which IoT Hub devices does the Connected factory simulation use?

The simulation self registers the following devices:

- proxy.beijing.corp.contoso
- proxy.capetown.corp.contoso
- proxy.mumbai.corp.contoso
- proxy.munich0.corp.contoso
- proxy.rio.corp.contoso
- proxy.seattle.corp.contoso
- publisher.beijing.corp.contoso
- publisher.capetown.corp.contoso
- publisher.mumbai.corp.contoso
- publisher.munich0.corp.contoso
- publisher.rio.corp.contoso
- publisher.seattle.corp.contoso

Using the DeviceExplorer or iothub-explorer tool, you can check which devices are registered with the IoT hub your solution is using. To use these tools, you need the connection string for the IoT hub in your deployment.

### How can I get log data from the simulation components?

All components in the simulation log information in to log files. These files can be found in the VM in the folder `home/docker/Logs`. To retrieve the logs, you can use the PowerShell script `Simulation/Factory/Get-SimulationLogs.ps1` in the repository.

This script needs to sign in to the VM. You may need to provide credentials for the sign-in. See How do I sign in to the simulation VM? to find the credentials.

The script adds/removes a public IP address to the VM, if it does not yet have one and removes it. The script puts all log files in an archive and downloads the archive to your development workstation.

Alternatively log in to the VM via SSH and inspect the log files at runtime.

### How can I check if the simulation is sending data to the cloud?

With the DeviceExplorer or the iothub-explorer tool, you can inspect the data sent to IoT Hub from certain devices. To use these tools, you need to know the connection string for the IoT hub in your deployment. See How do I find out the connection string of the IoT hub used by my solution?

Inspect the data sent by one of the publisher devices:

- publisher.beijing.corp.contoso
- publisher.capetown.corp.contoso
- publisher.mumbai.corp.contoso
- publisher.munich0.corp.contoso
- publisher.rio.corp.contoso
- publisher.seattle.corp.contoso

If you see no data sent to IoT Hub, then there is an issue with the simulation. As a first analysis step you should analyze the log files of the simulation components. See How can I get log data from the simulation components? Next, try to stop and start the simulation and if there's still no data sent, update the simulation completely. See How do I update the simulation in the VM?

### Next steps

You can also explore some of the other features and capabilities of the IoT Suite preconfigured solutions:

- Predictive maintenance preconfigured solution overview
- Connected factory preconfigured solution overview
- IoT security from the ground up