

# AZ-220: Microsoft Azure IoT Developer

- **Download Latest Student Handbook and AllFiles Content**  
`/home/ll/Azure_clone/Azure_new/AZ-220-Microsoft-Azure-IoT-Developer/../../releases/latest`
- **Are you a MCT?** - Have a look at our [GitHub User Guide for MCTs](#)
- **Need to manually build the lab instructions?** - Instructions are available in the [MicrosoftLearning/Docker-Build](#) repository
- [\*\*Want to know what is installed on the lab VM?\*\*](#)

## What are we doing?

- To support this course, we will need to make frequent updates to the course content to keep it current with the Azure services used in the course. We are publishing the lab instructions and lab files on GitHub to allow for open contributions between the course authors and MCTs to keep the content current with changes in the Azure platform.
- We hope that this brings a sense of collaboration to the labs like we've never had before - when Azure changes and you find it first during a live delivery, go ahead and make an enhancement right in the lab source. Help your fellow MCTs.

## **How should I use these files relative to the released MOC files?**

- The instructor handbook and PowerPoints are still going to be your primary source for teaching the course content.
- These files on GitHub are designed to be used in conjunction with the student handbook, but are in GitHub as a central repository so MCTs and course authors can have a shared source for the latest lab files.
- It will be recommended that for every delivery, trainers check GitHub for any changes that may have been made to support the latest Azure services, and get the latest files for their delivery.

## **What about changes to the student handbook?**

- We will review the student handbook on a quarterly basis and update through the normal MOC release channels as needed.

## How do I contribute?

- Updates to course labs are in development. Some minor changes are being implemented and some more global updates are being developed. It may be advisable for contributors to propose changes by raising issues before creating pull requests.
- Any MCT can submit a pull request (against the Master branch) to the code or content in the GitHub repro, Microsoft and the course author will triage and include content and lab code changes as needed. If you encounter any blocking issues in the labs, please raise issues immediately.
- You can submit bugs, changes, improvement and ideas. Find a new Azure feature before we have? Submit a new demo!

## How do I use these labs for self-study?

These labs are written as companions to the instructor-led AZ-220 course. Although great effort has been made to ensure that the labs include explanations of each step to allow them to be instructional tools, not all requirements for successful completion are documented. For example, in a classroom environment, students have access to a relatively unrestricted Azure subscription; as a result, a complete list of resource providers used in the labs does not exist. This means that lab steps in a locked-down subscription (one with [Azure Policy](#) restrictions, for example) may fail. This also means that one-on-one lab support is not available outside of an instructor-led environment; please do not open issues requesting one-on-one lab support.

Microsoft Learn offers several [learning.paths](#) for IoT that include sandboxed lab environments. These are going to be more suitable for many students looking for IoT self-study opportunities.

# Notes

## Classroom Materials

It is strongly recommended that MCTs and Partners access these materials and in turn, provide them separately to students. Pointing students directly to GitHub to access Lab steps as part of an ongoing class will require them to access yet another UI as part of the course, contributing to a confusing experience for the student. An explanation to the student regarding why they are receiving separate Lab instructions can highlight the nature of an always-changing cloud-based interface and platform. Microsoft Learning support for accessing files on GitHub and support for navigation of the GitHub site is limited to MCTs teaching this course only.

You can find a processed, formatted version of the Markdown files at <https://aka.ms/az220labs>.

## Known Issues in the Current release

**Please** refer to the current open issues to be sure that a new issue doesn't duplicate an existing issue. The following are known issues that are not in the open issues as we are working on them already:

- Most labs require running the setup script in order to create specifically named IoT devices. Work is being done to "clean up" the device resources within and across labs.
- Many of the lab Exercises have not been broken up into defined Tasks, which can make it easy to get lost or miss a step. Work is being done to address this formatting.

**\* There are places within the lab steps where images could improve the clarity of the instructions. Images are currently being added to the labs where we feel they are required. However, we intend to keep image count low due to the fluid nature of the Azure UI.**

title: Online Hosted Instructions permalink: index.html layout: home

---



# Content Directory

Hyperlinks to each of the lab exercises and demos are listed below.

# Labs

```
{% assign labs = site.pages | where_exp:"page", "page.url contains  
'/Instructions/Labs'" %} | Module | Lab | | --- | --- | {% for activity in labs %}|  
{{ activity.lab.module }} | {{ activity.lab.title }} | {% if activity.lab.type %}-  
{{ activity.lab.type }}{% endif %} | {% endfor %}  
  
{% comment %}
```

## Demos

```
{% assign demos = site.pages | where_exp:"page", "page.url contains  
'/Instructions/Demos'" %} | Module | Demo | | --- | --- | {% for activity in  
demos %}| {{ activity.demo.module }} | .{{ activity.demo.title }} | {% endfor  
%}
```

```
{% endcomment %}
```

# Lab Virtual Machine Installed Software List

Software	Link
Windows 10	<a href="https://www.microsoft.com/software-download/windows10">https://www.microsoft.com/software-download/windows10</a>
Windows Subsystem for Linux 2	<a href="https://docs.microsoft.com/en-us/windows/wsl/install-win10">https://docs.microsoft.com/en-us/windows/wsl/install-win10</a>
Visual Studio Code	<a href="https://code.visualstudio.com/">https://code.visualstudio.com/</a>
Visual Studio Code Azure IoT Tools	<a href="https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools">https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools</a>
Azure Functions for Visual Studio Code	<a href="https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions">https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions</a>
Visual Studio Code Azure Resource Manager Tools Extension	<a href="https://marketplace.visualstudio.com/items?itemName=msazurermttools.azurearm-vscode-tools">https://marketplace.visualstudio.com/items?itemName=msazurermttools.azurearm-vscode-tools</a>
Visual Studio Code Azure CLI Tools Extension	<a href="https://marketplace.visualstudio.com/items?itemName=ms-vscode.azurecli">https://marketplace.visualstudio.com/items?itemName=ms-vscode.azurecli</a>
Visual Studio Code PowerShell Extension	<a href="https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell">https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell</a>
Visual Studio Code C# Extension	<a href="https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp">https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp</a>
Azure PowerShell	<a href="https://docs.microsoft.com/powershell/azure/install-az-ps">https://docs.microsoft.com/powershell/azure/install-az-ps</a>
Azure CLI	<a href="https://docs.microsoft.com/cli/azure/install-azure-cli">https://docs.microsoft.com/cli/azure/install-azure-cli</a>
Git for Windows	<a href="https://git-scm.com/download/win">https://git-scm.com/download/win</a>
Python 3.9	<a href="https://www.python.org/downloads/release/python-390/">https://www.python.org/downloads/release/python-390/</a>
Docker Desktop 3.2.2 (or later)	<a href="https://docs.docker.com/docker-for-windows/install/">https://docs.docker.com/docker-for-windows/install/</a>
.NET Core 3 SDK (current/latest)	<a href="https://dotnet.microsoft.com/download">https://dotnet.microsoft.com/download</a>
Power BI Desktop (current/latest)	<a href="https://powerbi.microsoft.com/en-us/desktop/">https://powerbi.microsoft.com/en-us/desktop/</a>
Postman	<a href="https://www.postman.com/downloads/">https://www.postman.com/downloads/</a>
Node.js	<a href="https://nodejs.org/en/download/">https://nodejs.org/en/download/</a>

Software	Link
# Placeholder	
# Placeholder	
# Placeholder	
---	
demo:	
title: 'Demo: Deploying an ARM Template'	
module: 'Module 1: Exploring Azure Resource Manager'	
---	

# Demo: Deploying an ARM Template

## Instructions

1. Quisque dictum convallis metus, vitae vestibulum turpis dapibus non.

1. Suspendisse commodo tempor convallis.

2. Nunc eget quam facilisis, imperdiet felis ut, blandit nibh.

3. Phasellus pulvinar ornare sem, ut imperdiet justo volutpat et.

2. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

3. Vestibulum hendrerit orci urna, non aliquet eros eleifend vitae.

4. Curabitur nibh dui, vestibulum cursus neque commodo, aliquet accumsan risus.

Sed at malesuada orci, eu volutpat ex

5. In ac odio vulputate, faucibus lorem at, sagittis felis.

6. Fusce tincidunt sapien nec dolor congue facilisis lacinia quis urna.

**Note:** Ut feugiat est id ultrices gravida.

7. Phasellus urna lacus, luctus at suscipit vitae, maximus ac nisl.

◦ Morbi in tortor finibus, tempus dolor a, cursus lorem.

◦ Maecenas id risus pharetra, viverra elit quis, lacinia odio.

◦ Etiam rutrum pretium enim.

# 1. Curabitur in pretium urna, nec ullamcorper diam.

lab: title: 'Lab 01: Getting Started with Azure' module: 'Module 1: Introduction to IoT and Azure IoT Services'

---

# Getting Started with Azure

## Lab Scenario

You work for a gourmet cheese company named Contoso. The company's Chief Technology Officer has evaluated the business opportunity for implementing IoT and has concluded that Contoso can realize significant benefits by implementing an IoT solution. Contoso has selected the Microsoft Azure IoT tools based on their evaluations.

As one of the individuals assigned to the project, you need to become familiar with the Azure tools.



## **In This Lab**

In this lab, you will become familiar with the Azure portal and you will setup a Resource Group. The lab includes the following exercises:

- Explore the Azure Portal
- Create an Azure Dashboard and Resource Group

# Lab Instructions

## Exercise 1: Explore the Azure Portal and Dashboard

Before you begin working with the Azure IoT services, it's good to be familiar with how Azure itself works.

Although Azure commonly referred to as a 'cloud', it is actually a web portal that is designed to make Azure resources accessible from a single web site. All of Azure is accessible through the Azure portal.

### Task 1: Examine the Azure portal Home page

1. In your Web browser, to open the Azure portal, navigate to [portal.azure.com](https://portal.azure.com).

When you log into Azure you will arrive at the Azure portal. The Azure portal provides you with a customizable UI that you can use to access your Azure resources.

2. In the upper left corner of the portal window, to open the Azure portal menu, click the hamburger menu icon.

At the top of the portal menu, you should see a section containing four menu options:

- The **Create a resource** button opens a page displaying the services available through the Azure Marketplace, many of which provide free options. Notice that services are grouped by technology, including "Internet of Things", and that a search box is provided.
- The **Home** button opens a customized page that displays links to Azure services, your recently accessed services, and other tools.
- The **Dashboard** button opens a page displaying your default (or most recently used) dashboard. You will be creating a dashboard later in this lab.
- The **All services** button opens a page similar to the **Create a resource** button described above.

The bottom section of the portal menu is a **FAVORITES** section that can be customized to show your favorite, or most commonly used, resources.

Later in this lab, you will learn how to customize this default list of common services to make it a list of your own favorites.

3. On the Azure portal menu, click **Home**.

The Home page provides a customized view of recently used resources and services, as well as other helpful links.

4. On the home page, under **Tools**, click **Azure Monitor**.

Azure Monitor is a tool that can help you to manage your Azure resources. You will be using Azure Monitor later in this course when you have implemented the services that comprise your IoT solution.

5. On the left side navigation menu, to display a map of data center regions, click **Service Health**.

Notice the dropdowns for **Subscription**, **Region**, and **Service**. When you subscribe to a resource in Azure you'll pick a region to deploy it to. Azure is supported by a series of regions placed all around the world.

This map shows the current status of regions associated with your subscription(s). A green circle is used to indicate that services are running normally at that region.

With any cloud vendor (Azure, AWS, Google Cloud, etc.), services will go down from time to time. If you see a blue 'i' next to a region on the Service Health map, it means the region is experiencing a problem with one or more services. Azure mitigates these issues by running multiple copies of your application in different regions (a practice referred to as *Geo-redundancy*). If a region experiences an issue with a particular service, those requests will roll over to another region to fulfill the request. This is one of the big advantages of hosting apps in the Azure cloud. Azure deals with the issues, so you don't have to.

6. In the upper-left corner of your Azure portal, to navigate back to your home page, click **Home**.

You can also use the portal menu to perform some simple navigation. You will have a chance to try out some options for portal navigation shortly.

## **Task 2: Explore the Azure Service options**

1. Open the Azure portal menu, and then click **All services**.

The All services page provides you with a few different viewing options and access to all of the PaaS, IaaS, and SaaS services that Azure offers. The first time that you open the All services page, you will see the Overview page. This view is accessible from the left side menu.

**Definitions:** The term **PaaS** is an acronym for **Platform as a Service**, the term **IaaS** is an acronym for **Infrastructure as a Service**, and the term **SaaS** is an acronym for **Software as a Service**

2. On the **All services** page, on the left side menu under **Categories**, click **All**.

This view displays all of the services organized into groups corresponding to each Category. The Search box at the top can be very helpful.

3. On the left side menu, under **Categories**, click **Internet of Things**.

The list of services is now limited to the services directly related to an IoT solution.

Service/Resource pages on the Azure portal are sometimes referred to as *blades*. When you opened the Service Health page a couple of steps back, you opened a Service Health blade.

The Azure portal uses blades as a kind of navigation pattern, opening new blades to the right as you drill deeper and deeper into a service. This gives you a form of breadcrumb navigation as you navigate horizontally, and Azure provides a File Explorer style path at the top of the blade that is clickable. For example: Home > Monitor > Service Health. But not every page is a blade. You will get used to it pretty quickly.

4. On the **All services** page, hover your mouse pointer over **IoT Hub**.

A dialog box should be displayed. In the top-right corner, notice the "star" shape. When the star shape is filled-in, the service is selected as a favorite. Favorites will appear on the list of your favorite services on the left navigation menu of the portal window. This makes it easier to access the services that you use most often. You can customize your favorites list by selecting the services that you use most.

5. In the top-right corner of the IoT Hub dialog, to add IoT Hub to the list of your favorite services, click the star shaped icon.

The star should now appear filled. If the star is shown as an outline, click the star icon again.

**Tip:** When you add a new item to your list of favorites, it is placed at the bottom of the favorites list on the Azure portal menu. You can rearrange your favorites into the order that you want by using a drag-and-drop operation.

6. Use the same process to add the following services to your favorites: **Device Provisioning Services, Logic Apps, Stream Analytics jobs, and Storage Accounts.**

**Note:** You can remove a service from the list of your favorite services by clicking the star of a selected service.

7. On the left side menu, under **Categories**, click **General**.
8. Ensure that the following services are selected as favorites:
  - **Subscriptions**
  - **Resource groups**

The favorites that you've added are enough to get you started, but you can use the Internet of Things category to add additional favorites to the portal menu if you want.

### **Task 3: Examine the Toolbar menu**

1. Notice the toolbar at the top of the portal that runs the full width of the window.

In addition to the hamburger menu icon on the far left of this toolbar, there are several tool items that you will find helpful.

First, notice that you have a *Search resources* tool that can be used to quickly find a particular resource.

To the right of the search tool are several buttons that provide access to common tools. You can hover the mouse pointer over a button to display the button name.

- The *Cloud Shell* button opens an interactive, authenticated shell right in the portal window that you can use to manage Azure resources. The Azure Cloud Shell supports Bash and PowerShell.
- The *Directory + Subscriptions* button opens a pane that you can use to manage your Azure subscriptions and account directory (the Azure Active Directory authentication mechanism).
- The *Notifications* button that opens a notifications pane. The notifications pane is useful when working with a long running

process. You will be monitoring notifications when you create and configure resources throughout this course.

- There are also buttons for *Settings*, *Help*, and *Feedback*. The *Help* button contains links to help documents and a list of useful keyboard shortcuts.

On the far right is a button for your account information, providing you with access to things like your account password and billing information.

2. On the toolbar, click **Help**, and then click **Help + support**
3. On the **Help + support** blade, notice the four Tiles for *Getting started*, *Documentation*, *Billing FAQs*, and *Support plans*.

The Help + support blade gives you access to lots of great resources. You may want to come back to this later for further exploration.

4. On the **Help + support** blade, click **Billing FAQs**

A new browser tab should open to display Azure billing documentation.

5. Take a moment to scan the contents of the **Prevent unexpected charges with Azure billing and cost management** page.

If *you* are using a paid Azure subscription and you are responsible for billing (you are the Account Administrator), you can set up cost alerts to help manage your billing.

## Exercise 2: Create an Azure Dashboard and Resource Group

On the Azure portal, dashboards are used to present a customized view of your resources. Information is displayed through the use of tiles which can be arranged and sized to help you organize your resources in useful ways. You can create many different dashboards that provide different views and serve different purposes.

Each tile that you place on your dashboard exposes one or more of your resources. In addition to tiles that expose the data of an individual resource, you can create a tile for something called a resource group.

A resource group is a logical group that contains related resources for a project or application. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. Generally, add resources that

share the same lifecycle to the same resource group so you can easily deploy, update, and delete them as a group.

In this exercise, you will:

- create a custom dashboard that you can use during this course
- create a Resource Group and add a Resource Group tile to your dashboard

### **Task 1: Create a Dashboard**

1. If necessary, open your Web browser and navigate to your Azure portal.

You can use the following link to open the Azure portal: [Azure portal](#)

2. On the Azure portal menu, click **Dashboard**.
3. On the **My Dashboard** page, click **+ New dashboard**

You can create a custom dashboard to organize and access your Azure resources for a project. In this case, you will create a custom dashboard for this course.

4. To name your new dashboard, replace **My Dashboard** with **AZ-220**

In the upcoming steps you will be adding a tile to your dashboard manually. Another option would be to use drag-and-drop operations to add tiles from the Tile gallery to the space provided.

5. At the top of the dashboard editor, click **Done customizing**

You should see an empty dashboard at this point.

### **Task 2: Create a Resource Group and add a Resource Group tile to your Dashboard**

1. On the Azure portal menu, click **Resource groups**

This blade displays all of the resource groups that you have created using your Azure subscription(s). If you are just getting started with Azure, you probably don't have any resource groups yet.

2. On the **Resource groups** blade, in the top-left corner area, click **+ Add**

This will open a new blade named Create a resource group.

3. Take a moment to review the contents of the **Create a resource group** blade.

Notice that the resource group is associated with a Subscription and a Region. Consider the following:

- How might associating a subscription with your resource group be helpful?
- How might associating a region with your resource group affect what you include in your resource group?

4. In the **Subscription** dropdown, select the Azure subscription that you are using for this course.
5. In the **Resource group** textbox, enter **rg-az220**

The name of the resource group must be **unique** within your subscription. A green check mark will appear if the name that you enter has not already been used and confirms to resource group naming rules.

**Tip:** The Azure documentation describes all Azure [naming rules and restrictions](#).

6. In the **Region** dropdown, select a region that is near you.

You should check with your instructor as well, [as not all regions offer all services](#).

You need to provide a location for the resource group because the resource group stores metadata about the resources and acts as the default location for where new resources in the resource group will be created. For compliance reasons, you may want to specify where that metadata is stored. In general, it is recommended that you specify a location where most of your resources will reside. Using the same location can simplify the template used to manage your resources.

7. At the bottom of the **Create a resource group** blade, click **Review + create**.

You should see a message informing you that the settings for your resource group have been validated successfully.

8. To create your resource group, click **Create**.
9. On the top menu of the **Resource groups** blade, to see your new resource group, click **Refresh**



You will learn more about managing your resources as you continue through this course.

10. In the list of named resource groups, click the box to the left of the **rg-az220** resource group that you just created.

**Note:** You don't want to open the resource group in a new blade, you just want to select it (check mark on the left).

11. On the right side of the screen, click the ellipsis (...) corresponding to your resource group, and then click **Pin to dashboard**
12. Close your **Resource groups** blade.

**Your dashboard should now contain an empty Resources tile, but don't worry, you will fill it up soon enough.**

lab: title: 'Lab 02: Getting Started with Azure IoT Services' module: 'Module 1: Introduction to IoT and Azure IoT Services'

---

# Getting Started with Azure IoT Services

## Lab Scenario

You are an Azure IoT Developer working for Contoso, a company that crafts and distributes gourmet cheeses.

You have been tasked with exploring Azure and the Azure IoT services that you will use to develop Contoso's IoT solution. You have already become familiar with the Azure portal and created a resource group for your project. Now you need to begin investigating the Azure IoT services.

## **In This Lab**

In this lab, you will create and examine an Azure IoT Hub and an IoT Hub Device Provisioning Service. The lab includes the following exercises:

- Explore Globally Unique Resource Naming Requirements
- Create an IoT Hub using the Azure portal
- Examine features of the IoT Hub service
- Create a Device Provisioning Service and link it to your IoT Hub
- Examine features of the Device Provisioning Service

# Lab Instructions

## Exercise 1: Explore Globally Unique Resource Naming Requirements

In labs 2-20 of this course, you will be creating Azure resources that are used to develop your IoT solution. To ensure consistency across the labs and to help in tidying up resources when you are finished with them, suggested resource names will be provided within the lab instructions. As much as possible, the suggested resource names will follow the naming guidelines recommended here: [Recommended naming and tagging conventions](#). However, many of the resources that you will create during this course expose services that can be consumed across the web, which means that they must have globally unique names. To ensure that these resources satisfy the globally unique requirement, you will be adding a unique identifier to the end of the resource names when needed.

In this exercise, you will create your unique ID and review some examples that help to illustrate how you will use your unique ID during labs 2-20 of this course.

### Task 1: Create Your Unique ID

1. Construct your unique ID by using your lower-case initials and the current date in the following pattern:

```
text YourInitialsYYMMDD
```

The first part of your unique ID will be your initials in lower-case. The second part will be the last two digits of the current year, the current numeric month, and the current numeric day. Here are some examples:

```
text gwb200123 bho200504 cah201216 dm200911
```

Within the lab instructions, you will see {your-id} listed as part of the suggested resource name whenever you need to enter your unique ID. The {your-id} portion of the suggested resource name is a placeholder. You will replace the entire placeholder string (including the {}) with your unique value.

2. Make a note of your unique ID now and then **use the same value through the entire course.**

**Note:** Don't change the date portion of your unique ID each day. Use the same unique ID each day of the course.

## Task 2: Review How and When to Apply Your Unique ID

Many of the resources that you create during the labs in this course will have publicly-addressable (although secured) endpoints and therefore must be globally unique. Examples of resources that require globally unique names include IoT Hubs, Device Provisioning Services, and Azure Storage Accounts.

As noted above, when you create these types of resources, you will be provided with a resource name that follows suggested guidelines and you will be instructed to include your unique ID as part of the resource name. To help clarify when you need to enter your unique ID, the suggested resource name will include a placeholder value for your unique ID. You will be instructed to replace the placeholder value, `{your-id}`, with your unique ID.

1. Review the following resource naming examples:

If your Unique ID is: **cah191216**

Resource Type	Name Template	Example
IoT Hub	iot-az220-training- {your-id}	iot-az220-training- cah191216
Device Provisioning Service	dps-az220-training- {your-id}	dps-az220-training- cah191216
Azure Storage Account (name must be lower-case and no dashes)	az220storage{your- id}	az220storagecah191216

2. Review the following example for applying your unique ID within a Bash script:

In some of the labs later in this course, you will be instructed to apply your unique ID value within a Bash script. The Bash script file, which is provided for you, might include code that is similar to the following:

```
```bash
```

# !/bin/bash

```
YourID="{your-id}" RGName="rg-az220" IoTHubName="iot-az220-  
training-$YourID" ``
```

In the code above, if the value of your unique ID is `cah191216`, then the line containing `YourID="{your-id}"` should be updated to `YourID="cah191216"`.

**Note:** Notice that you do not change the `$YourID` value on the final code line. If it isn't `{your-id}` then don't replace it.

3. Review the following example for applying your unique ID within C# code:

In some of the labs later in this course, you will be instructed to apply your unique ID value within C# source files. The C# source code, which will be provided to you, might include a code section that looks similar to the following:

```
csharp private string _yourId = "{your-id}"; private string  
_rgName = "rg-az220"; private string _ioTHubName = $"iot-  
az220-training-{_yourId}";
```

In the code above, if the value of your unique ID is `cah191216`, then the line containing `private string _yourId = "{your-id}";` should be updated to `private string _yourId = "cah191216";`

**Note:** Notice that you do not change the `_yourId` value on the final code line. Once again, if it isn't `{your-id}` then don't replace it.

4. Notice that not all resource names require you to apply your unique ID.

As you may have already considered, the Resource Group that you created in the previous lab did not include your unique ID value.

Some resources, like the Resource Group, must have a unique name within your subscription, but the name does not need to be globally unique. Therefore, each student taking this course can use the resource group name: **rg-az220**. Of course this is only true if each student uses their own subscription, but that should be the case.

5. Apply an additional 01 or 02 if it turns out that your unique ID isn't so unique.

It could happen that two or more people with the same initials start the course on the same day. You may not know, unless the person is sitting next to you, until you create your IoT Hub in the next exercise. Azure will let you know if the suggested resource name, including your unique ID, isn't globally unique. In that case you will be instructed to update your unique ID by appending an additional ## value. For example, if the value of your unique ID is `cah191216`, your updated unique ID value could become:

```
text cah20121600 cah20121601 cah20121602 ... cah20121699
```

If you do have to update your unique ID, try to use it consistently.

## Exercise 2: Create an IoT Hub using the Azure portal

The Azure IoT Hub is a fully managed service that enables reliable and secure bidirectional communications between IoT devices and Azure. The Azure IoT Hub service provides the following:

- Establish bidirectional communication with billions of IoT devices
- Multiple device-to-cloud and cloud-to-device communication options, including one-way messaging, file transfer, and request-reply methods.
- Built-in declarative message routing to other Azure services.
- A queryable store for device metadata and synchronized state information.
- Secure communications and access control using per-device security keys or X.509 certificates.
- Extensive monitoring for device connectivity and device identity management events.
- SDK device libraries for the most popular languages and platforms.

There are several methods that you can use to create an IoT Hub. For example, you can create an IoT Hub resource using the Azure portal, or you can create an IoT Hub (and other resources) programmatically. During this course you will be investigating various methods that can be used to create and manage Azure resources, including Azure CLI and Bash scripts.

In this exercise, you will use the Azure portal to create and configure your IoT Hub.

### Task 1: Use the Azure portal to create a resource (IoT Hub)



1. Login to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course. You may find it easiest to use an InPrivate / Incognito browser session to avoid accidentally using the wrong account.

2. Notice that the AZ-220 dashboard that you created in the previous lab has been loaded.

You will be adding resources to your dashboard as the course continues.

3. On the Azure portal menu, click **+ Create a resource**.

The **New** blade that opens is a front-end to the Azure Marketplace, which is a collection of all the resources you can create in Azure. The marketplace contains resources from both Microsoft and the community.

4. In the Search textbox, type **IoT Hub** and then press **Enter**.

The **Marketplace** blade will open to display the available services matching your search criteria.

**Note:** Marketplace services provided by private contributors may include a cost that is not covered by a Microsoft Azure Pass or other Microsoft Azure free credit offerings. You will be using Microsoft provided resources during the labs in this course.

5. On the **Marketplace** blade, click the **IoT Hub** search result.

**Note:** A **Create** action is shown at the bottom of the **IoT Hub** search result - that will navigate directly to the IoT Hub creation view. In normal use you may chose to click this - for the purpose of the tutorial, click anywhere in the main body of the **IoT Hub** search result.

6. On the **IoT Hub** blade, click **Usage Information + Support**

Under **Useful Links**, notice the list of resource links.

There is no need to explore these links now, but it's worth noting that they are available. The *Documentation* link, for example, takes you to the root page for IoT Hub resources and documentation. You can use this page to review the most up-to-date Azure IoT Hub documentation and explore additional resources that are outside the scope of this course. You will be

referred to the docs.microsoft.com site throughout this course for additional reading on specific topics.

If you opened one of the links, close it now and use your browser to navigate back to the Azure portal tab.

## Task 2: Create an IoT Hub with required property settings

1. To begin the process of creating your new IoT Hub, click **Create**.

**Tip:** In the future, there are two other ways to get to the *Create* experience of any Azure resource type:

1. If you have the service in your Favorites, you can click the service to navigate to the list of instances, then click the + *Add* button at the top.
2. You can search for the service name in the *Search* box at the top of the portal to get to the list of instances, then click the + *Add* button at the top.

The following steps walk you through the settings required to create your IoT Hub, explaining each of the fields as you fill them in.

2. On the **IoT hub** blade, in the **Subscription** dropdown, ensure that the Azure subscription that you intend to use for this course is selected.

The *Basics* tab that is selected initially contains uninitialized fields that you are required to fill in, but there are settings on other tabs that you will need to be familiar with as well.

3. To the right of **Resource group**, open the dropdown, and then click **rg-az220**

This is the resource group that you created in the previous lab. You will be grouping the resources that you create for this course together in the same resource group. It is best practice to group related resources in this way, and will help you to clean up your resources when you no longer need them.

4. To the right of **Region**, open the drop-down list and select the same region that you selected for your resource group.

**Note:** One of the upcoming labs will use Event Grid. To support this future lab, you need to select a Region that supports Event Grid. For the current list of regions that support Event Grid, see the following link: [Products available by region](#)

As you saw previously, Azure is supported by a series of datacenters that are placed in regions all around the world. When you create something in Azure, you deploy it to one of these datacenter locations.

**Note:** When picking a region to host your resources, keep in mind that picking a region close to your end users will decrease load/response times. In a production environment, if you are on the other side of the world from your end users, you should not be picking the region nearest you.

5. To the right of **IoT hub name**, enter a globally unique name for your IoT Hub as follows:

To provide a globally unique name, enter **iot-az220-training-{your-id}** (remember to replace **{your-id}** with the unique ID you created in Exercise 1).

For example: **iot-az220-training-cah191216**

The name of your IoT Hub must be globally unique because it is a publicly accessible resource that you must be able to access from any of your IP enabled IoT devices.

Consider the following when you specify a unique name for your new IoT Hub:

- The value that you apply to **IoT hub name** must be unique across all of Azure. This is true because the value assigned to the name will be used in the IoT Hub's connection string. Since Azure enables you to connect devices from anywhere in the world to your hub, it makes sense that all Azure IoT hubs must be accessible from the Internet using the connection string and that connection strings must therefore be unique. You will explore connection strings later in this lab.
- The value that you assign to **IoT hub name** cannot be changed once your resource has been created. If you do need to change the name, you'll need to create a new IoT Hub with the desired name, re-register your devices from the original hub and register them with the new one, and delete your old IoT Hub.
- The **IoT hub name** field is a required field.

**Note:** Azure will ensure that the name you enter is unique. If the name that you enter is not unique, Azure will display a message below the name field as a warning. If you see the warning message,

you should update your unique ID. Try appending your unique ID with '00', or '01', or '02', etc. as necessary to achieve a globally unique name.

**Note:** Some resource names do not allow extended characters like the dash (-) or underscore (\_), so stick with numeric digits when updating your unique ID.

6. At the top of the blade, click **Management**.

Take a minute to review the fields and other information presented on this tab.

7. To the right of **Pricing and scale tier**, ensure that **S1: Standard tier** is selected.

Azure IoT Hub provides several tier options depending on how many features you require and how many messages you need to send within your solution per day. The *S1* tier that you are using in this course allows a total of 400,000 messages per unit per day and provides all of the services that are required in this training. You won't actually need 400,000 messages per unit per day, but you will be using features provided by the Standard tier, such as *Cloud-to-device commands*, *Device management*, and *IoT Edge*. IoT Hub also offers a Free tier that is meant for testing and evaluation. It has the same capabilities as the Standard tier, but limited messaging allowances. It is important to note that you cannot upgrade from the Free tier to either Basic or Standard. The Free tier allows 500 devices to be connected to the IoT hub and up to 8,000 messages per day. Each Azure subscription can create one IoT Hub in the Free tier.

**Note:** The *S1 - Standard* tier has a cost of \$25.00 USD per month per unit. You will be specifying 1 unit. For details about the other tier options, see [Choosing the right IoT Hub tier for your solution](#).

8. To the right of **Number of S1 IoT hub units**, ensure that **1** is selected.

As mentioned above, the pricing tier that you choose establishes the number of messages that your hub can process per unit per day. To increase the number of messages that your hub can process without moving to a higher pricing tier, you can increase the number of units. For example, if you want your IoT hub to support ingress of up to 800,000 messages per day, you could specify *two* S1 tier units. For this course you will be using just 1 unit.

9. To the right of **Defender for IoT**, ensure that **Off** is selected.

Azure Defender for IoT is a unified security solution for identifying IoT/OT devices, vulnerabilities, and threats. It enables you to secure your entire IoT/OT environment, whether you need to protect existing IoT/OT devices or build security into new IoT innovations.

**TIP: Azure Defender for IoT** was formerly known as **Azure Security Center** and you may still see places in Azure, and in this content, where the name has not yet been updated.

Azure Defender for IoT is on by default because security is important to your IoT solution. You will be exploring Azure Defender for IoT in Lab 19 of this course. Disabling it for now ensures that the Lab 19 instructions work as expected.

Currently, you can enable Azure Defender at the subscription level, through the Azure portal. Azure Defender is free for the first 30 days. Any usage beyond 30 days will be automatically charged as per the pricing information detailed [here](#).

10. Expand **Advanced Settings**, and then ensure that **Device-to-cloud partitions** is set to **4**.

The number of partitions relates the device-to-cloud messages to the number of simultaneous readers of these messages. Most IoT hubs will only need four partitions, which is the default value. For this course you will create your IoT Hub using the default number of partitions.

11. Within the **Transport Layer Security (TLS)** section, ensure the **Minimum TLS Version** is set to **1.0**.

IoT Hub uses Transport Layer Security (TLS) to secure connections from IoT devices and services. Three versions of the TLS protocol are currently supported, namely versions 1.0, 1.1, and 1.2.

[!Important] The **Minimum TLS Version** property cannot be changed once your IoT Hub resource is created. It is therefore essential that you properly test and validate that all your IoT devices and services are compatible with TLS 1.2 and the recommended ciphers in advance. You can learn more about IoT Hub and TLS below: \* [Transport Layer Security \(TLS\) support in IoT Hub](#)

12. At the top of the blade, click **Review + create**.

Take a minute to review the settings that your provided.

13. At the bottom of the blade, to finalize the creation of your IoT Hub, click **Create**.

Deployment can take a minute or more to complete. You can open the Azure portal Notification pane to monitor progress.

14. Notice that after a couple of minutes you receive a notification stating that your IoT Hub was successfully deployed to your **rg-az220** resource group.
15. On the Azure portal menu, click **Dashboard**, and then click **Refresh**.

You should see that your resource group tile lists your new IoT Hub.

### Exercise 3: Examine the IoT Hub Service

As you have already learned, IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your Azure IoT services and your connected devices.

IoT Hub's capabilities help you build scalable, full-featured IoT solutions such as managing industrial equipment used in manufacturing, tracking valuable assets in healthcare, monitoring office building usage, and many more scenarios. IoT Hub monitoring helps you maintain the health of your solution by tracking events such as device creation, device failures, and device connections.

In this exercise, you will examine some of the features that IoT Hub provides.

#### Task 1: Explore the IoT Hub Overview information

1. If necessary, log in to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. Verify that your AZ-220 dashboard is being displayed.
3. On the **rg-az220** resource group tile, click **iot-az220-training-{your-id}**

When you first open your IoT Hub blade, the **Overview** information will be displayed. As you can see, the area at the top of this blade provides some essential information about your IoT Hub service, such as

datacenter location and subscription. But this blade also includes tiles that provide information about how you are using your hub and recent activities. Let's take a look at these tiles before exploring further.

4. At the bottom-left of your IoT Hub blade, notice the **IoT Hub Usage** tile.

**Note:** The tiles positions are based upon the width of the browser window, so the layout may be a little different than described.

This tile provides a quick overview of what is connected to your hub and message count. As you add devices and start sending messages, this tile will provide nice "at-a-glance" information.

5. To the right of the **IoT Hub Usage** tile, notice the **Number of messages used** tile and the **Device to cloud messages** tile.

The **Device to cloud messages** tile provides a quick view of the incoming messages from your devices over time. You will be registering a device and sending messages to your hub during a module in the next module, so you will begin to see information on these tiles pretty soon.

The **Number of messages used** tile can help you to keep track of the total number of messages used.

## **Task 2: View features of IoT Hub using the left-side menu**

1. On the IoT Hub blade, take a minute to scan the left-side menu options.

As you would expect, these menu options are used to open panes that provide access to properties and features of your IoT Hub. For example, some panes provides access to devices that are connected to your hub.

2. On the left-side menu, under **Explorers**, click **IoT devices**

This pane can be used to add, modify, and delete devices registered to your hub. You will get pretty familiar with this pane by the end of this course.

3. On the left-side menu, near the top, click **Activity log**

As the name implies, this pane gives you access to a log that can be used to review activities and diagnose issues. You can also define queries that help with routine tasks. Very handy.

4. On the left-side menu, under **Settings**, click **Built-in endpoints**

IoT Hub exposes "endpoints" that enable external connections. Essentially, an endpoint is anything connected to or communicating with your IoT Hub. You should see that your hub already has two endpoints defined:

- *Events*
- *Cloud to device messaging*

5. On the left-side menu, under **Messaging**, click **Message routing**

The IoT Hub message routing feature enables you to route incoming device-to-cloud messages to service endpoints such as Azure Storage containers, Event Hubs, and Service Bus queues. You can also create routing rules to perform query-based routes.

6. At the top of the **Message routing** pane, click **Custom endpoints**.

Custom endpoints (such as Service Bus queue and Storage) are often used within an IoT implementation.

7. Take a minute to scan through the menu options under **Settings**

**Note:** This lab exercise is only intended to be an introduction to the IoT Hub service and get you more comfortable with the UI, so don't worry if you feel a bit overwhelmed at this point. You will be configuring and managing your IoT Hub, devices, and their communications as this course continues.

## **Exercise 4: Create a Device Provisioning Service using the Azure portal**

The Azure IoT Hub Device Provisioning Service is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention. The Device Provisioning Service provides the following:

- Zero-touch provisioning to a single IoT solution without hardcoding IoT Hub connection information at the factory (initial setup)
- Load balancing devices across multiple hubs
- Connecting devices to their owner's IoT solution based on sales transaction data (multitenancy)
- Connecting devices to a particular IoT solution depending on use-case (solution isolation)
- Connecting a device to the IoT hub with the lowest latency (geo-sharding)



- Reprovisioning based on a change in the device
- Rolling the keys used by the device to connect to IoT Hub (when not using X.509 certificates to connect)

There are several methods that you can use to create an instance of the IoT Hub Device Provisioning Service. For example, you can use the Azure portal, which is what you will do in this task. But you can also create a DPS instance using Azure CLI or an Azure Resource Manager Template.

### **Task 1: Use the Azure portal to create a resource (Device Provisioning Service)**

1. If necessary, log in to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. On the Azure portal menu, click **+ Create a resource**.

As you saw previously, the **New** blade provides you with the capability to search the Azure Marketplace for services.

3. In the Search textbox, type **Device Provisioning Service** and then press Enter.
4. On the **Marketplace** blade, click **IoT Hub Device Provisioning Service** search result.

**Note:** A **Create** action is shown at the bottom of the **IoT Hub Device Provisioning Service** search result - that will navigate directly to the IoT Hub Device Provisioning Service creation view. In normal use you may choose to click this - for the purpose of the tutorial, click anywhere in the main body of the **IoT Hub Device Provisioning Service** search result.

5. On the **IoT Hub Device Provisioning Service** blade, click **Usage Information + Support**.

Under **Useful links**, notice the list of resource links.

Again, there is no need to explore this documentation now, but it is good to know that it is available. The IoT Hub Device Provisioning Service Documentation page is the root page for DPS. You can use this page to explore current documentation and find tutorials and other resources that

will help you to explore activities that are outside the scope of this course. You will be referred to the docs.microsoft.com site throughout this course for additional reading on specific topics.

If you opened one of the links, close it now and use your browser to navigate back to the Azure portal tab.

## **Task 2: Create a Device Provisioning Service with required property settings**

1. To begin the process of creating your new DPS instance, click **Create**.

Next, you need to specify information about the Hub and your subscription. The following steps walk you through the settings, explaining each of the fields as you fill them in.

2. Under **Name**, enter a globally unique name for your IoT Hub Device Provisioning Service as follows:

To provide a globally unique name, enter **dps-az220-training-{your-id}** (remember to replace **{your-id}** with the unique ID you created in Exercise 1).

For example: **dps-az220-training-cah191216**

3. Under **Subscription**, ensure that the subscription you are using for this course is selected.
4. Under **Resource Group**, open the dropdown, and then click **rg-az220**

You will be grouping the resources that you create for this course together in the same resource group. It's a best practice to group related resources in this way, and will help you to clean up your resources when you no longer need them.

5. Under **Location**, open the drop-down list and select the same region that you selected for your resource group.

**Note:** When picking a datacenter to host your resources, keep in mind that picking a datacenter close to your end users will decrease load/response times. If you are on the other side of the world from your end users, you should not be picking the datacenter nearest you.

6. At the bottom of the blade, click **Create**.

Deployment can take a minute or more to complete. You can open the Azure portal Notification pane to monitor progress.

7. Notice that after a couple of minutes you receive a notification stating that your IoT Hub Device Provisioning Service instance was successfully deployed to your **rg-az220** resource group.
8. On the Azure portal menu, click **Dashboard**, and then click **Refresh**.

You should see that your resource group tile lists your new IoT Hub Device Provisioning Service.

### **Task 3: Link your IoT Hub and Device Provisioning Service.**

1. Notice that the AZ-220 dashboard lists both your IoT Hub and DPS resources.

You should see both your IoT Hub and DPS resources listed - (you may need to hit **Refresh** if the resources were only recently created)

2. On the **rg-az220** resource group tile, click **dps-az220-training-{your-id}**.
3. On the **Device Provisioning Service** blade, under **Settings**, click **Linked IoT hubs**.
4. At the top of the blade, click **+ Add**.

You will use the **Add link to IoT hub** blade to provide the information required to link your Device Provisioning service instance to an IoT hub.

5. On the **Add link to IoT hub** blade, ensure that the **Subscription** dropdown is displaying the subscription that you are using for this course.

The subscription is used to provide a list of the available IoT hubs.

6. Open the IoT hub dropdown, and then click **iot-az220-training-{your-id}**.

This is the IoT Hub that you created in the previous exercise.

7. In the Access Policy dropdown, click **iothubowner**.

The *iothubowner* credentials provide the permissions needed to establish the link with the specified IoT hub.

8. To complete the configuration, click **Save**.

You should now see the selected hub listed on the Linked IoT hubs pane. You might need to click **Refresh** to show Linked IoT hubs.

9. On the Azure portal menu, click **Dashboard**.

## Exercise 5: Examine the Device Provisioning Service

The IoT Hub Device Provisioning Service is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention, enabling customers to provision millions of devices in a secure and scalable manner.

### Task 1: Explore the Device Provisioning Service Overview information

1. If necessary, log in to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. Verify that your AZ-220 dashboard is being displayed.
3. On the **rg-az220** resource group tile, click **dps-az220-training-{your-id}**

When you first open your Device Provisioning Service instance, it will display the Overview information. As you can see, the area at the top of the blade provides some essential information about your DPS instance, such as status, datacenter location and subscription. This blade also provides the *Quick Links* section, which provide access to:

- [Azure IoT Hub Device Provisioning Service Documentation](#)
- [Learn more about IoT Hub Device Provisioning Service](#)
- [Device Provisioning concepts](#)
- [Pricing and scale details](#)

When time permits, you can come back and explore these links.

### Task 2: View features of Device Provisioning Service using the navigation menu

1. Take a minute to scan the left-side menu options.

As you might expect, these options open panes that provide access to activity logs, properties and feature of the DPS instance.

2. On the left-side menu, near the top, click **Activity log**

As the name implies, this pane gives you access to a log that can be used to review activities and diagnose issues. You can also define queries that help with routine tasks. Very handy.

3. On the left-side menu, under **Settings**, click **Quick Start**.

This pane lists the steps to start using the Iot Hub Device Provisioning Service, links to documentation and shortcuts to other blades for configuring DPS.

4. On the left-side menu, under **Settings**, click **Shared access policies**.

This pane provides management of access policies, lists the existing policies and the associated permissions.

5. On the left-side menu, under **Settings**, click **Linked IoT hubs**.

Here you can see the linked IoT Hub from earlier. The Device Provisioning Service can only provision devices to IoT hubs that have been linked to it. Linking an IoT hub to an instance of the Device Provisioning service gives the service read/write permissions to the IoT hub's device registry; with the link, a Device Provisioning service can register a device ID and set the initial configuration in the device twin. Linked IoT hubs may be in any Azure region. You may link hubs in other subscriptions to your provisioning service.

6. On the left-side menu, under **Settings**, click **Certificates**.

Here you can manage the X.509 certificates that can be used to secure your Azure IoT hub using the X.509 Certificate Authentication. You will investigate X.509 certificates in a later lab.

7. On the left-side menu, under **Settings**, click **Manage enrollments**.

Here you can manage the enrollment groups and individual enrollments.

Enrollment groups can be used for a large number of devices that share a desired initial configuration, or for devices all going to the same tenant. An enrollment group is a group of devices that share a specific attestation mechanism. Enrollment groups support both X.509 as well as symmetric. All devices in the X.509 enrollment group present X.509 certificates that have been signed by the same root or intermediate Certificate Authority (CA). Each device in the symmetric key enrollment group present SAS tokens derived from the group symmetric key. The enrollment group

name and certificate name must be alphanumeric, lowercase, and may contain hyphens.

An individual enrollment is an entry for a single device that may register. Individual enrollments may use either X.509 leaf certificates or SAS tokens (from a physical or virtual TPM) as attestation mechanisms. The registration ID in an individual enrollment is alphanumeric, lowercase, and may contain hyphens. Individual enrollments may have the desired IoT hub device ID specified.

8. Take a minute to review some of the other menu options under **Settings**

**> Note: This lab exercise is only intended to be an introduction to the IoT Hub Device Provisioning Service and get you more comfortable with the UI, so don't worry if you feel a bit overwhelmed at this point. You will be covering DPS in much more detail as the course continues.**

lab: title: 'Lab 03: Setup the Development Environment' module: 'Module 2: Devices and Device Communication'

---

# Set up the Development Environment


## Lab Scenario

As one of the developers at Contoso, you know that setting up your development environment is an important step before starting to build your Azure IoT solution. You also know that Microsoft and other companies provide a number of tools that can be used to develop and support your IoT solutions, and that some decisions should be made about which tools your team will use.

You decide to prepare a development environment that the team can use to work on your IoT solution. The environment will need to support your work in Azure and on your local PC. After some discussion, your team has made the following high-level decisions about the dev environment:

- Operating System: Windows 10 will be used as the OS. Windows is used by most of your team, so it was a logical choice. You make a note to the team that Azure services support other operating systems (such as Mac OS and Linux), and that Microsoft provides supporting documentation for the members of your team who choose one of these alternatives.
- General Coding Tools: Visual Studio Code and Azure CLI will be used as the primary coding tools. Both of these tools support extensions for IoT that leverage the Azure IoT SDKs.
- IoT Edge Tools: Docker Desktop Community and Python will be used to support custom IoT Edge module development (along with Visual Studio Code).

In support of these decisions, you will be setting up the following environment:

- Windows 10 64-bit: Pro, Enterprise, or Education (Build 15063 or later). Including
- 4GB  8GB system RAM (higher the better for Docker)
- Hyper-V and Containers features of Windows must be enabled.
- BIOS-level hardware virtualization support must be enabled in the BIOS settings.

**Note:** When setting up the development environment on a virtual machine, the VM environment must support nested virtualization - [nested](#)



## virtualization

- Azure CLI (current/latest)
- .NET Core 3.1.200 (or later) SDK
- VS Code (latest)
- Python 3.9
- Docker Desktop Community 2.1.0.5 (or later) set to Linux Containers
- IoT Extensions for VS Code and Azure CLI
- node.js (latest)

**Note:** A virtual machine has been created for this course that provides a majority of the tools specified above. The instructions below support using the prepared VM or setting up the development environment locally using your PC.

## **In This Lab**

In this lab, you will set up the base developer tools for your development environment, install the Azure IoT extensions for Visual Studio Code and Azure CLI, and then download some files from GitHub that you will use during the labs. The lab includes the following exercises:

- Install Developer Tools and Products
- Install Dev Tool Extensions
- Set Up Course Lab Files and Alternative Tools

# Lab Instructions

## Exercise 1: Install Developer Tools and Products

**Important:** The tools and products associated with this Exercise are pre-installed on the virtual machine created for this course. Before continuing, check with your course Instructor to understand if you will be completing labs using the hosted lab VM environment or setting up the dev environment locally on your PC.

### Task 1: Install .NET Core

.NET Core is a cross-platform version of .NET for building websites, services, and console apps.

1. To open the .NET Core download page, use the following link: [.NET Download](#)
2. On the .NET download page, under .NET Core, click **Download .NET Core SDK**.

The .NET Core SDK is used to build .NET Core apps. You will be using it to build/edit code files during the labs in this course.

3. On the popup menu, click **Run**, and then follow the on-screen instructions to complete the installation.

The installation should take less than a minute to complete. The following components will be installed:

- .NET Core SDK 3.1.100 or later
- .NET Core Runtime 3.1.100 or later
- ASP.NET Core Runtime 3.1.100 or later
- .NET Core Windows Desktop Runtime 3.1.0 or later

The following resources are available for further information:

- [.NET Core Documentation](#)
- [.NET Core dependencies and requirements](#)
- [SDK Documentation](#)
- [Release Notes](#)
- [Tutorials](#)

## Task 2: Install Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and run times (such as .NET and Unity).

1. To open the Visual Studio Code download page, click the following link:  
[Download Visual Studio Code](#)

Instructions for installing Visual Studio Code on Mac OS X and Linux can be found on the Visual Studio Code set up guide [here](#). This page also includes more detailed instructions and tips for the Windows installation.

2. On the Download Visual Studio Code page, click **Windows**.

When you start the download, two things will happen: a popup dialog opens and some getting started guidance will be displayed.

3. On the popup dialog, to begin the set up process, click **Run** and then follow the on-screen instructions.

If you choose to Save the installer to your Downloads folder, you can complete the installation by opening the folder and then double-clicking the VSCodeSetup executable.

By default, Visual Studio Code is installed in the "C:\Program Files (x86)\Microsoft VS Code" folder location (for a 64-bit machine). The set up process should only take about a minute.

**Note:** .NET Framework 4.5 is required for Visual Studio Code when installing on Windows. If you are using Windows 7, please ensure [.NET Framework 4.5](#) is installed.

For detailed instructions on installing Visual Studio Code, see the Microsoft Visual Studio Code Installation Instruction guide here:  
<https://code.visualstudio.com/Docs/editor/setup>

## Task 3: Install Azure CLI

Azure CLI is a command-line tool that is designed to make scripting Azure-related tasks easier. It also enables you to flexibly query data, and it supports long-running operations as non-blocking processes.

1. Open your browser, and then navigate to the Azure CLI tools download page: [Install Azure CLI](#)

You should be installing the latest version of the Azure CLI tools (currently version 2.4). If version 2.4 is not the latest version listed on this "azure-cli-latest" download page, install the more recent version.

2. On the **Install Azure CLI** page, select the install option for your OS (such as **Install on Windows**), and then follow the on-screen instructions to install the Azure CLI tool.

You will be given detailed instructions for using the Azure CLI tools during the labs in this course, but if you want more information now, see [Get started with Azure CLI](#)

#### **Task 4: Install Python 3.9**

You will be using Python 3.9 in support of IoT Edge and Docker.

1. In your Web browser, navigate to <https://www.python.org/downloads/>
2. On the Python download page, select the installer file that is appropriate for your Operating System.
3. When prompted, select the option to run the installer
4. On the Install Python dialog, click **Add Python 3.9 to PATH**.
5. Click **Install Now**.
6. When the "Setup was successful" page appears, click **Disable path length limit**.
7. To finish the installation process, click **Close**.

#### **Task 5: Install Docker Desktop**

You will be using Docker Desktop Community (latest stable version) during a lab that covers creating and deploying custom IoT Edge modules.

1. In your Web browser, navigate to <https://docs.docker.com/docker-for-windows/install/>

The left-side navigation menu provides access to installations for additional operating systems.

2. Verify that your PC meets the System Requirements.

You can use Windows Settings to open the Windows Features dialog, and use that to verify that Hyper-V and Containers are enabled.

3. Click **Download from Docker Hub**

4. Under Docker Desktop for Windows, click **Get Docker Desktop for Windows (stable)**.

5. To start the installation, click **Run**.

It can take a little while for the installation dialog for Docker Desktop to appear.

6. When the Installation Succeeded message appears, click **Close**.

Docker Desktop does not start automatically after installation. To start Docker Desktop, search for Docker, and select Docker Desktop in the search results. When the whale icon in the status bar stays steady, Docker Desktop is up-and-running, and is accessible from any terminal window.

## Task 6 - Install node.js

Some sample web applications are run locally using node.js. The following steps ensure node.js is installed and running the latest version:

1. Using a browser, open the [node.js download page](#)
2. Download the latest LTS (Long Term Support) version - 14.16.0 at the time of writing.
3. When prompted, select the option to run the installer.
4. Step through the installer:
5. **End-User License Agreement** - accept terms and click **Next**.
6. **Destination Folder** - accept the default (or change if required) and click **Next**.
7. **Custom Setup** - accept the defaults and click **Next**.
8. **Tools for Native Modules** - check the **Automatically install** and click **Next**.
9. **Ready to install Node.js** - click **Install**

- On the UAC dialog, click **Yes**.
- 10. Wait for the install to complete and click **Finish**.
- 11. In the **Install Additional Tools for Node.js** command window, when prompted, press **Enter** to continue.
- 12. On the UAC dialog, click **Yes**.

Multiple packages will be downloaded and installed. This will take some time.

- 13. Once the installation has completed, open a **new** command shell and enter the following command:

```
powershell node --version
```

If node is installed successfully, the installed version will be displayed.

## Exercise 2: Install Dev Tool Extensions

The Visual Studio Code and Azure CLI tools both support an Azure IoT extension that helps developers to create their solutions more efficiently. These extensions leverage the Azure IoT SDKs and will often reduce development time while ensuring security provisions. You will also be adding a C# extension for Visual Studio Code.

### Task 1: Install Visual Studio Code Extensions

1. Open Visual Studio Code.
2. On the left side of the Visual Studio Code window, click **Extensions**.

You can hover the mouse pointer over the buttons to display the button titles. The Extensions button is sixth from the top.

3. In the Visual Studio Code Extension manager, search for and then install the following Extensions:
  - [Azure IoT Tools](#) (vsciot-vscode.azure-iot-tools) by Microsoft
  - [C# for Visual Studio Code](#) (ms-vscode.csharp) by Microsoft
  - [Azure Tools for Visual Studio Code](#) (ms-vscode.vscode-node-azure-pack) by Microsoft
  - [DTDLEditor for Visual Studio Code](#) (vsciot-vscode.vscode-dtdl) by Microsoft

4. Close Visual Studio Code.

## Task 2: Install Azure CLI Extension - local environment

1. Open a new command-line / terminal window, to install the Azure IoT CLI extensions locally.

For example, you can use the Windows **Command Prompt** command-line application.

2. At the command prompt, to install the Azure CLI extension for IoT, enter the following command:

```
bash az extension add --name azure-iot
```

3. At the command prompt, to install the Azure CLI extension for Time Series Insights, enter the following command:

```
bash az extension add --name timeseriesinsights
```

## Task 3: Install Azure CLI Extension - cloud environment

Many of the labs will require the use of the Azure CLI IoT extensions via the Azure Cloud Shell. The following steps ensure the extension is installed and running the latest version.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.
2. If you are prompted about setting up storage for Cloud Shell, accept the defaults.
3. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

4. At the command prompt, to install the Azure CLI extension for IoT, enter the following command:

```
bash az extension add --name azure-iot
```

**Note:** If the extension is already installed, you can ensure you are running the latest version by entering the following command:



```
bash az extension update --name azure-iot
```

5. At the command prompt, to install the Azure CLI extension for Time Series Insights, enter the following command:

```
bash az extension add --name timeseriesinsights
```

#### Task 4: Verify Development Environment Setup

You should verify that the development environment has been set up successfully. Once this is complete, you will be ready to start building your IoT solutions.

1. Open a new command-line / terminal window.
2. Validate the **Azure CLI** installation by running the following command that will output the version information for the currently installed version of the Azure CLI.

```
cmd/sh az --version
```

The `az --version` command will output the version information for Azure CLI that you have installed (the `azure-cli` version number). This command also outputs the version number for all the Azure CLI modules installed, including the IoT extension. You should see output similar to the following:

```
```cmd/sh azure-cli 2.20.0  
  
core 2.20.0 telemetry 1.0.6  
  
Extensions: azure-iot 0.10.9 ```
```

3. Validate the **.NET Core 3.x SDK** installation by running the following command that will output the version number for the currently installed version of the .NET Core SDK.

```
cmd/sh dotnet --version
```

The `dotnet --version` command will output the version of the .NET Core SDK that is currently installed.

4. Verify that .NET Core 3.1 or higher is installed.

Your development environment should now be set up!

## Exercise 3: Set Up Course Lab Files and Alternative Tools

A number of the labs in this course rely on pre-built resources, such as a code project that can be used as a starting point for the lab activity. These lab resources are provided in a GitHub project that you should download to your dev environment.

In addition to the resources that directly support the course labs (the resources contained in the GitHub project), there are some optional tools that you may choose to install because they support learning opportunities outside of this course. One example is PowerShell, which you may see referenced within Microsoft tutorials and other resources.

The instructions below lead you through the configuration of both these resource types.

### Task 1: Download Course Lab Files

Microsoft has created a GitHub repo to provide access to lab resource files. Having these files local to the dev environment is required in some cases and convenient in many others. In this task you will be downloading and extracting the contents of the repo within your development environment.

1. In your Web browser, navigate to the following location:  
<https://github.com/MicrosoftLearning/AZ-220-Microsoft-Azure-IoT-Developer>
2. On the right side of the page, click **Clone or download**, and then click **Download ZIP**.
3. To save the ZIP file to your dev environment, click **Save**.
4. Once the file has been saved, click **Open folder**.
5. Right-click the saved ZIP file, and then click **Extract all**
6. Click **Browse**, and then navigate to folder location that is convenient to access.

**Important:** By default, Windows has a [maximum file path length of 260](#). As the file paths within the ZIP are already long, avoid extracting the archive within nested folders with a large file path. For example, the default path prompted to extract the zip could be similar to **c:\users\[username]\downloads\AZ-220-Microsoft-**

**Azure-IoT-Developer-master** - it is recommended that you shorten this as much as possible, to perhaps **c:\az220**

7. To extract the files, click **Extract**.

Be sure to make note of where you located the files.

## **Task 2: Install Azure PowerShell Module**

**Note:** The lab activities in this course do NOT include using PowerShell, however, you may see sample code in reference documents that use PowerShell. If you want to run PowerShell code, you can use the following instructions to complete the installation steps.

Azure PowerShell is a set of cmdlets for managing Azure resources directly from the PowerShell command line. Azure PowerShell is designed to make it easy to learn and get started with, but provides powerful features for automation. Written in .NET Standard, Azure PowerShell works with PowerShell 5.1 on Windows, and PowerShell 6.x and higher on all platforms.

**Warning:** You can't have both the AzureRM and Az modules installed for PowerShell 5.1 for Windows at the same time. If you need to keep AzureRM available on your system, install the Az module for PowerShell Core 6.x or later. To do this, install PowerShell Core 6.x or later and then follow these instructions in a PowerShell Core terminal.

1. Decide if you wish to install the Azure PowerShell module for just the current user (recommended approach) or for all users.
2. Launch the PowerShell terminal of your choice - if you are installing for all users you must launch an elevated PowerShell session either by either selecting **run as administrator** or with the **sudo** command on macOS or Linux.
3. To only install for the current user, enter the following command:

```
powershell Install-Module -Name Az -AllowClobber -Scope  
CurrentUser
```

or to install for all users on a system, enter the following command:

```
powershell Install-Module -Name Az -AllowClobber -Scope  
AllUsers
```

4. By default, the PowerShell gallery isn't configured as a trusted repository for PowerShellGet. The first time you use the PSGallery you see the

following prompt:

```
```output Untrusted repository
```

You are installing the modules from an untrusted repository. If you trust this repository, change its InstallationPolicy value by running the Set-PSRepository cmdlet.

Are you sure you want to install the modules from 'PSGallery'? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): ```

5. Answer **Yes** or **Yes to All** to continue with the installation.

The Az module is a rollup module for the Azure PowerShell cmdlets. Installing it downloads all of the available Azure Resource Manager modules, and makes their cmdlets available for use.

**Note:** If the **Az** module is already installed, you can update to the latest version using:

```
powershell Update-Module -Name Az
```

## Exercise 3 - Register resource providers

Many different types of resources will be created during this course, some of which may not have been register for use in the current subscription. While some resources are registered automatically during the first use, others must be registered before they can be used, otherwise errors will be reported.

### Task 1 - Register resource providers using the Azure CLI

The Azure CLI provides a number of commands to help manage resource providers. In this task, you will ensure that the resource providers required for this course are registered.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.
2. To view a list of the current state of the resource providers, enter the following command:

```
powershell az provider list -o table
```

This will display a *long* list of resources, similar to:

```
```powershell Namespace RegistrationPolicy RegistrationState
```

---

```
Microsoft.OperationalInsights RegistrationRequired Registered  
microsoft.insights RegistrationRequired NotRegistered  
Microsoft.DataLakeStore RegistrationRequired Registered  
Microsoft.DataLakeAnalytics RegistrationRequired Registered  
Microsoft.Web RegistrationRequired Registered  
Microsoft.ContainerRegistry RegistrationRequired Registered  
Microsoft.ResourceHealth RegistrationRequired Registered  
Microsoft.BotService RegistrationRequired Registered Microsoft.Search  
RegistrationRequired Registered Microsoft.EventGrid  
RegistrationRequired Registered Microsoft.SignalRService  
RegistrationRequired Registered Microsoft.VSOnline  
RegistrationRequired Registered Microsoft.Sql RegistrationRequired  
Registered Microsoft.ContainerService RegistrationRequired Registered  
Microsoft.ManagedIdentity RegistrationRequired Registered ... ```
```

3. To see return a list of the namespaces that contains the string **Event**, run the following command:

```
powershell az provider list -o table --query "[?  
contains(namespace, 'Event')]"
```

The results will be similar to:

```
```powershell Namespace RegistrationState RegistrationPolicy
```

---

```
Microsoft.EventGrid NotRegistered RegistrationRequired  
Microsoft.EventHub Registered RegistrationRequired ```
```

4. To register the resources required for this course, execute the following commands:

```
powershell az provider register --namespace  
"Microsoft.EventGrid" --accept-terms az provider register --  
namespace "Microsoft.EventHub" --accept-terms az provider  
register --namespace "Microsoft.Insights" --accept-terms az  
provider register --namespace "Microsoft.TimeSeriesInsights"  
--accept-terms
```

**NOTE:** You may see a warning that **-accept-terms** is in preview - you can ignore this.

**NOTE:** The **microsoft.insights** is listed in lowercase - however the register/unregister commands are case-insensitive.

5. To view the updated status of the resources, execute the following commands:

```
powershell az provider list -o table --query "[?
(contains(namespace, 'insight') || contains(namespace,
'Event') || contains(namespace, 'TimeSeriesInsights'))]"
```

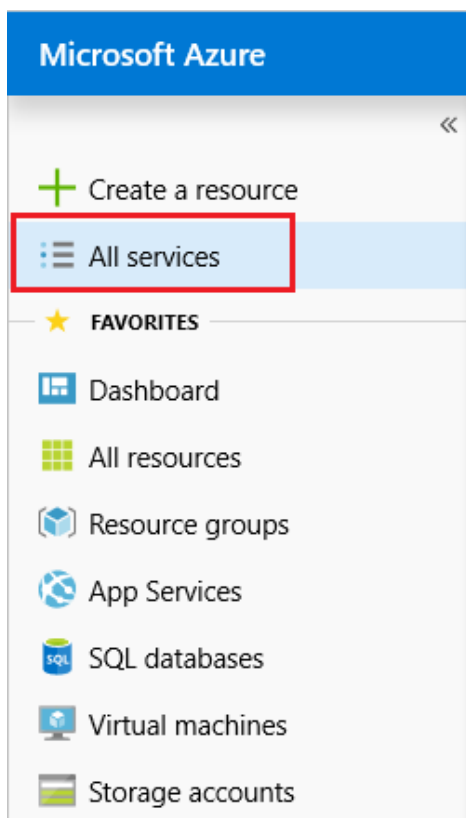
**NOTE:** Although the register/unregister commands are case-insensitive, the query language is not, so **insight** must be lowercase.

The resources should now be registered.

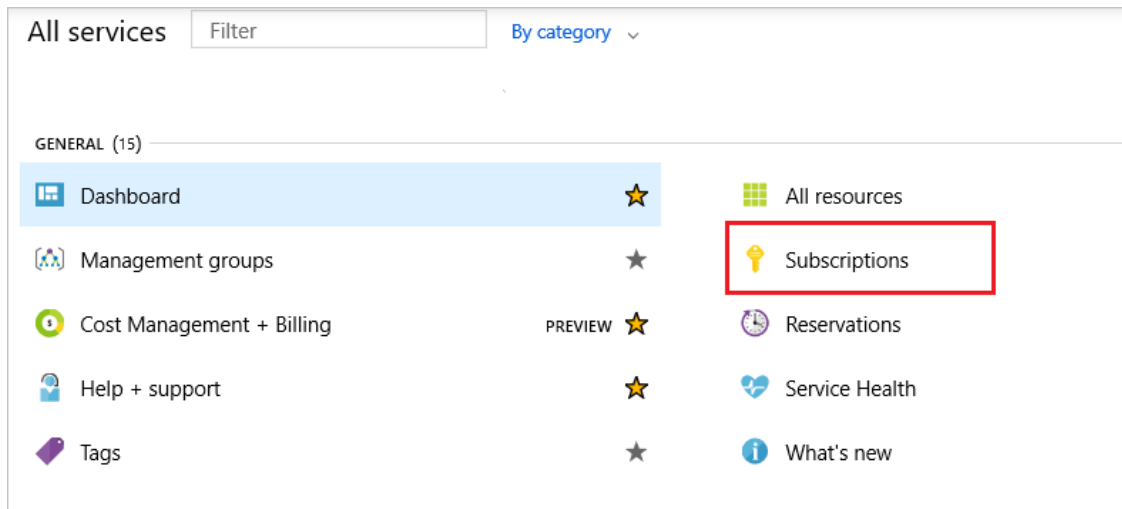
## Task 2 - Register resource providers using the Azure Portal

You can see the registration status and register a resource provider namespace through the portal. In this task, you will familiarize yourself with the UI.

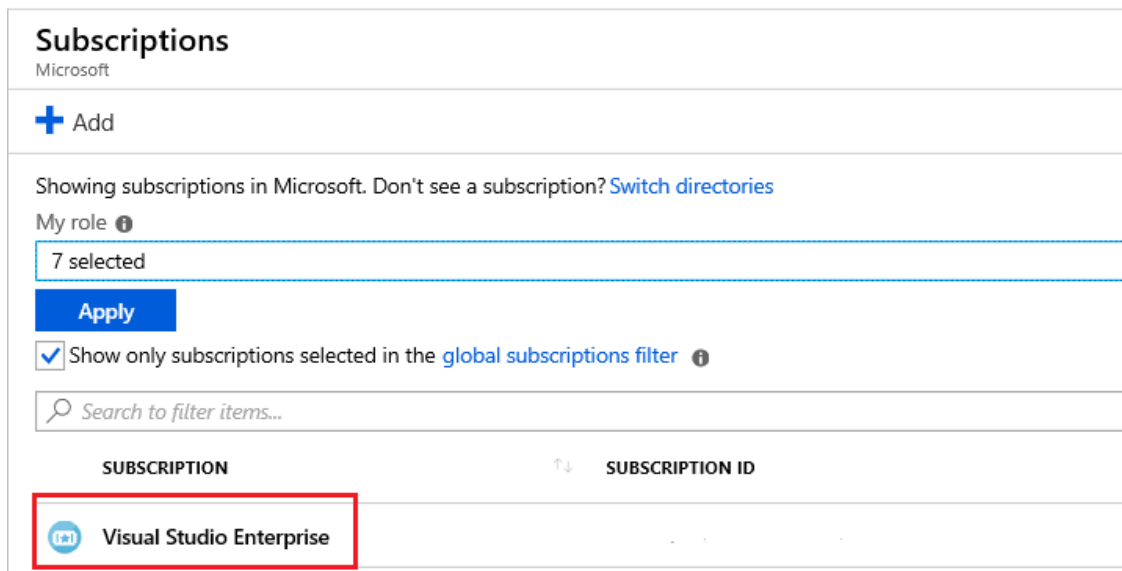
1. If necessary, log in to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.
2. From the portal, select **All services**.



3. Select Subscriptions.



4. From the list of subscriptions, select the subscription you want to use for registering the resource provider.



5. For your subscription, select **Resource providers**.

**Visual Studio Enterprise**  
Subscription

Search (Ctrl+/)

Manage Transfer Cancel subscription

Subscription ID

Directory  
Microsoft (microsoft.onmicrosoft.com)

My role  
Account admin

Offer  
MSDN

Offer ID

For more cost management and optimization capabilities

Costs

Costs by resource

No active resource emitted usage yet. Click

**Billing**

- Invoices
- External services
- Payment methods
- Partner information

**Settings**

- Programmatic deployment
- Resource groups
- Resources
- Usage + quotas
- Policies
- Management certificates
- My permissions
- Resource providers**
- Deployments

6. Look at the list of resource providers, resources can be registered or unregistered by clicking the appropriate action.

Refresh		
Search to filter resource providers...		
PROVIDER	STATUS	
Microsoft.ClassicStorage	✓ Registered	Unregister
Microsoft.Operationallnsights	✓ Registered	Unregister
Microsoft.Storage	✓ Registered	Unregister
84codes.CloudAMQP	✗ NotRegistered	Register
AppDynamics.APM	✗ NotRegistered	Register



7. To filter the listed resources, in the search textbox, enter **insights**.

**Notice that the list is filtered as search criteria is entered. The search is also case-insensitive.**

lab: title: 'Lab 04: Connect an IoT Device to Azure' module: 'Module 2: Devices and Device Communication'

---

# Connect an IoT Device to Azure

## Lab Scenario

Contoso is known for producing high quality cheeses. Due to the company's rapid growth in both popularity and sales, they want to take steps to ensure that their cheeses stay at the same high level of quality that their customers expect.

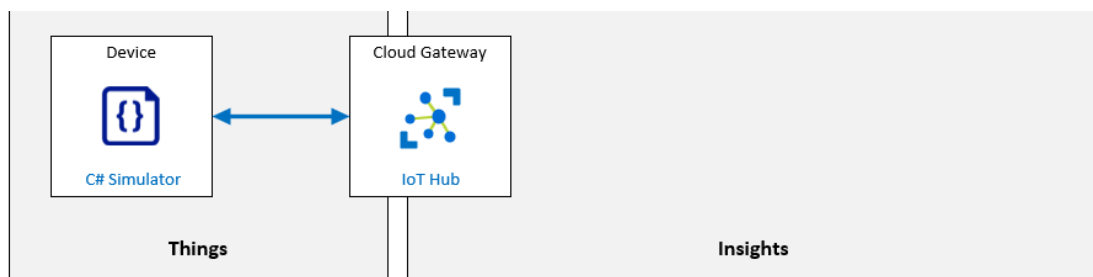
In the past, temperature and humidity data was collected by factory floor workers during each work shift. The company is concerned that the factory expansions will require increased monitoring as the new facilities come online and that a manual process for collecting data won't scale.

Contoso has decided to launch an automated system that uses IoT devices to monitor temperature and humidity. The rate at which telemetry data is communicated will be adjustable to help ensure that their manufacturing process is under control as batches of cheese proceed through environmentally sensitive processes.

To evaluate this asset monitoring solution prior to full scale implementation, you will be connecting an IoT device (that includes temperature and humidity sensors) to IoT Hub.

**Note:** For the purposes of this lab, you will be creating a .NET Core console application that simulates the physical IoT device and sensors. Your simulated device will implement the IoT Device SDK and it will connect to IoT Hub just like a physical device would. Your simulated device will also communicate telemetry values using the same SDK resources used by a physical device, but the sensor readings will be generated values rather than real values read from temperature and humidity sensors.

The following resources will be created:



## In This Lab

In this lab, you will begin by reviewing the lab prerequisites and you will run a script if needed to ensure that your Azure subscription includes the required resources. You will then use the Azure portal to register a device ID with Azure IoT Hub and develop the corresponding simulated device app in Visual Studio Code. You will then insert the connection string (created by IoT Hub when you registered the device) into your simulated device code and run the app to test the connection and verify that telemetry is reaching IoT Hub as intended. The lab includes the following exercises:

- Verify Lab Prerequisites
- Create an Azure IoT Hub Device ID using the Azure portal
- Create and Test a Simulated Device (C#)

# Lab Instructions

## Exercise 1: Verify Lab Prerequisites

This lab assumes that the following Azure resources are available:

Resource Type	Resource Name
Resource Group	rg-az220
IoT Hub	iot-az220-training-{your-id}

If these resources are not available, you will need to run the **lab04-setup.azcli** script as instructed below before moving on to Exercise 2. The script file is included in the GitHub repository that you cloned locally as part of the dev environment configuration (lab 3).

**Note:** The **lab04-setup.azcli** script is written to run in a **bash** shell environment - the easiest way to execute this is in the Azure Cloud Shell.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.
2. If you are prompted about setting up storage for Cloud Shell, accept the defaults.
3. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

4. On the Cloud Shell toolbar, click **Upload/Download files** (fourth button from the right).
5. In the dropdown, click **Upload**.
6. In the file selection dialog, navigate to the folder location of the GitHub lab files that you downloaded when you configured your development environment.

In Lab 3 of this course, "Setup the Development Environment", you cloned the GitHub repository containing lab resources by downloading a ZIP file and extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 04-Connect an IoT Device to Azure
    - Setup

The lab04-setup.azcli script file is located in the Setup folder for lab 4.

7. Select the **lab04-setup.azcli** file, and then click **Open**.

A notification will appear when the file upload has completed.

8. To verify that the correct file has uploaded, enter the following command:

```
bash ls
```

The `ls` command lists the content of the current directory. You should see the lab04-setup.azcli file listed.

9. To create a directory for this lab that contains the setup script and then move into that directory, enter the following Bash commands:

```
bash mkdir lab4 mv lab04-setup.azcli lab4 cd lab4
```

These commands will create a directory for this lab, move the **lab04-setup.azcli** file into that directory, and then change directory to make the new directory the current working directory.

10. To ensure the **lab04-setup.azcli** has the execute permission, enter the following command:

```
bash chmod +x lab04-setup.azcli
```

11. On the Cloud Shell toolbar, to enable access to the lab04-setup.azcli file, click **Open Editor** (second button from the right - { }).
12. In the **Files** list, to expand the lab4 folder and open the script file, click **lab4**, and then click **lab04-setup.azcli**.

The editor will now show the contents of the **lab04-setup.azcli** file.

13. In the editor, update the values of the `{your-id}` and `{your-location}` variables.

Referencing the sample below as an example, you need to set `{your-id}` to the Unique ID you created at the start of this course - i.e. **cah191211**, and set `{your-location}` to the location that you used for your resource group (see the explanation and examples below).

```
```bash
```

# !/bin/bash

## Change these values!

```
YourID="{your-id}" Location="{your-location}" ``
```

**Note:** The `{your-location}` variable should be set to the short name for the region where you are deploying all of your resources. You can see a list of the available locations and their short-names (the **Name** column) by entering this command:

```
``bash az account list-locations -o Table
```

```
DisplayName Latitude Longitude Name
```

---

```
East Asia 22.267 114.188 eastasia Southeast Asia 1.283 103.833  
southeastasia Central US 41.5908 -93.6208 centralus East US 37.3719  
-79.8164 eastus East US 2 36.6681 -78.3889 eastus2 ``
```

14. In the top-right of the editor window, to save the changes made to the file and close the editor, click **...**, and then click **Close Editor**.

If prompted to save, click **Save** and the editor will close.

**Note:** You can use **CTRL+S** to save at any time and **CTRL+Q** to close the editor.

15. To create the resources required for this lab, enter the following command:

```
bash ./lab04-setup.azcli
```

This will take a few minutes to run. You will see output as each step completes.

Once the script has completed, you will be ready to continue with the lab.

## Exercise 2: Create an Azure IoT Hub Device ID using the Azure portal



During this course you will be using IoT Hub's capabilities to help you build a scalable, full-featured IoT solution for Contoso, but in this lab you are focused on using IoT Hub to establish reliable and secure bidirectional communications between IoT Hub and your IoT device(s).

In this exercise, you will open your IoT Hub in the Azure portal, add a new IoT device to the device registry, and then get a copy of the Connection String that IoT Hub created for your device (which you will use in your device code later in the lab).

### Task 1: Create the Device

1. If necessary, log in to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. Verify that your AZ-220 dashboard is being displayed.
3. On the **rg-az220** resource group tile, click **iot-az220-training-{your-id}**
4. On the left-side menu of your IoT Hub blade, under **Explorers**, click **IoT devices**.
5. At the top of the **IoT devices** pane, click **+ New**.
6. In the **Device ID** field, enter **sensor-th-0001**

The device identity (Device ID) is used for device authentication and access control.

It is helpful to establish some form of naming convention for your device identities. There are several reasons for this, including that the device ID is the value IoT Hub uses represent a device. Having a device ID that succinctly and informatively differentiates one device from another is therefor helpful.

The suggested naming convention above, *sensor-th-0001*, identifies this device as a sensor enabled device (*sensor*) that reports temperature and humidity values (*-th*) and is the first device of this type in a series of up to 9999 (*-0001*). Contoso may have 200 or 5000 of these devices installed and reporting environmental conditions from the factory floor, and the device identity will be one of the ways that a device can be recognized.

7. Under **Authentication type**, ensure that **Symmetric key** is selected.

Notice that there are three types of authentication available. In this lab you will leverage the simplest of the three, Symmetric key. X.509 Certificates and their use for authentication will be covered in later labs.

8. Notice that the **Primary key** and **Secondary key** fields are disabled.
9. Under **Auto-generate keys**, ensure the checkbox is selected.

With **Auto-generate keys** selected, the **Primary key** and **Secondary key** fields are disabled and will be populated once the record is saved. Un-selecting **Auto-generate keys** will enable those fields, allowing for values to be entered directly.

10. Under **Connect this device to an IoT hub**, ensure that **Enable** is selected.

You could choose the Disable option here during the initial creation of a device if you were creating the device entry ahead of rollout. You could also choose to set this value to Disable at some future time if you wished to retain the device record, but prevent the associated device from connecting to the IoT Hub.

11. Under **Parent device**, leave **No parent device** as the value.

IoT devices may be parented by other devices such as IoT Edge devices. You will get a chance to implement a Parent-Child device relationship later in the course.

12. To add this device record to the IoT Hub, click **Save**.

After a few moments, the **IoT devices** pane will refresh and the new device will be listed.

**TIP:** You may need to refresh manually - click the **Refresh** button on the page, rather than refreshing the browser

## Task 2: Get the Device Connection String

In order for a device to connect to an IoT Hub, it needs to establish a connection. In this lab, you will use a connection string to connect your device directly to the IoT Hub (this for of authentication is often referred to as symmetric key authentication). When using Symmetric key authentication, there are two connection strings available - one that utilizes the Primary key, the other that uses the Secondary key. As noted above, the Primary and

Secondary keys are only generated once the device record is saved. Therefore, to obtain one of the connection strings, you must first save the record (as you did in the task above) and then re-open the device record (which is what you are about to do).

1. On the **IoT devices** pane of your IoT Hub, under **DEVICE ID**, click **sensor-th-0001**.
2. Take a minute to review the contents of the **sensor-th-0001** device detail blade.

In addition to the device properties, notice that the device detail blade provides access to a number of device related functions (such as Direct Method and Device Twin) along the top of the blade.

3. Notice that the key and connection string values are now populated.

The values are obfuscated by default, but you can click the "eye" icon on the right of each field to toggle between showing and hiding the values.

4. To the right of the **Primary Connection String** field, click **Copy**.

You can hover your mouse pointer over the button icons to display their names; the Copy button is on the far right.

**Note:** You will need to use the Primary Connection String value later in the lab, so you may want to save it to an accessible location (perhaps by pasting the value into a text editor such as NotePad).

The connection string will be in the following format:

```
text HostName={IoTHubName}.azure-devices.net;DeviceId=sensor-  
th-0001;SharedAccessKey={SharedAccessKey}
```

### Exercise 3: Create and Test a Simulated Device (C#)

The Azure IoT Device SDKs enable you to build apps that run on your IoT devices using the device client. Tools in the SDK will help you to establish secure connections as well as packaging messages and implementing communication with your IoT hub. The device SDK will also help you to receive messages, job, method, or device twin updates from your IoT hub.

In this exercise, you will create a simulated device application using Visual Studio Code and the Azure IoT Device SDKs. You will connect your device to Azure IoT Hub using the Device ID and Shared Access Key (Primary Connection String) that you created in the previous exercise. You will then test

your secured device connection and communication to ensure that IoT Hub is receiving the simulated temperature and humidity values from your device as expected.

**Note:** You will be writing your simulated device code using the C# programming language, but don't worry if you are more accustomed to another programming language or if your programming skills are a bit rusty, the instructions will be easy to follow. The important thing is for you to recognize how the IoT Device SDK is implemented in code (which is also explained in detail).

### Task 1: Create the initial project

1. Open a new command-line / terminal window.

For example, you can use the Windows **Command Prompt** command-line application.

2. Navigate to the folder location where you want to create your simulated device application.

The root folder location is not critical, but something easy to find with a short folder path may be helpful.

3. At the command prompt, to create a directory named "CaveDevice" and change the current directory to that directory, enter the following commands:

```
bash mkdir CaveDevice cd CaveDevice
```

1. To create a new .NET console application, enter the following command:

```
bash dotnet new console
```

This command creates a **Program.cs** file in your folder, along with a project file.

2. To install the Azure IoT Device SDK and code libraries required for your simulated device app, enter the following commands:

```
bash dotnet add package Microsoft.Azure.Devices.Client
```

**Note:** The **Microsoft.Azure.Devices.Client** package contains the Azure IoT Device SDK for .NET and has the **Newtonsoft.Json** package as a dependency. The **Newtonsoft.Json** package contains APIs that aid in the creation and manipulation of JSON.

You will build and test your simulated device app in the next task.

3. To ensure all of the application dependencies are downloaded, enter the following command

```
bash dotnet restore
```

4. Open **Visual Studio Code**.
5. On the **File** menu, click **Open Folder**.
6. In the **Open Folder** dialog, navigate to the location where you created the **CaveDevice** directory.
7. In the list of folders, click **CaveDevice**, and then click **Select Folder**.

The EXPLORER pane of Visual Studio Code should now list two C# project files:

- CaveDevice.csproj
- Program.cs

**Note:** If you see a message **Required assets to build and debug are missing from CaveDevice. Add them?**, you may click **Yes** to proceed.

## Task 2: Explore the application

As noted above, the application currently consists of two files:

- CaveDevice.csproj
- Program.cs

In this task, you will use Visual Studio Code to review the contents and purpose of the two application files.

1. In the **EXPLORER** pane, to open the application project file, click **CaveDevice.csproj**.

The **CaveDevice.csproj** file should now be opened in the code editor pane.

2. Take a minute to review the contents of the **CaveDevice.csproj** file.

Your file contents should be similar to the following:

```

<<<xml

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <PackageReference
    Include="Microsoft.Azure.Devices.Client" Version="1.*" />
</ItemGroup>

<<<

```

**Note:** The package version numbers in your file may differ from those shown above, that's okay.

The project file (.csproj) is an XML document that specifies the type of project that you are working on. In this case, the project is an **Sdk** style project.

As you can see, the project definition contains two sections - a **PropertyGroup** and an **ItemGroup**.

The **PropertyGroup** defines the type of output that building this project will produce. In this case you will be building an executable file that targets .NET Core 3.1.

The **ItemGroup** specifies any external libraries that are required for the application. These particular references are for NuGet packages, and each package reference specifies the package name and the version. The `dotnet add package` commands (that you entered in the steps above) added these references to the project file and the `dotnet restore` command ensured that all of the dependencies were downloaded.

**Tip:** You can learn more about NuGet [here](#).

3. In the **EXPLORER** pane, click **Program.cs**.

The **Program.cs** file should now be opened in the code editor pane.

4. Take a minute to review the contents of the **Program.cs** file.

Your file contents should be similar to the following:

```

<<<csharp using System;

```

```
namespace CaveSensor { class Program { static void Main(string[] args)
{ Console.WriteLine("Hello World!"); } } } ``
```

This program simply writes "Hello World!" to the command line window. Even though there isn't much code here, there are still some things worth noting:

- The `using` area - the source file lists the namespaces that the code is **using** (this is typically done at the top of the file as it is here). In this example, the code specifies that it is using `System`. This means that when your code uses a component that's contained within the **System** namespace, you don't have to explicitly list the word **System** within that code line. For example, in the code above, the `Console` class is used to write "Hello World!". The `Console` class is part of the **System** namespace, but you didn't have to include the word `System` when you used `Console`. The benefit of this becomes more apparent when you consider that some namespaces are nested quite deeply (five or more levels is common). Once again referring to the code above, if you didn't specify `using System;`, you would have to write the console line as:

```
csharp System.Console.WriteLine("Hello World!");
```

- The `namespace` area - this specifies that the classes contained within the `{ }` that follow the namespace are part of that namespace. So, similar to how **Console** is part of the **System** namespace, in the example above, the **Program** class is part of the **CaveSensor** namespace, and its full name is **CaveSensor.Program**.
- The `class` area - this defines the contents of the **Program** class. You can have more than one class within a single source file

**Note:** Developers will typically separate classes into their own source file (a single class per source file), especially in larger projects. However, in the labs for this course, you will be including multiple classes per file. This will help to simplify the lab instructions and does not imply best practice.

## 5. On the Visual Studio Code **View** menu, click **Terminal**.

This will open the integrated Terminal at the bottom of the Visual Studio Code window. You will be using the Terminal window to compile and run your console application.

## 6. In the Terminal pane, ensure that the current directory path is set to the `CaveDevice` folder.

The Terminal command prompt includes the current directory path. The commands that you enter are run at the current location, so be sure that you are located in the `CaveDevice` folder.

7. To build and run the **CaveDevice** project, enter the following command:

```
cmd/sh dotnet run
```

8. Notice that **Hello World!** is displayed.

After a moment, you should see **Hello World!** displayed on the line directly below the `dotnet run` command that you entered.

You will be using the same `Console.WriteLine` approach in your simulated device application to display information locally, which will help you see the information being sent to IoT Hub and keep track of processes that are being completed by your device.

Although this Hello World app demonstrates some basic concepts, it is clearly not a simulated device. In the next task you will replace this code with the code for your simulated device.

### Task 3: Implement the simulated device code

In this task, you will use Visual Studio Code to enter the code that leverages the Azure IoT Device SDK to connect to your IoT Hub resource.

1. In the **EXPLORER** pane, click **Program.cs**.
2. Select all of the existing code, and then delete it.
3. In the code editor pane, to create the basic structure of your simulated device application, enter the following code:

**Important:** If you are intending to paste the code into a learning environment such as LODS, there are a few things to be aware of:

- The **Type text -> Type clipboard text** buffer is limited, so it may truncate the code that is copied - double check your work and add any missing characters.
- As the **Type clipboard text** simulates typing, the default settings in Visual Studio Code will automatically indent code and insert closing braces - `)`, `}` and `]` - resulting in duplicate characters and incorrect indentation. These actions can be turned off with the following settings:
- **Editor: Auto Closing Brackets**



- **Editor: Auto Indent**
- The source can be reformatted at any time by using **F1** and entering **Format Document** or by pressing **SHIFT + ALT + F**

```
```csharp // INSERT using statements below here

namespace CaveDevice { class Program { // INSERT variables below
here

    // INSERT Main method below here

    // INSERT SendDeviceToCloudMessagesAsync method below
here

    // INSERT CreateMessageString method below here
}

// INSERT EnvironmentSensor class below here
} ```
```

**Note:** As you can see, the namespace and the class have been retained, however, the other items are placeholder comments. In the following steps you will be inserting code into the file below specific comments.

**Tip:** To reformat the pasted text in Visual Studio Code, press **SHIFT + ALT + F**, or press **F1** to open the command palette and search for **Format Document**.

4. Locate the `// INSERT using statements below here` comment.
5. To specify the namespaces that the application code will be using, enter the following code:

```
csharp using System; using System.Text; using
System.Threading.Tasks; using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
```

Notice that as well as specifying **System**, you are also declaring other namespaces that the code will be using, such as **System.Text** for encoding strings, **System.Threading.Tasks** for asynchronous tasks, and the namespaces for the two packages you added earlier.

**Tip:** When inserting code, the code layout may not be ideal. You can have Visual Studio Code format the document for you by right-clicking in the code editor pane and then clicking **Format**

**Document.** You can achieve the same result by opening the **Task** pane (press **F1**) and typing **Format Document** and then pressing **Enter**. And on Windows, the shortcut for this task is **SHIFT+ALT+F**.

6. Locate the `// INSERT variables below here` comment.
7. To specify the variables that the program is using, enter the following code:

```
```csharp // Contains methods that a device can use to send messages to
and receive from an IoT Hub. private static DeviceClient deviceClient;

// The device connection string to authenticate the device with your IoT
hub. // Note: in real-world applications you would not "hard-code" the
connection string // It could be stored within an environment variable,
passed in via the command-line or // stored securely within a TPM
module. private readonly static string connectionString = "{ Your device
connection string here}"; ```
```

8. Take a moment to review the code (and code comments) that you just entered.

The **deviceClient** variable is used to store an instance of **DeviceClient** - this class comes from the Azure IoT Device SDK and contains methods that a device can use to send messages to and receive from an IoT Hub.

The **connectionString** variable will contain the connection string for the device we created earlier. This value is used by the **DeviceClient** to connect to the IoT Hub.

**Important:** You will see examples in this and other labs throughout this course where connection strings, passwords and other configuration information is hard-coded into the application. This is done solely to simplify the labs and **is not** a recommended practice. As much as possible, security issues like this will be addressed as they come up in the labs. Security topics (and other important considerations) will be addressed during the instructor presentation and in your Student Handbook content in a manner that supports the overall flow of the course. The two may not always align perfectly. As a result, you may be exposed to topics in the labs that are not covered in detail until later in the course.

As noted within the code comments, connection strings and similar configuration values should be supplied via alternative means such as

environment variables, command-line parameters or, better still, stored in secured hardware such as Trusted Platform Modules (TPM).

9. In the code that you just entered, update the value for **connectionString** using the Primary Connection String that you copied from IoT Hub.

Once updated, the **connectionString** variable line should be similar to the following:

```
csharp private readonly static string connectionString =
"HostName=iot-az220-training-dm200420.azure-
devices.net;DeviceId=sensor-th-
0001;SharedAccessKey=hfavUmFgoCPA9feWjyfTx23SUHr+dqG9X193ctdE
d90=";
```

10. Locate the `// INSERT Main method below here` comment.
11. To construct the **Main** method of your simulated device application, enter the following code:

```
```csharp private static void Main(string[] args) { Console.WriteLine("IoT
Hub C# Simulated Cave Device. Ctrl-C to exit.\n");
```

```
// Connect to the IoT hub using the MQTT protocol
deviceClient =
DeviceClient.CreateFromConnectionString(connectionString,
TransportType.Mqtt);
SendDeviceToCloudMessagesAsync();
Console.ReadLine();
```

```
} ```
```

The **Main** method is the first part of your application that runs once your app is started.

12. Take a minute to review the code (and code comments) that you just entered.

The basic structure of a simple device app is as follows:

- Connect to the IoT Hub
- Send telemetry to the app (Device to Cloud messages)

Notice that the **deviceClient** variable is initialized with the result of the **DeviceClient** static method, **CreateFromConnectionString**. This method uses the connection string you specified earlier, as well as selecting the protocol that the device will use to send telemetry - in this case MQTT.

**Note:** In a production application, the **CreateFromConnectionString** method call would be wrapped in exception handling code to gracefully deal with any connection issues. This and other lab code is kept as simple as possible to highlight the key points, so most error-handling is omitted for brevity.

Once connected, the **SendDeviceToCloudMessagesAsync** method is called. You may notice that the method name is underlined with "red squiggles" - this is because Visual Studio Code has noticed that **SendDeviceToCloudMessagesAsync** is not yet implemented. We will add the method shortly.

Finally, the application waits for user input.

**Information:** The **DeviceClient** class is documented [here](#).

**Information:** The **CreateFromConnectionString** method is documented [here](#).

**Information:** The supported transport protocols are documented [here](#).

13. Locate the `// INSERT - SendDeviceToCloudMessagesAsync` below here comment.

14. To construct the **SendDeviceToCloudMessagesAsync** method, enter the following code:

```
``csharp private static async void SendDeviceToCloudMessagesAsync()
{ // Create an instance of our sensor var sensor = new
EnvironmentSensor();

while (true)
{
    // read data from the sensor
    var currentTemperature = sensor.ReadTemperature();
    var currentHumidity = sensor.ReadHumidity();

    var messageString =
CreateMessageString(currentTemperature, currentHumidity);

    // create a byte array from the message string using
ASCII encoding
    var message = new
Message(Encoding.ASCII.GetBytes(messageString));

    // Add a custom application property to the message.
    // An IoT hub can filter on these properties without
```

```

access to the message body.
    message.Properties.Add("temperatureAlert",
(currentTemperature > 30) ? "true" : "false");

    // Send the telemetry message
    await deviceClient.SendEventAsync(message);
    Console.WriteLine("{0} > Sending message: {1}",
DateTime.Now, messageString);

    await Task.Delay(1000);
}

} ``

```

Notice that the declaration for the **SendDeviceToCloudMessagesAsync** method includes the keyword `async`. This specifies that the method contains asynchronous code that uses the `await` keyword and instructs the compiler to handle the callback plumbing for you.

15. Take a minute to review the code (and code comments) that you just entered.

This method implements a typical message loop:

- Read from one or more sensors
- Create a message to send
- Send the message
- Wait for some time, or for an event to occur, etc.
- Repeat the loop

The following description explains the method code in more detail:

- The first thing that your code does is create an instance of the **EnvironmentSensor** class. This is done outside the loop and is used to support simulating the sensor data inside the loop. You will add the **EnvironmentSensor** class shortly.
- You then start an infinite loop - `while(true) {}` will repeat until the user hits **CTRL+C**.
- Within the loop, the first thing you do is read the temperature and humidity from your sensor and use those values to create a message string - you will add the code for **CreateMessageString** in a moment as well.
- Then you create the actual **message** that will be sent to IoT Hub. You do this by creating an instance of the **Message** class from the Azure IoT Device SDK - the data structure that represents the

message that is used for interacting with Iot Hub (IoT Hub expects a specific message format). The constructor that you use for the **Message** class requires that the message string be encoded as a byte array.

- Next, you augment the message with additional properties - here, for example, you set the **temperatureAlert** property to true if the **currentTemperature** is greater than 30, otherwise false.
- You then send the telemetry message via the `await deviceClient.SendEventAsync(message);` call. Note that this line contains an `await` keyword. This instructs the compiler that the following code is asynchronous and will complete some time in the future - when it does complete, this method will continue executing on the next line.
- Finally, you write the message string to the local console window to show that telemetry has been sent to IoT Hub, and then wait for 1000 milliseconds (1 second) before repeating the loop.

**Information:** You can learn more about `async`, `await` and asynchronous programming in C# [here](#).

**Information:** The **Message** class is documented [here](#)

16. Locate the `// INSERT CreateMessageString` method below here comment.

17. To construct the **CreateMessageString** method that creates a JSON string from the sensor readings, enter the following code:

```
```csharp private static string CreateMessageString(double temperature,
double humidity) { // Create an anonymous object that matches the data
structure we wish to send var telemetryDataPoint = new { temperature =
temperature, humidity = humidity };

// Create a JSON string from the anonymous object
return JsonConvert.SerializeObject(telemetryDataPoint);

} ```
```

This method creates an anonymous object with the temperature and humidity properties and assigns it to **telemetryDataPoint**.

The value of **telemetryDataPoint** is then converted to a JSON string via the **JsonConvert** class that is part of the **Newtonsoft.Json** package you

added earlier. The JSON string value is then returned to be used as the payload in the message.

18. Locate the `// INSERT EnvironmentSensor class below here` comment.

19. To construct the **EnvironmentSensor** class, enter the following code:

```
```csharp ///
/// This class represents a sensor /// real-world sensors would contain code
to initialize /// the device or devices and maintain internal state /// a real-
world example can be found here: https://bit.ly/IoT-BME280 ///
internal class EnvironmentSensor { // Initial telemetry values double
minTemperature = 20; double minHumidity = 60; Random rand = new
Random();

internal EnvironmentSensor()
{
    // device initialization could occur here
}

internal double ReadTemperature()
{
    return minTemperature + rand.NextDouble() * 15;
}

internal double ReadHumidity()
{
    return minHumidity + rand.NextDouble() * 20;
}

} ```
```

This is a very simple class that uses random numbers to return values that represent temperature and humidity. In reality, it is often much more complex to interact with sensors, especially if you have to communicate with them at a low-level and derive the measurement value (rather than getting a direct reading in the appropriate units).

**Information:** You can view a more representative example of the code that interacts with a simple temperature, humidity and pressure sensor [here](#).

20. On the **File** menu, click **Save**.

21. Take a minute to scan through your completed application.

Your completed application represents a simple simulated device. It demonstrates how to connect a device to an IoT Hub and send Device to Cloud messages.

You are now ready to test the application

#### Task 4: Test the application

1. In Visual Studio Code Explorer pane, on the **View** menu, click **Terminal**.

Verify that the selected terminal shell is the windows command prompt.

2. In the Terminal view, at the command prompt, enter the following command:

```
cmd/sh dotnet run
```

This command will build and run the Simulated Device application. Be sure the terminal location is set to the directory with the `CaveDevice.cs` file.

**Note:** If the command outputs a `Malformed Token` or other error message, then make sure the **Primary Connection String** value is configured correctly as the value of the `connectionString` variable.

If you receive additional error messages, you can verify that you constructed your code correctly by referring to completed solution code that is available for reference in the **Final** folder for this lab. This **Final** folder is included with the lab resources files that you downloaded when setting up your development environment in lab 3. The folder path is:

- Allfiles
- Labs
  - LAB\_AK\_04-connect-iot-device-to-azure
  - Final

3. Observe the message string output displayed in the Terminal.

Once the Simulated Device application is running, it will be sending event messages to the Azure IoT Hub that include `temperature` and `humidity` values, and displaying message string output in the console.

The terminal output will look similar to the following:

```
``text IoT Hub C# Simulated Cave Device. Ctrl-C to exit.
```



```
10/25/2019 6:10:12 PM > Sending message:
{"temperature":27.714212817472504,"humidity":63.88147743599558}
10/25/2019 6:10:13 PM > Sending message:
{"temperature":20.017463779085066,"humidity":64.53511070671263}
10/25/2019 6:10:14 PM > Sending message:
{"temperature":20.723927165718717,"humidity":74.07808918230147}
10/25/2019 6:10:15 PM > Sending message:
{"temperature":20.48506045736608,"humidity":71.47250854944461}
10/25/2019 6:10:16 PM > Sending message:
{"temperature":25.027703996760632,"humidity":69.21247714628115}
10/25/2019 6:10:17 PM > Sending message:
{"temperature":29.867399432634656,"humidity":78.19206098010395}
10/25/2019 6:10:18 PM > Sending message:
{"temperature":33.29597232085465,"humidity":62.8990878830194}
10/25/2019 6:10:19 PM > Sending message:
{"temperature":25.77350195766124,"humidity":67.27347029711747} ````
```

**Note:** Leave the simulated device app running for now. Your next task will be to verify that your IoT Hub is receiving the telemetry messages.

### Task 3: Verify Telemetry Stream sent to Azure IoT Hub

In this task, you will use the Azure CLI to verify telemetry sent by the simulated device is being received by Azure IoT Hub.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.
2. In the Azure Cloud Shell, to monitor the event messages that are being received by your IoT hub, enter the following command:

```
cmd/sh az iot hub monitor-events --hub-name {IoTHubName} --
device-id sensor-th-0001
```

*Be sure to replace the **{IoTHubName}** placeholder with the name of your Azure IoT Hub.*

**Note:** If you receive a message stating *"Dependency update required for IoT extension version"* when running the Azure CLI command, then press `y` to accept the update and press `Enter`. This will allow the command to continue as expected.

The `monitor-events` command (within the `az iot hub` Azure CLI module) offers the capability to monitor device telemetry and other message types sent to an Azure IoT Hub. This can be a very useful tool

during code development, and the convenience of the command-line interface is also nice.

The `--device-id` parameter is optional and allows you to monitor the events from a single device. If the parameter is omitted, the command will monitor all events sent to the specified Azure IoT Hub.

3. Notice that the `az iot hub monitor-events` Azure CLI command outputs a JSON representation of the events that are arriving at your specified Azure IoT Hub.

This command enables you to monitor the events being sent to IoT hub. You are also verifying that the device is able to connect to and communicate with the your IoT hub.

You should see messages displayed that are similar to the following:

```
cmd/sh Starting event monitor, filtering on device: sensor-  
th-0001, use ctrl-c to stop... { "event": { "origin":  
"sensor-th-0001", "payload": "  
{\"temperature\":25.058683971901743,\"humidity\":67.548169813  
83979}\" } } { "event": { "origin": "sensor-th-0001",  
"payload": "  
{\"temperature\":29.202181296051563,\"humidity\":69.138403036  
23043}\" } }
```

4. Once you have verified that IoT hub is receiving the telemetry, press **Ctrl-C** in the Azure Cloud Shell and Visual Studio Code windows.

**Ctrl-C is used to stop the running apps. Always remember to shut down unneeded apps and jobs.**

lab: title: 'Lab 05: Individual Enrollment of a Device in DPS' module: 'Module 3: Device Provisioning at Scale'

---

# Individual Enrollment of a Device in DPS

## Lab Scenario

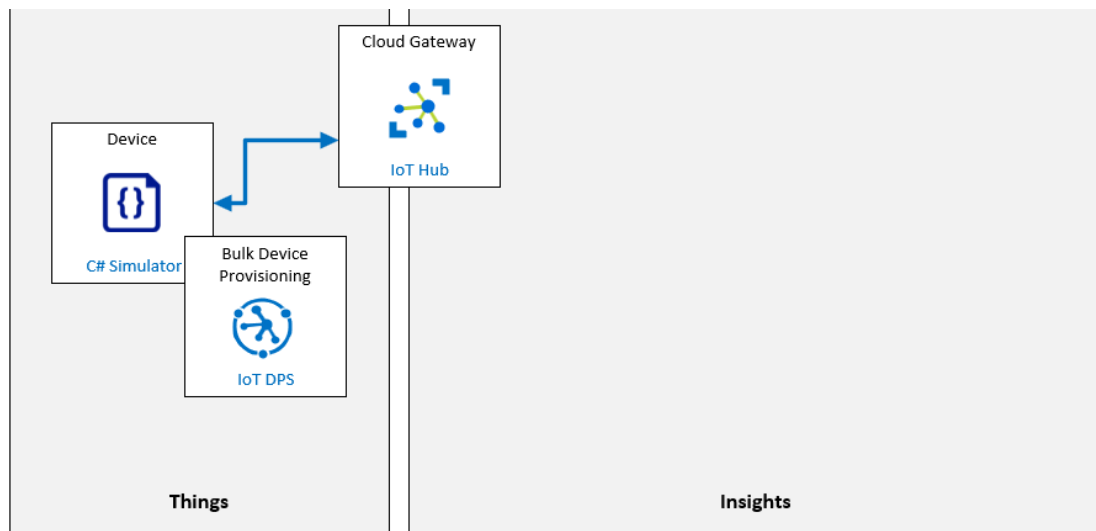
Contoso management is pushing for an update to their existing Asset Monitoring and Tracking Solution. The update will use IoT devices to reduce the manual data entry work that is required under the current system and provide more advanced monitoring during the shipping process. The solution relies on the ability to provision IoT devices when shipping containers are loaded and deprovision the devices when the container arrives at the destination. The best option for managing the provisioning requirements appears to be the IoT Hub Device Provisioning Service (DPS).

The proposed system will use IoT devices with integrated sensors for tracking the location, temperature, and pressure of shipping containers during transit. The IoT devices will be placed within the existing shipping containers that Contoso uses to transport their cheese, and will connect to Azure IoT Hub using vehicle-provided WiFi. The new system will provide continuous monitoring of the product environment and enable a variety of notification scenarios when issues are detected. The rate at which telemetry is sent to IoT hub must be configurable.

In Contoso's cheese packaging facility, when an empty container enters the system it will be equipped with the new IoT device and then loaded with packaged cheese products. The IoT device will be auto-provisioned to IoT hub using DPS. When the container arrives at the destination, the IoT device will be retrieved and must be fully deprovisioned (disenrolled and deregistered). The recovered devices will be recycled and re-used for future shipments following the same auto-provisioning process.

You have been tasked with validating the device provisioning and deprovisioning process using DPS. For the initial testing phase you will use an Individual Enrollment approach.

The following resources will be created:



## In This Lab

In this lab, you will begin by reviewing the lab prerequisites and you will run a script if needed to ensure that your Azure subscription includes the required resources. You will then create a new individual enrollment in DPS that uses Symmetric Key attestation and specifies an initial Device Twin State (telemetry rate) for the device. With the device enrollment saved, you will go back into the enrollment and get the auto-generated Primary and Secondary keys needed for device attestation. Next, you create a simulated device and verify that device connects successfully with IoT hub and that the initial device twin properties are applied by the device as expected. To finish up, you will complete a deprovisioning process that securely removes the device from your solution by both disenrolling and deregistering the device (from DPS and IoT hub respectively). The lab includes the following exercises:

- Verify Lab Prerequisites
- Create new individual enrollment (Symmetric keys) in DPS
- Configure Simulated Device
- Test the Simulated Device
- Deprovision the Device

# Lab Instructions

## Exercise 1: Verify Lab Prerequisites

This lab assumes that the following Azure resources are available:

Resource Type	Resource Name
Resource Group	rg-az220
IoT Hub	iot-az220-training-{your-id}
Device Provisioning Service	dps-az220-training-{your-id}

If these resources are not available, you will need to run the **lab05-setup.azcli** script as instructed below before moving on to Exercise 2. The script file is included in the GitHub repository that you cloned locally as part of the dev environment configuration (lab 3).

The **lab05-setup.azcli** script is written to run in a **bash** shell environment - the easiest way to execute this is in the Azure Cloud Shell.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.

If you are prompted about setting up storage for Cloud Shell, accept the defaults.

2. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

3. On the Cloud Shell toolbar, click **Upload/Download files** (fourth button from the right).
4. In the dropdown, click **Upload**.
5. In the file selection dialog, navigate to the folder location of the GitHub lab files that you downloaded when you configured your development environment.

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and

extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 05-Individual Enrollment of a Device in DPS
  - Setup

The lab05-setup.azcli script file is located in the Setup folder for lab 5.

6. Select the **lab05-setup.azcli** file, and then click **Open**.

A notification will appear when the file upload has completed.

7. To verify that the correct file has uploaded, enter the following command:

```
bash ls
```

The `ls` command lists the content of the current directory. You should see the lab05-setup.azcli file listed.

8. To create a directory for this lab that contains the setup script and then move into that directory, enter the following Bash commands:

```
bash mkdir lab5 mv lab05-setup.azcli lab5 cd lab5
```

These commands will create a directory for this lab, move the **lab05-setup.azcli** file into that directory, and then change directory to make the new directory the current working directory.

9. To ensure the **lab05-setup.azcli** script has the execute permission, enter the following command:

```
bash chmod +x lab05-setup.azcli
```

10. On the Cloud Shell toolbar, to enable access to the lab05-setup.azcli file, click **Open Editor** (second button from the right - { }).

11. In the **Files** list, to expand the lab5 folder and open the script file, click **lab5**, and then click **lab05-setup.azcli**.

The editor will now show the contents of the **lab05-setup.azcli** file.

12. In the editor, update the values of the {your-id} and {your-location} variables.



Referencing the sample below as an example, you need to set `{your-id}` to the Unique ID you created at the start of this course - i.e. **cah191211**, and set `{your-location}` to the location that makes sense for your resources.

```
```bash
```

# !/bin/bash

## Change these values!

YourID="{your-id}" Location="{your-location}" ``

**Note:** The {your-location} variable should be set to the short name for the region where you are deploying all of your resources. You can see a list of the available locations and their short-names (the **Name** column) by entering this command:

```
``bash az account list-locations -o Table
```

DisplayName Latitude Longitude Name

---

```
East Asia 22.267 114.188 eastasia Southeast Asia 1.283 103.833  
southeastasia Central US 41.5908 -93.6208 centralus East US 37.3719  
-79.8164 eastus East US 2 36.6681 -78.3889 eastus2 ``
```

13. In the top-right of the editor window, to save the changes made to the file and close the editor, click ..., and then click **Close Editor**.

If prompted to save, click **Save** and the editor will close.

**Note:** You can use **CTRL+S** to save at any time and **CTRL+Q** to close the editor.

14. To create the resources required for this lab, enter the following command:

```
bash ./lab05-setup.azcli
```

This will take a few minutes to run. You will see output as each step completes.

Once the script has completed, you will be ready to continue with the lab.

## Exercise 2: Create new individual enrollment (Symmetric keys) in DPS

In this exercise, you will create a new individual enrollment for a device within the Device Provisioning Service (DPS) using *symmetric key attestation*. You will also configure the initial device state within the enrollment. After saving your enrollment, you will go back in and obtain the auto-generated attestation Keys that get created when the enrollment is saved.

### Task 1: Create the enrollment

1. If necessary, log in to [portal.azure.com](https://portal.azure.com) using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. Notice that the **AZ-220** dashboard has been loaded and your Resources tile is displayed.

You should see both your IoT Hub and DPS resources listed.

3. On the **rg-az220** resource group tile, click **dps-az220-training-{your-id}**.
4. On the left-side menu under **Settings**, click **Manage enrollments**.
5. At the top of the **Manage enrollments** pane, click **+ Add individual enrollment**.
6. On the **Add Enrollment** blade, in the **Mechanism** dropdown, click **Symmetric Key**.

This sets the attestation method to use Symmetric key authentication.

7. Just below the Mechanism setting, notice that the **Auto-generate keys** option is checked.

This sets DPS to automatically generate both the **Primary Key** and **Secondary Key** values for the device enrollment when it's created. Optionally, un-checking this option enables custom keys to be manually entered.

**Note:** The Primary Key and Secondary Key values are generated after this record is saved. In the next task you will go back into this record to obtain the values, and then use them within a simulated device app later in this lab.

8. In the **Registration ID** field, to specify the Registration ID to use for the device enrollment within DPS, enter **sensor-thl-1000**

By default, the Registration ID will be used as the IoT Hub Device ID when the device is provisioned from the enrollment. When these values need to be different, you can enter the required IoT Hub Device ID in that field.

9. Leave the **IoT Hub Device ID** field blank.

Leaving this field blank ensures that the IoT Hub will use the Registration ID as the Device ID. Don't worry if you see a default text value in the field that is not selectable - this is placeholder text and will not be treated as an entered value.

10. Leave the **IoT Edge device** field set to **False**.

The new device will not be an edge device. Working with IoT Edge devices will be discussed later in the course.

11. Leave the **Select how you want to assign devices to hubs** field set to **Evenly weighted distribution**.

As you only have one IoT Hub associated with the enrollment, this setting is somewhat unimportant. In larger environments where you have multiple distributed hubs, this setting will control how to choose what IoT Hub should receive this device enrollment. There are four supported allocation policies:

- **Lowest latency:** Devices are provisioned to an IoT hub based on the hub with the lowest latency to the device.
- **Evenly weighted distribution (default):** Linked IoT hubs are equally likely to have devices provisioned to them. This is the default setting. If you are provisioning devices to only one IoT hub, you can keep this setting.
- **Static configuration via the enrollment list:** Specification of the desired IoT hub in the enrollment list takes priority over the Device Provisioning Service-level allocation policy.
- **Custom (Use Azure Function):** the device provisioning service calls your Azure Function code providing all relevant information about the device and the enrollment. Your function code is executed and returns the IoT hub information used to provisioning the device.

12. Notice that the **Select the IoT hubs this device can be assigned to** dropdown specifies the **iot-az220-training-{your-id}** IoT hub that you created.

This field is used to specify the IoT Hub(s) that your device can be assigned to.

13. Leave the **Select how you want device data to be handled on re-provisioning** field set to the default value of **Re-provision and migrate data**.

This field gives you high-level control over the re-provisioning behavior, where the same device (as indicated through the same Registration ID) submits a later provisioning request after already being provisioned successfully at least once. There are three options available:

- **Re-provision and migrate data:** This policy is the default for new enrollment entries. This policy takes action when devices associated with the enrollment entry submit a new provisioning request. Depending on the enrollment entry configuration, the device may be reassigned to another IoT hub. If the device is changing IoT hubs, the device registration with the initial IoT hub will be removed. All device state information from that initial IoT hub will be migrated over to the new IoT hub.
- **Re-provision and reset to initial config:** This policy is often used for a factory reset without changing IoT hubs. This policy takes action when devices associated with the enrollment entry submit a new provisioning request. Depending on the enrollment entry configuration, the device may be reassigned to another IoT hub. If the device is changing IoT hubs, the device registration with the initial IoT hub will be removed. The initial configuration data that the provisioning service instance received when the device was provisioned is provided to the new IoT hub.
- **Never re-provision:** The device is never reassigned to a different hub. This policy is provided for managing backwards compatibility.

14. In the **Initial Device Twin State** field, to specify a property named `telemetryDelay` with the value of "2", update the JSON object as follows:

The final JSON will be like the following:

```
json { "tags": {}, "properties": { "desired": {  
  "telemetryDelay": "2" } } }
```

This field contains JSON data that represents the initial configuration of desired properties for the device. The data that you entered will be used by the Device to set the time delay for reading sensor telemetry and sending events to IoT Hub.

15. Leave the **Enable entry** field set to **Enable**.

Generally, you'll want to enable new enrollment entries and keep them enabled.

16. At the top of the **Add Enrollment** blade, click **Save**.

## Task 2: Review Enrollment and Obtain Authentication Keys

1. On the **Manage enrollments** pane, to view the list of individual device enrollments, click **individual enrollments**.

As you may recall, you will be using the enrollment record to obtain the Authentication keys.

2. Under **REGISTRATION ID**, click **sensor-thl-1000**.

This blade enables you to view the enrollment details for the individual enrollment that you just created.

3. Locate the **Authentication Type** section.

Since you specified the Authentication Type as Symmetric Key when you created the enrollment, the Primary and Secondary key values have been created for you. Notice that there is a button to the right of each textbox that you can use to copy the values.

4. Copy the **Primary Key** and **Secondary Key** values for this device enrollment, and then save them to a file for later reference.

These are the authentication keys for the device to authenticate with the IoT Hub service.

5. Locate the **Initial device twin State**, and notice the JSON for the device twin Desired State contains the `telemetryDelay` property set to the value of "2".

6. Close the **sensor-thl-1000** individual enrollment blade.

## Exercise 3: Configure Simulated Device

In this exercise, you will configure a Simulated Device written in C# to connect to Azure IoT using the individual enrollment created in the previous exercise. You will also add code to the Simulated Device that will read and update device configuration based on the device twin within Azure IoT Hub.

The simulated device that you create in this exercise represents an IoT device that will be located within a shipping container/box, and will be used to monitor Contoso products while they are in transit. The sensor telemetry from the device that will be sent to Azure IoT Hub includes Temperature, Humidity, Pressure, and Latitude/Longitude coordinates of the container. The device is part of the overall asset tracking solution.

**Note:** You may have the impression that creating this simulated device is a bit redundant with what you created in the previous lab, but the attestation mechanism that you implement in this lab is quite different from what you did previously. In the previous lab, you used a shared access key to authenticate, which does not require device provisioning, but also does not give the provisioning management benefits (such as leveraging device twins), and it requires fairly large distribution and management of a shared key. In this lab, you are provisioning a unique device through the Device Provisioning Service.

### Task 1: Create the Simulated Device

1. On the left-side menu of the **dps-az220-training-{your-id}** blade, click **Overview**.
2. In the top-right area of the blade, hover the mouse pointer over value assigned to **ID Scope**, and then click **Copy to clipboard**.

You will be using this value shortly, so make note of the value if you are unable to use the clipboard. Be sure to differentiate between uppercase "O" and the number "0".

The **ID Scope** will be similar to this value: 0ne0004E52G

3. Open **Visual Studio Code**.
4. On the **File** menu, click **Open Folder** and then navigate to the Starter folder for Lab 5.

The Lab 5 Starter folder is part of the lab resources files that you downloaded when setting up your development environment in lab 3. The folder path is:

- Allfiles
- Labs
  - 05-Individual Enrollment of a Device in DPS
    - Starter

5. In the **Open Folder** dialog, click **ContainerDevice**, and then click **Select Folder**.

The ContainerDevice folder is a sub-folder of the Lab 5 Starter folder. It contains a Program.cs file and a ContainerDevice.csproj file.

**Note:** If Visual Studio Code prompts you to load required assets, you can click **Yes** to load them.

6. On the **View** menu, click **Terminal**.

Verify that the selected terminal shell is the windows command prompt.

7. At the Terminal command prompt, to restore all the application NuGet packages, enter the following command:

```
cmd/sh dotnet restore
```

8. In the Visual Studio Code **EXPLORER** pane, click **Program.cs**.
9. In the code editor, near the top of the Program class, locate the **dpsIdScope** variable.
10. Update the value assigned to **dpsIdScope** using the ID Scope that you copied from the Device Provisioning Service.

**Note:** If you don't have the value of ID Scope available to you, you can find it on the Overview blade of the DPS service (in the Azure portal).

11. Locate the **registrationId** variable, and update the assigned value using **sensor-thl-1000**

This variable represents the **Registration ID** value for the individual enrollment that you created in the Device Provisioning Service.

12. Update the **individualEnrollmentPrimaryKey** and **individualEnrollmentSecondaryKey** variables using the **Primary Key** and **Secondary Key** values that you saved.

**Note:** If you don't have these Key values available, you can copy them from the Azure portal as follows -

Open the **Manage enrollments** blade, click **Individual Enrollments**, click **sensor-thl-1000**. Copy the values and then paste as noted above.



## Task 2: Add the provisioning code

In this task, you will implement the code that provisions the device via DPS and creates a DeviceClient instance that can be used to connect to the IoT Hub.

1. Take a minute to scan through the code in the **Program.cs** file.

The overall layout of the **ContainerDevice** application is similar to the **CaveDevice** application that you created in Lab 4. Notice that both applications include the following:

- Using statements
  - Namespace definition
  - Program class - responsible for connecting to Azure IoT and sending telemetry
  - EnvironmentSensor class - responsible for generating sensor data
2. In the code editor, locate the `// INSERT Main method below here` comment.
  3. To create the **Main** method for your simulated device application, enter the following code:

```
``csharp public static async Task Main(string[] args) {

    using (var security = new
        SecurityProviderSymmetricKey(registrationId,

            individualEnrollmentPrimaryKey,

            individualEnrollmentSecondaryKey))
        using (var transport = new
            ProvisioningTransportHandlerAmqp(TransportFallbackType.TcpOnly))
        {
            ProvisioningDeviceClient provClient =
                ProvisioningDeviceClient.Create(GlobalDeviceEndpoint,
                    dpsIdScope, security, transport);

            using (deviceClient = await ProvisionDevice(provClient,
                security))
            {
                await deviceClient.OpenAsync().ConfigureAwait(false);

                // INSERT Setup OnDesiredPropertyChanged Event
                Handling below here

                // INSERT Load Device Twin Properties below here

                // Start reading and sending device telemetry
            }
        }
    }
}
```

```

        Console.WriteLine("Start reading and sending device
telemetry...");
        await SendDeviceToCloudMessagesAsync(deviceClient);

        await
deviceClient.CloseAsync().ConfigureAwait(false);
    }
}

} ``

```

Although the **Main** method in this application serves a similar purpose to the **Main** method of the **CaveDevice** application that you created in a previous lab, it is a little more complex. In the **CaveDevice** app, you used a device connection string to directly connect to an IoT Hub, this time you need to first provision the device (or, for subsequent connections, confirm the device is still provisioned), then retrieve the appropriate IoT Hub connection details.

To connect to DPS, you not only require the **dpsScopeId** and the **GlobalDeviceEndpoint** (defined in the variables), you also need to specify the following:

- **security** - the method used for authenticating the enrollment. Earlier you configured the individual enrollment to use symmetric keys, therefore the **SecurityProviderSymmetricKey** is the logical choice. As you might expect, there are variants of the providers that support X.509 and TPM as well.
- **transport** - the transport protocol used by the provisioned device. In this instance, the AMQP handler was chosen (**ProvisioningTransportHandlerAmqp**). Of course, HTTP and MQTT handlers are also available.

Once the **security** and **transport** variables are populated, you create an instance of the **ProvisioningDeviceClient**. You will use this instance to register the device and create a **DeviceClient** in the **ProvisionDevice** method, which you will add shortly.

The remainder of the **Main** method uses the device client a little differently than you did in the **CaveDevice** - this time you explicitly open the device connection so that the app can use device twins (more on this in the next exercise), and then call the **SendDeviceToCloudMessagesAsync** method to start sending telemetry.

The **SendDeviceToCloudMessagesAsync** method is very similar to what you created in the **CaveDevice** application. It creates an instance of the

**EnvironmentSensor** class (this one also returns pressure and location data), builds a message and sends it. Notice that instead of a fixed delay within the method loop, the delay is calculated by using the **telemetryDelay** variable: `await Task.Delay(telemetryDelay * 1000);`. If time permits, take a deeper look yourself and compare this to the class used in the earlier lab.

Finally, back in the **Main** method, the device client is closed.

**Information:** You can find the documentation for the **ProvisioningDeviceClient** [here](#) - from there you can easily navigate to the other related classes.

4. Locate the `// INSERT ProvisionDevice method below here` comment.

5. To create the **ProvisionDevice** method, enter the following code:

```
```csharp private static async Task
ProvisionDevice(ProvisioningDeviceClient provisioningDeviceClient,
SecurityProviderSymmetricKey security) { var result = await
provisioningDeviceClient.RegisterAsync().ConfigureAwait(false);
Console.WriteLine($"ProvisioningClient AssignedHub:
{result.AssignedHub}; DeviceID: {result.DeviceId}"); if (result.Status !=
ProvisioningRegistrationStatusType.Assigned) { throw new
Exception($"DeviceRegistrationResult.Status is NOT 'Assigned'"); }

var auth = new DeviceAuthenticationWithRegistrySymmetricKey(
    result.DeviceId,
    security.GetPrimaryKey());

return DeviceClient.Create(result.AssignedHub, auth,
TransportType.Amqp);

} ```
```

As you can see, this method receives the the provisioning device client and security instances you created earlier. The `provisioningDeviceClient.RegisterAsync()` is called, which returns a **DeviceRegistrationResult** instance. This result contains a number of properties including the **DeviceId**, **AssignedHub** and the **Status**.

**Information:** Full details of the **DeviceRegistrationResult** properties can be found [here](#).

The method then checks to ensure that the provisioning status has been set and throws an exception if the device is not *Assigned* - other possible results here include *Unassigned*, *Assigning*, *Failed* and *Disabled*.

- The **Program.ProvisionDevice** method contains the logic for registering the device via DPS.
- The **Program.SendDeviceToCloudMessagesAsync** method sends the telemetry as Device-to-Cloud messages to Azure IoT Hub.
- The **EnvironmentSensor** class contains the logic for generating the simulated sensor readings for Temperature, Humidity, Pressure, Latitude, and Longitude.

6. Locate the **SendDeviceToCloudMessagesAsync** method.

7. At the bottom of the **SendDeviceToCloudMessagesAsync** method, notice the call to `Task.Delay()`.

`Task.Delay()` is used to "pause" the `while` loop for a period of time before creating and sending the next telemetry message. The **telemetryDelay** variable is used to define how many seconds to wait before sending the next telemetry message. Contoso is requiring that the delay time be configurable.

8. Near the top of the **Program** class, locate the **telemetryDelay** variable declaration.

Notice that the default value for the delay is set to **1** second. Your next step is to integrate the code that uses a device twin value to control the delay time.

### Task 3: Integrate Device Twin Properties

In order to use the device twin properties (from Azure IoT Hub) on a device, you need to create the code that accesses and applies the device twin properties. In this case, you want to update your simulated device code to read the `telemetryDelay` device twin Desired Property, and then assign that value to the corresponding **telemetryDelay** variable in your code. You also want to update the device twin Reported Property (maintained by IoT Hub) to have a record of the delay time that is currently implemented on our device.

1. In the Visual Studio Code editor, locate the **Main** method.

To begin the integration of device twin properties, your code that enables the simulated device needs to be notified when a device twin property is updated.

To achieve this, you will use the

`DeviceClient.SetDesiredPropertyUpdateCallbackAsync` method, and

set up an event handler by creating an `OnDesiredPropertyChanged` method.

2. Locate the `// INSERT Setup OnDesiredPropertyChanged Event Handling below here` comment.
3. To set up the `DeviceClient` for an `OnDesiredPropertyChanged` event, enter the following code:

```
csharp await
deviceClient.SetDesiredPropertyUpdateCallbackAsync(OnDesiredP
ropertyChanged, null).ConfigureAwait(false);
```

The **`SetDesiredPropertyUpdateCallbackAsync`** method is used to set up the **`DesiredPropertyUpdateCallback`** event handler to receive device twin desired property changes. This code configures **`deviceClient`** to call a method named **`OnDesiredPropertyChanged`** when a device twin property change event is received.

Now that the **`SetDesiredPropertyUpdateCallbackAsync`** method is in place to set up the event handler, you need to create the **`OnDesiredPropertyChanged`** method that it calls.

4. Locate the `// INSERT OnDesiredPropertyChanged method below here` comment.
5. To create the **`OnDesiredPropertyChanged`** method, enter the following code:

```
```csharp private static async Task
OnDesiredPropertyChanged(TwinCollection desiredProperties, object
userContext) { Console.WriteLine("Desired Twin Property Changed:");
Console.WriteLine($"{desiredProperties.ToJson()}");

// Read the desired Twin Properties
if (desiredProperties.Contains("telemetryDelay"))
{
    string desiredTelemetryDelay =
desiredProperties["telemetryDelay"];
    if (desiredTelemetryDelay != null)
    {
        telemetryDelay = int.Parse(desiredTelemetryDelay);
    }
    // if desired telemetryDelay is null or unspecified,
don't change it
}

// Report Twin Properties
var reportedProperties = new TwinCollection();
```

```

reportedProperties["telemetryDelay"] =
telemetryDelay.ToString();
await
deviceClient.UpdateReportedPropertiesAsync(reportedProperties
).ConfigureAwait(false);
Console.WriteLine("Reported Twin Properties:");
Console.WriteLine($"{reportedProperties.ToJson()}");

} ``

```

Notice that the **OnDesiredPropertyChanged** event handler accepts a **desiredProperties** parameter of type **TwinCollection**.

Notice that if the value of the **desiredProperties** parameter contains **telemetryDelay** (a device twin desired property), the code will assign the value of the device twin property to the **telemetryDelay** variable. You may recall that the **SendDeviceToCloudMessagesAsync** method includes a **Task.Delay** call that uses the **telemetryDelay** variable to set the delay time between messages sent to IoT hub.

Notice the next block of code is used to report the current state of the device back up to Azure IoT Hub. This code calls the **DeviceClient.UpdateReportedPropertiesAsync** method and passes it a **TwinCollection** that contains the current state of the device properties. This is how the device reports back to IoT Hub that it received the device twin desired properties changed event, and has now updated its configuration accordingly. Note that it reports what the properties are now set to, not an echo of the desired properties. In the case where the reported properties sent from the device are different than the desired state that the device received, IoT Hub will maintain an accurate Device Twin that reflects the state of the device.

Now that the device can receive updates to the device twin desired properties from Azure IoT Hub, it also needs to be coded to configure its initial setup when the device starts up. To do this the device will need to load the current device twin desired properties from Azure IoT Hub, and configure itself accordingly.

6. In the **Main** method, locate the `// INSERT Load Device Twin Properties below here` comment.
7. To read the device twin desired properties and configure the device to match on device startup, enter following code:

```

csharp var twin = await
deviceClient.GetTwinAsync().ConfigureAwait(false); await
OnDesiredPropertyChanged(twin.Properties.Desired, null);

```

This code calls the `DeviceTwin.GetTwinAsync` method to retrieve the device twin for the simulated device. It then accesses the `Properties.Desired` property object to retrieve the current Desired State for the device, and passes that to the **OnDesiredPropertyChanged** method that will configure the simulated devices **telemetryDelay** variable.

Notice, this code reuses the **OnDesiredPropertyChanged** method that was already created for handling *OnDesiredPropertyChanged* events. This helps keep the code that reads the device twin desired state properties and configures the device at startup in a single place. The resulting code is simpler and easier to maintain.

8. On the Visual Studio Code **File** menu, click **Save**.

Your simulated device will now use the device twin properties from Azure IoT Hub to set the delay between telemetry messages.

## Exercise 4: Test the Simulated Device

In this exercise, you will run the Simulated Device and verify that it's sending sensor telemetry to Azure IoT Hub. You will also change the rate at which telemetry is sent to Azure IoT Hub by updating the `telemetryDelay` device twin property for the simulated device within Azure IoT Hub.

### Task 1: Build and run the device

1. Ensure that you have your code project open in Visual Studio Code.
2. On the **View** menu, click **Terminal**.
3. In the Terminal pane, ensure the command prompt shows the directory path for the `Program.cs` file.
4. At the command prompt, to build and run the Simulated Device application, enter the following command:

```
cmd/sh dotnet run
```

**Note:** When the Simulated Device application runs, it will first write some details about its status to the console (terminal pane).

5. Notice that the JSON output following the `Desired Twin Property Changed:` line contains the desired value for the `telemetryDelay` for the device.

You can scroll up in the terminal pane to review the output. It should be similar to the following:

```
text ProvisioningClient AssignedHub: iot-az220-training-
{your-id}.azure-devices.net; DeviceID: sensor-thl-1000
Desired Twin Property Changed:
{"telemetryDelay":"2","$version":1} Reported Twin Properties:
{"telemetryDelay":"2"} Start reading and sending device
telemetry...
```

6. Notice that the Simulated Device application begins sending telemetry events to the Azure IoT Hub.

The telemetry events include values for temperature, humidity, pressure, latitude, and longitude, and should be similar to the following:

```
text 11/6/2019 6:38:55 PM > Sending message:
{"temperature":25.59094770373355,"humidity":71.17629229611545
,"pressure":1019.9274696347665,"latitude":39.82133964767944,"
longitude":-98.18181981142438} 11/6/2019 6:38:57 PM > Sending
message:
{"temperature":24.68789062681044,"humidity":71.52098010830628
,"pressure":1022.6521258267584,"latitude":40.05846882452387,"
longitude":-98.08765031156229} 11/6/2019 6:38:59 PM > Sending
message:
{"temperature":28.087463226675737,"humidity":74.7607135375778
7,"pressure":1017.614206096327,"latitude":40.269273772972454,
"longitude":-98.28354453319591} 11/6/2019 6:39:01 PM >
Sending message:
{"temperature":23.575667940813894,"humidity":77.6640950691253
4,"pressure":1017.0118147748344,"latitude":40.21020096551372,
"longitude":-98.48636739129239}
```

Notice the timestamp differences between telemetry readings. The delay between telemetry messages should be 2 seconds as configured through the device twin; instead of the default of 1 second in the source code.

7. Leave the simulated device app running.

You will verify that the device code is behaving as expected during the next activities.

## **Task 2: Verify Telemetry Stream sent to Azure IoT Hub**

In this task, you will use the Azure CLI to verify telemetry sent by the simulated device is being received by Azure IoT Hub.



1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.
2. In the Azure Cloud Shell, enter the following command:

```
cmd/sh az iot hub monitor-events --hub-name {IoTHubName} --  
device-id sensor-thl-1000
```

*Be sure to replace the **{IoTHubName}** placeholder with the name of your Azure IoT Hub.*

3. Notice that your IoT hub is receiving the telemetry messages from the sensor-thl-1000 device.

Continue to leave the simulated device application running for the next task.

### **Task 3: Change the device configuration through its twin**

With the simulated device running, the `telemetryDelay` configuration can be updated by editing the device twin Desired State within Azure IoT Hub. This can be done by configuring the Device in the Azure IoT Hub within the Azure portal.

1. Open the Azure portal (if it is not already open), and then navigate to your **Azure IoT Hub** service.
2. On the IoT Hub blade, on the left-side menu under **Explorers**, click **IoT devices**.
3. Under **DEVICE ID**, click **sensor-thl-1000**.

**IMPORTANT:** Make sure you select the device that you are using for this lab.

4. On the **sensor-thl-1000** device blade, at the top of the blade, click **Device Twin**.

The **Device twin** blade provides an editor with the full JSON for the device twin. This enables you to view and/or edit the device twin state directly within the Azure portal.

5. Locate the JSON for the `properties.desired` object.

This contains the desired state for the device. Notice the `telemetryDelay` property already exists, and is set to "2", as was configured when the

device was provisioned based on the Individual Enrollment in DPS.

6. To update the value assigned to the `telemetryDelay` desired property, change the value to "5"

The value includes the quotes ("").

7. At the top of the **Device twin** blade, click **Save**

The `OnDesiredPropertyChanged` event will be triggered automatically within the code for the Simulated Device, and the device will update its configuration to reflect the changes to the device twin Desired state.

8. Switch to the Visual Studio Code window that you are using to run the simulated device application.

9. In Visual Studio Code, scroll to the bottom of the Terminal pane.

10. Notice that the device recognizes the change to the device twin properties.

The output will show a message that the `Desired Twin Property Changed` along with the JSON for the new `desiredtelemetryDelay` property value. Once the device picks up the new configuration of device twin desired state, it will automatically update to start sending sensor telemetry every 5 seconds as now configured.

```
text Desired Twin Property Changed:
{"telemetryDelay":"5","$version":2} Reported Twin Properties:
{"telemetryDelay":"5"} 4/21/2020 1:20:16 PM > Sending
message:
{"temperature":34.417625961088405,"humidity":74.1240352644231
3,"pressure":1023.7792049974805,"latitude":40.172799921919186
,"longitude":-98.28591913777421} 4/21/2020 1:20:22 PM >
Sending message:
{"temperature":20.963297521678403,"humidity":68.3691603263696
5,"pressure":1023.7596862048422,"latitude":39.83252821949164,
"longitude":-98.31669969393461}
```

11. Switch to the browser page where you are running the Azure CLI command in the Azure Cloud Shell.

Ensure that you are still running the `az iot hub monitor-events` command. If it isn't running, re-start the command.

12. Notice that the telemetry events sent to Azure IoT Hub being received at the new interval of 5 seconds.

13. Use **Ctrl-C** to stop both the `az` command and the Simulated Device application.
14. Switch to your browser window for the Azure portal.
15. Close the **Device twin** blade.
16. Still in the Azure Portal, on the **sensor-thl-1000** device blade, click **Device Twin**.
17. Locate the JSON for the `properties.reported` object.

This portion of the JSON contains the state reported by the device. Notice the `telemetryDelay` property exists here as well, and is also set to 5.

There is also a `$metadata` value that shows you when the value was reported data was last updated and when the specific reported value was last updated.

18. Close the **Device twin** blade.
19. Close the simulated device blade, and then close the IoT Hub blade.

## Exercise 5: Deprovision the Device

In your Contoso scenario, when the shipping container arrives at its final destination, the IoT device will be removed from the container and returned to a Contoso location. Contoso will need to deprovision the device before it can be tested and placed in inventory. In the future the device could be provisioned to the same IoT hub or an IoT hub in a different region. Complete device deprovisioning is an important step in the life cycle of IoT devices within an IoT solution.

In this exercise, you will perform the tasks necessary to deprovision the device from both the Device Provisioning Service (DPS) and Azure IoT Hub. To fully deprovision an IoT device from an Azure IoT solution it must be removed from both of these services.

### Task 1: Disenroll the device from the DPS

1. If necessary, log in to your Azure portal using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. On your Resource group tile, to open your Device Provisioning Service, click **dps-az220-training-{your-id}**.
3. On the left-side menu under **Settings**, click **Manage enrollments**.
4. On the **Manage enrollments** pane, to view the list of individual device enrollments, click **Individual Enrollments**.
5. To the left of **sensor-thl-1000**, click the checkbox.

**Note:** You don't want to open the sensor-thl-1000 individual device enrollment, you just want to select it.

6. At the top of the blade, click **Delete**.

**Note:** Deleting the individual enrollment from DPS will permanently remove the enrollment. To temporarily disable the enrollment, you can set the **Enable entry** setting to **Disable** within the **Enrollment Details** for the individual enrollment.

7. On the **Remove enrollment** prompt, click **Yes**.

The individual enrollment is now removed from the Device Provisioning Service (DPS). To complete the deprovisioning process, the **Device ID** for the Simulated Device also must be removed from the **Azure IoT Hub** service.

## **Task 2: Deregister the device from the IoT Hub**

1. In the Azure portal, navigate back to your Dashboard.
2. On your Resource group tile, to open your Azure IoT Hub blade, click **iot-az220-training-{your-id}**.
3. On the left-side menu under **Explorers**, click **IoT devices**.
4. To the left of **sensor-thl-1000**, click the checkbox.

**IMPORTANT:** Make sure you select the device representing the simulated device that you used for this lab.

5. At the top of the blade, click **Delete**.
6. On the **Are you certain you wish to delete selected device(s)** prompt, click **Yes**.

**Note:** Deleting the device ID from IoT Hub will permanently remove the device registration. To temporarily disable the device from connecting to IoT Hub, you can set the **Enable connection to IoT Hub** to **Disable** within the properties for the device.

**Now that the Device Enrollment has been removed from the Device Provisioning Service, and the matching Device ID has been removed from the Azure IoT Hub, the simulated device has been fully retired from the solution.**

lab: title: 'Lab 06: Automatically provision IoT devices securely and at scale with DPS' module: 'Module 3: Device Provisioning at Scale'

---

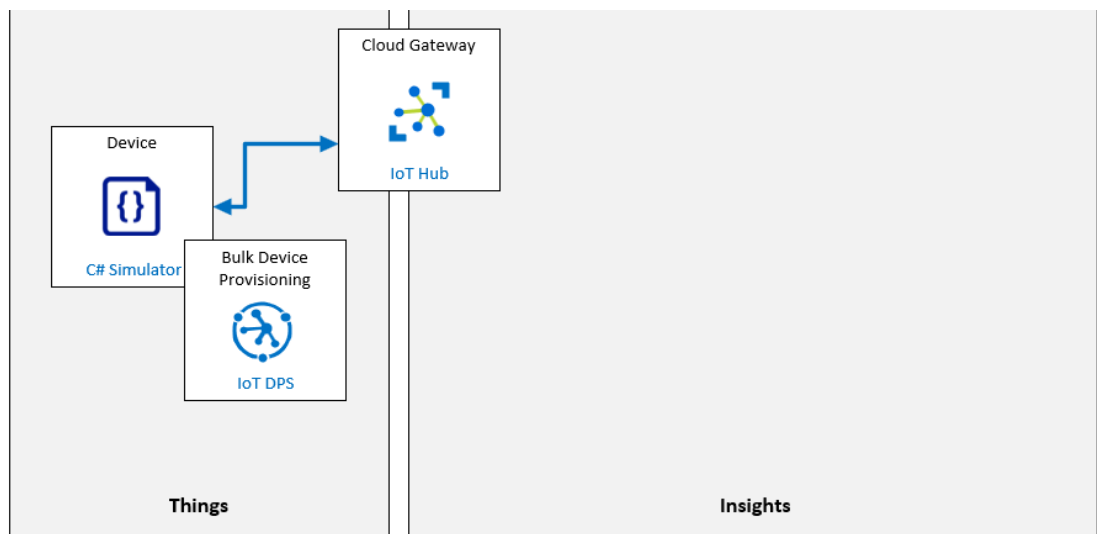
# Automatically provision IoT devices securely and at scale with DPS

## Lab Scenario

Your work to-date on Contoso's Asset Monitoring and Tracking Solution has enabled you to validate the device provisioning and deprovisioning process using an Individual Enrollment approach. The management team has now asked that you begin testing the process for a larger scale rollout.

To keep the project moving forward you need to demonstrate that the Device Provisioning Service can be used to enroll larger numbers of devices automatically and securely using X.509 certificate authentication. You will be setting up a group enrollment to verify that Contoso's requirements are met.

The following resources will be created:



## In This Lab

In this lab, you will begin by reviewing the lab prerequisites and you will run a script if needed to ensure that your Azure subscription includes the required resources. You will then generate an X.509 root CA Certificate using OpenSSL within the Azure Cloud Shell, and use the root certificate to configure the Group Enrollment within the Device Provisioning Service (DPS). After that, you will use the root certificate to generate a device certificate, which you will use within a simulated device code to provision your device to IoT hub. While in your device code, you will implement access to the device twin properties used to perform initial configuration of the device. You will then test your simulated device. To finish up this lab, you will deprovision the entire group enrollment. The lab includes the following exercises:

- Verify Lab Prerequisites
- Generate and Configure X.509 CA Certificates using OpenSSL
- Configure simulated device with X.509 certificate
- Test the Simulated Device
- Deprovision a Group Enrollment



# Lab Instructions

## Exercise 1: Verify Lab Prerequisites

This lab assumes that the following Azure resources are available:

Resource Type	Resource Name
Resource Group	rg-az220
IoT Hub	iot-az220-training-{your-id}
Device Provisioning Service	dps-az220-training-{your-id}

If these resources are not available, you will need to run the **lab06-setup.azcli** script as instructed below before moving on to Exercise 2. The script file is included in the GitHub repository that you cloned locally as part of the dev environment configuration (lab 3).

The **lab06-setup.azcli** script is written to run in a **bash** shell environment - the easiest way to execute this is in the Azure Cloud Shell.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.

If you are prompted about setting up storage for Cloud Shell, accept the defaults.

2. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

3. On the Cloud Shell toolbar, click **Upload/Download files** (fourth button from the right).
4. In the dropdown, click **Upload**.
5. In the file selection dialog, navigate to the folder location of the GitHub lab files that you downloaded when you configured your development environment.

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and

extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 06-Automatic Enrollment of Devices in DPS
  - Setup

The lab06-setup.azcli script file is located in the Setup folder for lab 6.

6. Select the **lab06-setup.azcli** file, and then click **Open**.

A notification will appear when the file upload has completed.

7. To verify that the correct file has uploaded in Azure Cloud Shell, enter the following command:

```
bash ls
```

The `ls` command lists the content of the current directory. You should see the lab06-setup.azcli file listed.

8. To create a directory for this lab that contains the setup script and then move into that directory, enter the following Bash commands:

```
bash mkdir lab6 mv lab06-setup.azcli lab6 cd lab6
```

9. To ensure the **lab06-setup.azcli** script has the execute permission, enter the following command:

```
bash chmod +x lab06-setup.azcli
```

10. On the Cloud Shell toolbar, to enable access to the lab06-setup.azcli file, click **Open Editor** (second button from the right - { }).
11. In the **Files** list, to expand the lab6 folder and open the script file, click **lab6**, and then click **lab06-setup.azcli**.

The editor will now show the contents of the **lab06-setup.azcli** file.

12. In the editor, update the values of the `{your-id}` and `{your-location}` variables.

Referencing the sample below as an example, you need to set `{your-id}` to the Unique ID you created at the start of this course - i.e. **cah191211**, and set `{your-location}` to the location that makes sense for your resources.

```
```bash
```

# !/bin/bash

## Change these values!

```
YourID="{your-id}" Location="{your-location}" ``
```

**Note:** The `{your-location}` variable should be set to the short name for the region where you are deploying all of your resources. You can see a list of the available locations and their short-names (the **Name** column) by entering this command:

```
``bash az account list-locations -o Table
```

```
DisplayName Latitude Longitude Name
```

---

```
East Asia 22.267 114.188 eastasia Southeast Asia 1.283 103.833  
southeastasia Central US 41.5908 -93.6208 centralus East US 37.3719  
-79.8164 eastus East US 2 36.6681 -78.3889 eastus2 ``
```

13. In the top-right of the editor window, to save the changes made to the file and close the editor, click **...**, and then click **Close Editor**.

If prompted to save, click **Save** and the editor will close.

**Note:** You can use **CTRL+S** to save at any time and **CTRL+Q** to close the editor.

14. To create the resources required for this lab, enter the following command:

```
bash ./lab06-setup.azcli
```

This will take a few minutes to run. You will see output as each step completes.

Once the script has completed, you will be ready to continue with the lab.

## Exercise 2: Generate and Configure X.509 CA Certificates using OpenSSL

In this exercise, you will generate an X.509 CA Certificate using OpenSSL within the Azure Cloud Shell. This certificate will be used to configure the Group Enrollment within the Device Provisioning Service (DPS).

### Task 1: Generate the certificates

1. If necessary, log in to the [Azure portal](#) using the Azure account credentials that you are using for this course.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. In the top right of the portal window, to open the Azure Cloud Shell, click **Cloud Shell**.

The Cloud Shell button has an icon that appears to represent a command prompt - >\_.

A Cloud Shell window will open near the bottom of the display screen.

3. In the upper left corner of the Cloud Shell window, ensure that **Bash** is selected as the environment option.

**Note:** Both *Bash* and *PowerShell* interfaces for the Azure Cloud Shell support the use of **OpenSSL**. In this Exercise you will be using some helper scripts written for the *Bash* shell.

4. At the Cloud Shell command prompt, to create and then move into a new directory, enter the following commands:

```
```sh
```

**ensure the current directory is the  
user's home directory**

`cd ~`

**make a directory named  
'certificates'**

```
mkdir certificates
```

## **change directory to the 'certificates' directory**

```
cd certificates ``
```

5. At the Cloud Shell command prompt, to download and prepare the Azure IoT helper scripts that you will be using, enter the following commands:

```
``sh
```



# download helper script files

```
curl https://raw.githubusercontent.com/Azure/azure-iot-sdk-  
c/master/tools/CACertificates/certGen.sh --output certGen.sh curl  
https://raw.githubusercontent.com/Azure/azure-iot-sdk-  
c/master/tools/CACertificates/openssl_device_intermediate_ca.cnf --  
output openssl_device_intermediate_ca.cnf curl  
https://raw.githubusercontent.com/Azure/azure-iot-sdk-  
c/master/tools/CACertificates/openssl_root_ca.cnf --output  
openssl_root_ca.cnf
```

# update script permissions so user can read, write, and execute it

```
chmod 700 certGen.sh ``
```

The helper script and supporting files are being downloaded from the **Azure/azure-iot-sdk-c** open source project hosted on Github, which is a component of the Azure IoT Device SDK. The **certGen.sh** helper script will provide you with a chance to see how CA Certificates are used without diving too deeply into the specifics of OpenSSL configuration (which is outside the scope of this course).

For additional instructions on using this helper script, and for instructions on how to use PowerShell instead of Bash, please see this link:

<https://github.com/Azure/azure-iot-sdk-c/blob/master/tools/CACertificates/CACertificateOverview.md>

**WARNING:** Certificates created by this helper script **MUST NOT** be used for Production. They contain hard-coded passwords ("1234"), expire after 30 days, and most importantly are provided for demonstration purposes only to help you quickly understand CA Certificates. When building products against CA Certificates, be sure to apply your company's security best practices for certificate creation and lifetime management.

If you are interested, you can quickly scan the contents of the script file that you downloaded by using the editor that's built-in to the Cloud Shell.

- In the Cloud Shell, to open the editor, click `{}`.
- In the FILES list, click **certificates**, and then click **certGen.sh**

**Note:** If you are experienced with other text file viewing tools in the Bash environment, such as the `more` or `vi` commands, you could also use those tools.

Next, you will use the script to create your root and intermediate certificates.

6. To generate the root and intermediate certificates, enter the following command:

```
sh ./certGen.sh create_root_and_intermediate
```

Notice that you ran the script with the `create_root_and_intermediate` option. This command assumes that you are running the script from within the `~/certificates` directory.

This command generated a Root CA Certificate named `azure-iot-test-only.root.ca.cert.pem` and placed it within a `./certs` directory (under the certificates directory that you created).

7. To download the root certificate to your local machine (so it can be uploaded to DPS), enter the following command:

```
sh download ~/certificates/certs/azure-iot-test-only.root.ca.cert.pem
```

You will be prompted to save the file to your local machine. Make a note of where the file is being saved, you will need it in the next task.

## Task 2: Configure DPS to trust the root certificate

1. In the Azure portal, open your Device Provisioning Service.

The Device Provisioning Service is accessible from the Resources tile on your dashboard by clicking `dps-az220-training-{your-id}`.

2. On the left-side menu of the **dps-az220-training-{your-id}** blade, under **Settings**, click **Certificates**.
3. On the **Certificates** pane, at the top of the pane, click **+ Add**.

Clicking **+ Add** will start the process of uploading the X.509 CA Certificate to the DPS service.

4. On the **Add Certificate** blade, under **Certificate Name**, enter **root-ca-cert**

It is important to provide a name that enables you to differentiate between certificates, such as your root certificate and intermediate certificate, or multiple certificates at the same hierarchy level within the chain.

**Note:** The root certificate name that you entered could be the same as the name of the certificate file, or something different. The name that you provided is a logical name that has no correlation to the *Common Name* that is embedded within the contents X.509 CA Certificate.

5. Under **Certificate .pem or .cer file.**, to the right of the text box, click **Open**.

Clicking the **Open** button to the right of the text field will open an Open file dialog that enables you to navigate to the `azure-iot-test-only.root.ca.cert.pem` CA Certificate that you downloaded earlier.

6. Navigate to the folder location where you downloaded the root CA Certificate file, click **azure-iot-test-only.root.ca.cert.pem**, and then click **Open**.

7. At the bottom of the **Add Certificate** blade, click **Save**.

Once the X.509 CA Certificate has been uploaded, the Certificates pane will display the certificate with the **Status** value set to **Unverified**. Before this CA Certificate can be used to authenticate devices to DPS, you will need to verify *Proof of Possession* of the certificate.

8. To start the process of verifying Proof of Possession of the certificate, click **root-ca-cert**.

9. At the bottom of the **Certificate Details** pane, click **Generate Verification Code**.

You may need to scroll down to see the **Generate Verification Code** button.

When you click the button it will place the generated code into the Verification Code field (located just above the button).

10. To the right of **Verification Code**, click **Copy to clipboard**.

Proof of Possession of the CA certificate is provided to DPS by uploading a certificate generated from the CA certificate with the verification code that was just generated within DPS. This is how you provide proof that you actually own the CA Certificate.

**IMPORTANT:** You will need to leave the **Certificate Details** pane open while you generate the verification certificate. If you close the pane, you will invalidate the verification code, and will need to generate a new one.

11. Open the **Azure Cloud Shell**, if it's not still open from earlier, and navigate to the `~/certificates` directory.

12. To create the verification certificate, enter the following command:

```
sh ./certGen.sh create_verification_certificate  
<verification-code>
```

Be sure to replace the `<verification-code>` placeholder with the **Verification Code** generated by the Azure portal.

For example, the command that you run will look similar to the following:

```
sh ./certGen.sh create_verification_certificate  
49C900C30C78D916C46AE9D9C124E9CFFD5FCE124696FAEA
```

This generates a *verification certificate* that is chained to the CA certificate. The subject of the certificate is the verification code. The generated Verification Certificate named `verification-code.cert.pem` is located within the `./certs` directory of the Azure Cloud Shell.

The next step is to download the verification certificate to your local machine (similar to what you did with the root certificate earlier), so that you can then upload it to DPS.

13. To download the verification certificate to your local machine, enter the following command:

```
sh download ~/certificates/certs/verification-code.cert.pem
```

**Note:** Depending on the web browser, you may be prompted to allow multiple downloads at this point. If there appears to be no response to your download command, make sure there's not a prompt elsewhere on the screen asking for permission to allow the download.

14. Switch back to the **Certificate Details** pane.

This pane of your DPS service should still be open in the Azure portal.

15. At the bottom the **Certificate Details** pane, below and to the right of **Verification Certificate .pem or .cer file.**, click **Open**.
16. In the Open file dialog, navigate to your downloads folder, click **verification-code.cert.pem**, and then click **Open**.
17. At the bottom the **Certificate Details** pane, click **Verify**.
18. On the **Certificates** pane, ensure that the **Status** for the certificate is now set to **Verified**.

You may need to click **Refresh** at the top of the pane (to the right of the **Add** button) to see this change.

### Task 3: Create Group Enrollment (X.509 Certificate) in DPS

In this task, you will create a new enrollment group within the Device Provisioning Service (DPS) that uses X.509 certificate attestation.

1. On the left-side menu of the **dps-az220-training-{your-id}** blade, under **Settings**, click **Manage enrollments**.

2. At the top of the **Manage enrollments** pane, click **Add enrollment group**.

Recall that an enrollment group is basically a record of the devices that may register through auto-provisioning.

3. On the **Add Enrollment Group** blade, under **Group name**, enter **eg-test-simulated-devices**
4. Ensure that the **Attestation Type** is set to **Certificate**.
5. Ensure that the **Certificate Type** field is set to **CA Certificate**.
6. In the **Primary Certificate** dropdown, select the CA certificate that was uploaded to DPS previously, likely **root-ca-cert**.
7. Leave the **Secondary Certificate** dropdown set to **No certificate selected**.

The secondary certificate is generally used for certificate rotation, to accommodate expiring certificates or certificates that have been compromised. You can find more information on rolling certificates here: <https://docs.microsoft.com/en-us/azure/iot-dps/how-to-roll-certificates>

8. Leave the **Select how you want to assign devices to hubs** field set to **Evenly weighted distribution**.

In large environments where you have multiple distributed hubs, this setting will control how to choose what IoT Hub should receive this device enrollment. You will have a single IoT Hub associated with the enrollment in this lab, so how you assign devices to IoT hubs doesn't really apply within this lab scenario.

9. Notice that the **Select the IoT hubs this group can be assigned to** dropdown has your **iot-az220-training-{your-id}** IoT Hub selected.

This field is used to ensure that when the device is provisioned, it gets added to the correct IoT Hub.

10. Leave the **Select how you want device data to be handled on re-provisioning** field set to **Re-provision and migrate data**.

This field gives you high-level control over the re-provisioning behavior, where the same device (as indicated through the same Registration ID) submits a later provisioning request after already being provisioned successfully at least once.

11. In the **Initial Device Twin State** field, modify the JSON object as follows:

```
json { "tags": {}, "properties": { "desired": {  
  "telemetryDelay": "1" } } }
```

This JSON data represents the initial configuration of device twin desired properties for any device that participates in this enrollment group.

The devices will use the `properties.desired.telemetryDelay` property to set the time delay for reading and sending telemetry to IoT Hub.

12. Leave **Enable entry** set to **Enable**.

Generally, you'll want to enable new enrollment entries and keep them enabled.

13. At the top of the **Add Enrollment Group** blade, click **Save**.

### Exercise 3: Configure simulated device with X.509 certificate

In this exercise, you will generate a device certificate using the root certificate, and configure a simulated device that connects by using the device certificate for attestation.

#### Task 1: Generate a device certificate

1. If necessary, log in to your Azure portal using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. On the Azure portal toolbar, click **Cloud Shell**

The Azure portal toolbar runs across the top of the portal window. The Cloud Shell button is the 6th in from the right.

3. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

4. At the Cloud Shell command prompt, to navigate to the `~/certificates` directory, enter the following command:

```
sh cd ~/certificates
```

The `~/certificates` directory is where the `certGen.sh` helper scripts were downloaded. You used them to generate the CA Certificate for DPS earlier in this lab. This helper script will also be used to generate a device certificate within the CA Certificate chain.

5. To generate an X.509 device certificate within the CA certificate chain, enter the following command:

```
sh ./certGen.sh create_device_certificate sensor-th1-2000
```

This command will create a new X.509 certificate that's signed by the CA certificate that was generated previously. Notice that the device id (`sensor-th1-2000`) is passed to the `create_device_certificate` command of the `certGen.sh` script. This device id will be set within the *common name*, or `CN=`, value of the device certificate. This certificate will generate a leaf device X.509 certificate for your simulated device, and will be used to authenticate the device with the Device Provisioning Service (DPS).

Once the `create_device_certificate` command has completed, the generated X.509 device certificate will be named `new-device.cert.pfx`, and will be located within the `/certs` sub-directory.

**Note:** This command overwrites any existing device certificate in the `/certs` sub-directory. If you want to create a certificate for multiple devices, ensure that you save a copy of the `new-device.cert.pfx` each time you run the command.

6. To rename the device certificate that you just created, enter the following commands:



```
sh mv ~/certificates/certs/new-device.cert.pfx  
~/certificates/certs/sensor-thl-2000-device.cert.pfx mv  
~/certificates/certs/new-device.cert.pem  
~/certificates/certs/sensor-thl-2000-device.cert.pem
```

7. To create four additional device certificates, enter the following commands:

```
``sh ./certGen.sh create_device_certificate sensor-thl-2001 mv  
~/certificates/certs/new-device.cert.pfx ~/certificates/certs/sensor-thl-  
2001-device.cert.pfx mv ~/certificates/certs/new-device.cert.pem  
~/certificates/certs/sensor-thl-2001-device.cert.pem
```

```
./certGen.sh create_device_certificate sensor-thl-2002 mv  
~/certificates/certs/new-device.cert.pfx ~/certificates/certs/sensor-thl-  
2002-device.cert.pfx mv ~/certificates/certs/new-device.cert.pem  
~/certificates/certs/sensor-thl-2002-device.cert.pem
```

```
./certGen.sh create_device_certificate sensor-thl-2003 mv  
~/certificates/certs/new-device.cert.pfx ~/certificates/certs/sensor-thl-  
2003-device.cert.pfx mv ~/certificates/certs/new-device.cert.pem  
~/certificates/certs/sensor-thl-2003-device.cert.pem
```

```
./certGen.sh create_device_certificate sensor-thl-2004 mv  
~/certificates/certs/new-device.cert.pfx ~/certificates/certs/sensor-thl-  
2004-device.cert.pfx mv ~/certificates/certs/new-device.cert.pem  
~/certificates/certs/sensor-thl-2004-device.cert.pem ``
```

8. To download the generated X.509 device certificates from the Cloud Shell to your local machine, enter the following commands:

```
sh download ~/certificates/certs/sensor-thl-2000-  
device.cert.pfx download ~/certificates/certs/sensor-thl-  
2001-device.cert.pfx download ~/certificates/certs/sensor-  
thl-2002-device.cert.pfx download  
~/certificates/certs/sensor-thl-2003-device.cert.pfx download  
~/certificates/certs/sensor-thl-2004-device.cert.pfx
```

In the next task, you will start building the simulated devices that will use the X.509 device certificates to authenticate with the Device Provisioning Service.

## Task 2: Configure a simulated device

In this task, you will complete the following:

- Get the ID Scope from DPS that will be placed in code

- Copy the downloaded device certificate into the root folder of the application
- Configure the application in Visual Studio Code
- In the Azure portal, open your Device Provisioning Service blade and ensure that the **Overview** pane is selected.
- On the **Overview** pane, copy the **ID Scope** for the Device Provisioning Service, and save it for later reference.

There is a copy button to the right of the value that will appear when you hover over the value.

The **ID Scope** will be similar to this value: `0ne0004E52G`

- Open Windows File Explorer, and then navigate to the folder where the `sensor-th1-2000-device.cert.pfx` certificate file was downloaded.
- Use File Explorer to create a copy of the 5 device certificate files.

It will save some time to copy all five certificate files now, but you will only be using the first one, `sensor-th1-2000-device.cert.pfx`, in the code project that you build initially.

- In File Explorer, navigate to the Starter folder for lab 6 (Automatic Enrollment of Devices in DPS).

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 06-Automatic Enrollment of Devices in DPS
    - Starter
    - ContainerDevice
- With the ContainerDevice folder open, paste-in the copied device certificate files.

The root directory of the ContainerDevice folder includes the `Program.cs` file for your simulated device app. The simulated device app will use the device certificate file when authenticating to the Device Provisioning Service.

- Open **Visual Studio Code**.
- On the **File** menu, click **Open Folder**
- In the **Open Folder** dialog, navigate to the Starter folder for lab 6 (Automatic Enrollment of Devices in DPS).
- Click **ContainerDevice**, and then click **Select Folder**.

You should see the following files listed in the EXPLORER pane of Visual Studio Code:

- ContainerDevice.csproj
- Program.cs
- sensor-thl-2000-device.cert.pfx

**Note:** The other device certificate files that you copied to this folder will be used later in this lab, but for now you will focus on implementing just the first device certificate file.

- In the **EXPLORER** pane, to open the ContainerDevice.csproj file, click **ContainerDevice.csproj**.
- In the code editor pane, within the <ItemGroup> tag, update the certificate file name as follows:

```
xml <ItemGroup> <None Update="sensor-thl-2000-
device.cert.pfx" CopyToOutputDirectory="PreserveNewest" />
<PackageReference Include="Microsoft.Azure.Devices.Client"
Version="1.*" /> <PackageReference
Include="Microsoft.Azure.Devices.Provisioning.Transport.Mqtt"
Version="1.*" /> <PackageReference
Include="Microsoft.Azure.Devices.Provisioning.Transport.Amqp"
Version="1.*" /> <PackageReference
Include="Microsoft.Azure.Devices.Provisioning.Transport.Http"
Version="1.*" /> </ItemGroup>
```

This configuration ensures that the sensor-thl-2000-device.cert.pfx certificate file is copied to the build folder when the C# code is compiled, and made available for the program to access when it executes.

- On the Visual Studio Code **File** menu, click **Save**.

**Note:** If you are prompted by Visual Studio Code to Restore, do that now.

- In the **EXPLORER** pane, click **Program.cs**.

A cursory glance will reveal that this version of the **ContainerDevice** application is virtually identical to the version used in the preceding lab. The only changes will be those that relate specifically to the use of X.509 certificates as an attestation mechanism. From the application perspective, it matters little that this device will be connecting via a Group Enrollment vs an Individual Enrollment.

- Locate the **GlobalDeviceEndpoint** variable, and notice that its value is set to the Global Device Endpoint for the Azure Device Provisioning Service (`global.azure-devices-provisioning.net`).

Within the Public Azure Cloud, **global.azure-devices-provisioning.net** is the Global Device Endpoint for the Device Provisioning Service (DPS). All devices connecting to Azure DPS will be configured with this Global Device Endpoint DNS name. You should see code that is similar to the following:

```
csharp private const string GlobalDeviceEndpoint =  
"global.azure-devices-provisioning.net";
```

- Locate the **dpsIdScope** variable, and update the assigned value using the **ID Scope** that you copied from the Overview pane of the Device Provisioning Service.

When you have updated your code, it should look similar to the following:

```
csharp private static string dpsIdScope = "0ne000CBD6C";
```

- Locate the **certificateFileName** variable, and notice that its value is set to the default name of the device certificate file that you generated (**new-device.cert.pfx**).

Rather than using symmetric keys as in the earlier lab, this time the application is using an X.509 certificate. The **new-device.cert.pfx** file is the X.509 device certificate file that you generated using the **certGen.sh** helper script within the Cloud Shell. This variable tells the device code which file contains the X.509 device certificate that it will use when authenticating with the Device Provisioning Service.

- Update the value assigned to the **certificateFileName** variable as follows:

```
csharp private static string certificateFileName = "sensor-  
th1-2000-device.cert.pfx";
```

- Locate the **certificatePassword** variable, and notice that its value is set to the default password defined by the **certGen.sh** script.

The **certificatePassword** variable contains the password for the X.509 device certificate. It's set to 1234, as this is the default password used by the **certGen.sh** helper script when generating the X.509 certificates.

**Note:** For the purpose of this lab, the password is hard coded. In a *production* scenario, the password will need to be stored in a more secure manner, such as in an Azure Key Vault. Additionally, the certificate file (PFX) should be stored securely on a production device using a Hardware Security Module (HSM).

An HSM (Hardware Security Module), is used for secure, hardware-based storage of device secrets, and is the most secure form of secret storage. Both X.509 certificates and SAS tokens can be stored in the HSM. HSMs can be used with all attestation mechanisms the provisioning service supports. HMS will be discussed in more detail later in this course.

### Task 3: Add the provisioning code

In this task, you will enter code that completes the implementation associated with the Main method, device provisioning, device twin properties.

1. In the code editor pane for the Program.cs file, locate the `// INSERT Main method below here` comment.

2. To implement the Main method, enter the following code:

```
```csharp public static async Task Main(string[] args) { X509Certificate2
certificate = LoadProvisioningCertificate();

using (var security = new
SecurityProviderX509Certificate(certificate))
using (var transport = new
ProvisioningTransportHandlerAmqp(TransportFallbackType.TcpOnly))
{
    ProvisioningDeviceClient provClient =
        ProvisioningDeviceClient.Create(GlobalDeviceEndpoint,
dpsIdScope, security, transport);

    using (deviceClient = await ProvisionDevice(provClient,
security))
    {
        await deviceClient.OpenAsync().ConfigureAwait(false);

        // INSERT Setup OnDesiredPropertyChanged Event
Handling below here
    }
}
```

```

        // INSERT Load Device Twin Properties below here

        // Start reading and sending device telemetry
        Console.WriteLine("Start reading and sending device
telemetry...");
        await SendDeviceToCloudMessagesAsync();

        await
deviceClient.CloseAsync().ConfigureAwait(false);
    }
}

} ``

```

This Main method is very similar to that used in the earlier lab. The two significant changes are the need to load the X.509 certificate and then the change to using **SecurityProviderX509Certificate** as the security provider. The remaining code is identical - you should note that the device twin property change code is also present.

3. Locate the `// INSERT LoadProvisioningCertificate` method below here comment.

4. To implement the `LoadProvisioningCertificate` method, insert the following code:

```

``csharp private static X509Certificate2 LoadProvisioningCertificate() {
var certificateCollection = new X509Certificate2Collection();
certificateCollection.Import(certificateFileName, certificatePassword,
X509KeyStorageFlags.UserKeySet);

X509Certificate2 certificate = null;

foreach (X509Certificate2 element in certificateCollection)
{
    Console.WriteLine($"Found certificate:
{element?.Thumbprint} {element?.Subject}; PrivateKey:
{element?.HasPrivateKey}");
    if (certificate == null && element.HasPrivateKey)
    {
        certificate = element;
    }
    else
    {
        element.Dispose();
    }
}

if (certificate == null)
{
    throw new FileNotFoundException($"{certificateFileName}

```

```

did not contain any certificate with a private key.");
}

Console.WriteLine($"Using certificate
{certificate.Thumbprint} {certificate.Subject}");
return certificate;

} ``

```

As you might expect from the name, the purpose of this method is to load the X.509 certificate from disk. Should the load succeed, the method returns an instance of the **X509Certificate2** class.

**Information:** You may be curious as to why the result is an **X509Certificate2** type rather than an **X509Certificate**. The **X509Certificate** is an earlier implementation and is limited in its functionality. The **X509Certificate2** is a subclass of **X509Certificate** with additional functionality that supports both V2 and V3 of the X509 standard.

The method creates an instance of the **X509Certificate2Collection** class and then attempts to import the certificate file from disk, using the the hard-coded password. The **X509KeyStorageFlags.UserKeySet** values specifies that private keys are stored in the current user store rather than the local computer store. This occurs even if the certificate specifies that the keys should go in the local computer store.

Next, the method iterates through the imported certificates (in this case, there should only be one) and verifies that the certificate has a private key. Should the imported certificate not match this criteria, an exception is thrown, otherwise the method returns the imported certificate.

5. Locate the `// INSERT ProvisionDevice method below here` comment.

6. To implement the `ProvisionDevice` method, enter the following code:

```

``csharp private static async Task
ProvisionDevice(ProvisioningDeviceClient provisioningDeviceClient,
SecurityProviderX509Certificate security) { var result = await
provisioningDeviceClient.RegisterAsync().ConfigureAwait(false);
Console.WriteLine($"ProvisioningClient AssignedHub:
{result.AssignedHub}; DeviceID: {result.DeviceId}"); if (result.Status !=
ProvisioningRegistrationStatusType.Assigned) { throw new
Exception($"DeviceRegistrationResult.Status is NOT 'Assigned'"); }

var auth = new DeviceAuthenticationWithX509Certificate(
    result.DeviceId,

```

```

        security.GetAuthenticationCertificate());

return DeviceClient.Create(result.AssignedHub, auth,
    TransportType.Amqp);

}

```

This version of **ProvisionDevice** is very similar to that you used in an earlier lab. The primary change is that the **security** parameter is now of type **SecurityProviderX509Certificate**. This means that the **auth** variable used to create a **DeviceClient** must now be of type **DeviceAuthenticationWithX509Certificate** and uses the `security.GetAuthenticationCertificate()` value. The actual device registration is the same as before.

#### Task 4: Add the Device Twin Integration Code

To use the device twin properties (from Azure IoT Hub) on a device, you need to create the code that accesses and applies the device twin properties. In this case, you wish to update the simulated device code to read a device twin Desired Property, and then assign that value to the **telemetryDelay** variable. You also want to update the device twin Reported Property to indicate the delay value that is currently implemented on the device.

1. In the Visual Studio Code editor, locate the **Main** method.
2. Take a moment to review the code, and then locate the `// INSERT Setup OnDesiredPropertyChanged Event Handling below here` comment.

To begin the integration of device twin properties, you need code that enables the simulated device to be notified when a device twin property is updated.

To achieve this, you can use the **DeviceClient.SetDesiredPropertyUpdateCallbackAsync** method, and set up an event handler by creating an **OnDesiredPropertyChanged** method.

3. To set up the DeviceClient for an OnDesiredPropertyChanged event, enter the following code:

```

csharp await
deviceClient.SetDesiredPropertyUpdateCallbackAsync (OnDesiredP
ropertyChanged, null).ConfigureAwait(false);

```

The **SetDesiredPropertyUpdateCallbackAsync** method is used to set up the **DesiredPropertyUpdateCallback** event handler to receive device



twin desired property changes. This code configures **deviceClient** to call a method named **OnDesiredPropertyChanged** when a device twin property change event is received.

Now that the **SetDesiredPropertyUpdateCallbackAsync** method is in place to set up the event handler, you need to create the **OnDesiredPropertyChanged** method that it calls.

4. Locate the `// INSERT OnDesiredPropertyChanged method below here` comment.

5. To complete the setup of the event handler, enter the following code:

```
```csharp private static async Task
OnDesiredPropertyChanged(TwinCollection desiredProperties, object
userContext) { Console.WriteLine("Desired Twin Property Changed:");
Console.WriteLine($"{desiredProperties.ToJson()}");

// Read the desired Twin Properties
if (desiredProperties.Contains("telemetryDelay"))
{
    string desiredTelemetryDelay =
desiredProperties["telemetryDelay"];
    if (desiredTelemetryDelay != null)
    {
        telemetryDelay = int.Parse(desiredTelemetryDelay);
    }
    // if desired telemetryDelay is null or unspecified,
don't change it
}

// Report Twin Properties
var reportedProperties = new TwinCollection();
reportedProperties["telemetryDelay"] =
telemetryDelay.ToString();
await
deviceClient.UpdateReportedPropertiesAsync(reportedProperties
).ConfigureAwait(false);
Console.WriteLine("Reported Twin Properties:");
Console.WriteLine($"{reportedProperties.ToJson()}");

} ```
```

Notice that the **OnDesiredPropertyChanged** event handler accepts a **desiredProperties** parameter of type **TwinCollection**.

Notice that if the value of the **desiredProperties** parameter contains **telemetryDelay** (a device twin desired property), the code will assign the value of the device twin property to the **telemetryDelay** variable. You may recall that the **SendDeviceToCloudMessagesAsync** method include

a **Task.Delay** call that uses the **telemetryDelay** variable to set the delay time between messages sent to IoT hub.

Notice the next block of code is used to report the current state of the device back up to Azure IoT Hub. This code calls the **DeviceClient.UpdateReportedPropertiesAsync** method and passes it a **TwinCollection** that contains the current state of the device properties. This is how the device reports back to IoT Hub that it received the device twin desired properties changed event, and has now updated its configuration accordingly. Note that it reports what the properties are now set to, not an echo of the desired properties. In the case where the reported properties sent from the device are different than the desired state that the device received, IoT Hub will maintain an accurate Device Twin that reflects the state of the device.

Now that the device can receive updates to the device twin desired properties from Azure IoT Hub, it also needs to be coded to configure its initial setup when the device starts up. To do this the device will need to load the current device twin desired properties from Azure IoT Hub, and configure itself accordingly.

6. In the **Main** method, locate the `// INSERT Load Device Twin Properties below here` comment.
7. To read the device twin desired properties and configure the device to match on device startup, enter the following code:

```
csharp var twin = await
deviceClient.GetTwinAsync().ConfigureAwait(false); await
OnDesiredPropertyChanged(twin.Properties.Desired, null);
```

This code calls the **DeviceTwin.GetTwinAsync** method to retrieve the device twin for the simulated device. It then accesses the **Properties.Desired** property object to retrieve the current Desired State for the device, and passes that to the **OnDesiredPropertyChanged** method that will configure the simulated devices **telemetryDelay** variable.

Notice, this code reuses the **OnDesiredPropertyChanged** method that was already created for handling *OnDesiredPropertyChanged* events. This helps keep the code that reads the device twin desired state properties and configures the device at startup in a single place. The resulting code is simpler and easier to maintain.

8. On the Visual Studio Code **File** menu, click **Save**.

Your simulated device will now use the device twin properties from Azure IoT Hub to set the delay between telemetry messages.

9. On the **Terminal** menu, click **New Terminal**.
10. At the Terminal command prompt, to verify that your code will build correctly, enter **dotnet build**

If you see build error listed, fix them now before continuing to the next exercise. Work with your instructor if needed.

## **Exercise 4: Create Additional Instances of your Simulated Device**

In this exercise, you will make copies of your simulated device project, and then update your code to use the different device certificates that you created and added to the project folder.

### **Task 1: Make copies of your code project**

1. Open Windows File Explorer.
2. In File Explorer, navigate to the Starter folder for lab 6 (Automatic Enrollment of Devices in DPS).

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 06-Automatic Enrollment of Devices in DPS
    - Starter

3. Right-click **ContainerDevice**, and then click **Copy**.

The ContainerDevice folder should be the folder containing your simulated device code.

4. Right-click in the empty space below **ContainerDevice**, and then click **Paste**

You should see that a folder named "ContainerDevice - Copy" has been created.

5. Right-click **ContainerDevice** - **Copy**, click **Rename**, and then type **ContainerDevice2001**
6. Repeat steps 3-5 to create folders with the following names:
  - **ContainerDevice2002**
  - **ContainerDevice2003**
  - **ContainerDevice2004**

## **Task 2: Update the certificate file references in your code project**

1. If necessary, open Visual Studio Code.
2. On the **File** menu, click **Open Folder**.
3. Navigate to the lab 6 Starter folder.
4. Click **ContainerDevice2001**, and then click **Select Folder**.
5. In the EXPLORER pane, click **Program.cs**.
6. In the code editor, locate the **certificateFileName** variable, and then update the value assigned to the **certificateFileName** variable as follows:

```
csharp private static string certificateFileName = "sensor-  
thl-2001-device.cert.pfx";
```

7. In the **EXPLORER** pane, to open the ContainerDevice.csproj file, click **ContainerDevice.csproj**.
8. In the code editor pane, within the <ItemGroup> tag, update the certificate file name as follows:

```
xml <ItemGroup> <None Update="sensor-thl-2001-  
device.cert.pfx" CopyToOutputDirectory="PreserveNewest" />  
<PackageReference Include="Microsoft.Azure.Devices.Client"  
Version="1.*" /> <PackageReference  
Include="Microsoft.Azure.Devices.Provisioning.Transport.Mqtt"  
Version="1.*" /> <PackageReference  
Include="Microsoft.Azure.Devices.Provisioning.Transport.Amqp"  
Version="1.*" /> <PackageReference  
Include="Microsoft.Azure.Devices.Provisioning.Transport.Http"  
Version="1.*" /> </ItemGroup>
```

9. On the **File** menu, click **Save All**.
10. Repeat steps 3-9 above to update **Program.cs** and **ContainerDevice.csproj** files for each of the remaining code projects as

follows:

<b>Project Folder</b>	<b>Certificate Name</b>
ContainerDevice2002	sensor-thl-2002-device.cert.pfx
ContainerDevice2003	sensor-thl-2003-device.cert.pfx
ContainerDevice2004	sensor-thl-2004-device.cert.pfx

**Note:** Be sure to **Save All** each time before continuing to the next folder.

## **Exercise 5: Test the Simulated Device**

In this exercise, you will run the simulated device. When the device is started for the first time, it will connect to the Device Provisioning Service (DPS) and automatically be enrolled using the configured group enrollment. Once enrolled into the DPS group enrollment, the device will be automatically registered within the Azure IoT Hub device registry. Once enrolled and registered, the device will begin communicating with Azure IoT Hub securely using the configured X.509 certificate authentication.

### **Task 1: Build and run the simulated device projects**

1. Ensure that you have Visual Studio Code open.
2. On the **File** menu, click **Open Folder**.
3. Navigate to the lab 6 Starter folder.
4. Click **ContainerDevice**, and then click **Select Folder**.
5. On the **View** menu, click **Terminal**.

This will open the integrated Terminal at the bottom of the Visual Studio Code window.

6. At the Terminal command prompt, ensure that the current directory path is set to the `\ContainerDevice` folder.

You should see something similar to the following:

```
Allfiles\Labs\06-Automatic Enrollment of Devices in  
DPS\Starter\ContainerDevice>
```

7. To build and run the **ContainerDevice** project, enter the following command:

```
cmd/sh dotnet run
```

**Note:** When you run your simulated device for the first time, the most common error is an *Invalid certificate* error. If a `ProvisioningTransportException` exception is displayed, it is most likely due to this error. If you see a message similar to the one shown below, you will need to ensure that the CA Certificate in DPS, and the Device Certificate for the simulated device application are configured correctly before you can continue.

```
text localmachine:LabFiles User$ dotnet run Found
certificate: AFF851ED016CA5AEB71E5749BCBE3415F8CF4F37
CN=sensor-thl-2000; PrivateKey: True Using certificate
AFF851ED016CA5AEB71E5749BCBE3415F8CF4F37 CN=sensor-thl-
2000 RegistrationID = sensor-thl-2000 ProvisioningClient
RegisterAsync . . . Unhandled exception.
Microsoft.Azure.Devices.Provisioning.Client.ProvisioningT
ransportException:
{"errorCode":401002,"trackingId":"2e298c80-0974-493c-
9fd9-6253fb055ade","message":"Invalid
certificate.","timestampUtc":"2019-12-
13T14:55:40.2764134Z"} at
Microsoft.Azure.Devices.Provisioning.Client.Transport.Pro
visioningTransportHandlerAmqp.ValidateOutcome(Outcome
outcome) at
Microsoft.Azure.Devices.Provisioning.Client.Transport.Pro
visioningTransportHandlerAmqp.RegisterDeviceAsync(AmqpCli
entConnection client, String correlationId,
DeviceRegistration deviceRegistration) at
Microsoft.Azure.Devices.Provisioning.Client.Transport.Pro
visioningTransportHandlerAmqp.RegisterAsync(ProvisioningT
ransportRegisterMessage message, CancellationToken
cancellation_token) at
X509CertificateContainerDevice.ProvisioningDeviceLogic.Ru
nAsync() in /Users/User/Documents/AZ-
220/LabFiles/Program.cs:line 121 at
X509CertificateContainerDevice.Program.Main(String[]
args) in /Users/User/Documents/AZ-
220/LabFiles/Program.cs:line 55 ...
```

8. Notice that the simulated device app sends output to the Terminal window.

When the simulated device application is running correctly, the **Terminal** will display the Console output from the app.

Scroll up to the top of the information displayed in the Terminal window.

Notice the X.509 certificate was loaded, the device was registered with the Device Provisioning Service, it was assigned to connect to the **iot-**

**az220-training-{your-id}** IoT Hub, and the device twin desired properties are loaded.

```
text localmachine:LabFiles User$ dotnet run Found
certificate: AFF851ED016CA5AEB71E5749BCBE3415F8CF4F37
CN=sensor-thl-2000; PrivateKey: True Using certificate
AFF851ED016CA5AEB71E5749BCBE3415F8CF4F37 CN=sensor-thl-2000
RegistrationID = sensor-thl-2000 ProvisioningClient
RegisterAsync . . . Device Registration Status: Assigned
ProvisioningClient AssignedHub: iot-az220-training-
CP1119.azure-devices.net; DeviceID: sensor-thl-2000 Creating
X509 DeviceClient authentication. simulated device. Ctrl-C to
exit. DeviceClient OpenAsync. Connecting
SetDesiredPropertyUpdateCallbackAsync event handler...
Loading Device Twin Properties... Desired Twin Property
Changed: {"$version":1} Reported Twin Properties:
{"telemetryDelay":1} Start reading and sending device
telemetry...
```

To review the source code for the simulated device, open the `Program.cs` source code file. Look for several `Console.WriteLine` statements that are used to output the messages seen to the console.

9. Notice that JSON formatted telemetry messages are being sent to Azure IoT Hub.

```
text Start reading and sending device telemetry... 12/9/2019
5:47:00 PM > Sending message:
{"temperature":24.047539159212047,"humidity":67.0050416267500
4,"pressure":1018.8478924248358,"latitude":40.129349260196875
,"longitude":-98.42877188146265} 12/9/2019 5:47:01 PM >
Sending message:
{"temperature":26.628804161040485,"humidity":68.0961079467535
5,"pressure":1014.6454375411363,"latitude":40.093269544242695
,"longitude":-98.22227128174003}
```

Once the simulated device has passed through the initial start up, it will begin sending simulated sensor telemetry messages to Azure IoT Hub.

Notice that the delay, as defined by the `telemetryDelay` Device Twin Property, between each message sent to IoT Hub is currently delaying **1 second** between sending sensor telemetry messages.

10. Leave the simulated device running.

## Task 2: Start the other simulated devices

1. Open a new instance of Visual Studio Code.

You can do this from the Windows 10 Start menu as follows: On the Windows 10 **Start** menu, right-click **Visual Studio Code**, and then click **New Window**.

2. In the new Visual Studio Code window, on the **File** menu, click **Open Folder**.
3. Navigate to the lab 6 Starter folder.
4. Click **ContainerDevice2001**, and then click **Select Folder**.
5. On the **View** menu, click **Terminal**.

This will open the integrated Terminal at the bottom of the Visual Studio Code window.

6. At the Terminal command prompt, ensure that the current directory path is set to the `\ContainerDevice2001` folder.

You should see something similar to the following:

```
Allfiles\Labs\06-Automatic Enrollment of Devices in  
DPS\Starter\ContainerDevice2001>
```

7. To build and run the **ContainerDevice** project, enter the following command:

```
cmd/sh dotnet run
```

8. Repeat steps 1-7 above to open and start the other simulated device projects as follows:

#### **Project Folder**

ContainerDevice2002

ContainerDevice2003

ContainerDevice2004

### **Task 3: Change the device configuration through its twin**

With the simulated devices running, the `telemetryDelay` configuration can be updated by editing the device twin Desired State within Azure IoT Hub.

1. Open the **Azure portal**, and then navigate to your **Azure IoT Hub** service.



2. On the left-side menu of your IoT hub blade, under **Explorers**, click **IoT devices**.
3. Within the list of IoT devices, click **sensor-thl-2000**.

**IMPORTANT:** Make sure you select the device from this lab. You may also see a device named *sensor-th-0001* that was created earlier in the course.

4. On the **sensor-thl-2000** device blade, at the top of the blade, click **Device Twin**.

On the **Device twin** blade, there is an editor with the full JSON for the device twin. This enables you to view and/or edit the device twin state directly within the Azure portal.

5. Locate the `properties.desired` node within the Device Twin JSON.
6. Update the `telemetryDelay` property to have the value of "2".

This will update the `telemetryDelay` of the simulated device to send sensor telemetry every **2 seconds**.

The resulting JSON for this section of the device twin desired properties will look similar to the following:

```
json "properties": { "desired": { "telemetryDelay": "2",
"$metadata": { "$lastUpdated": "2019-12-
09T22:48:05.9703541Z", "$lastUpdatedVersion": 2,
"telemetryDelay": { "$lastUpdated": "2019-12-
09T22:48:05.9703541Z", "$lastUpdatedVersion": 2 } },
"$version": 2 },
```

Leave the `$metadata` and `$version` value of the `properties.desired` node within the JSON. You should only update the `telemetryDelay` value to set the new device twin desired property value.

7. At the top of the blade, to apply the device twin desired properties for the device, click **Save**.

Once saved, the updated device twin desired properties will automatically be sent to the simulated device.

8. Switch back to the **Visual Studio Code** window that contains the original **ContainerDevice** project.

9. Notice that the application has been notified of the updated device twin `telemetryDelay` desired property setting.

The application outputs messages to the Console that show that the new device twin desired properties have been loaded, and the changes have been set and reported back to the Azure IoT Hub.

```
text Desired Twin Property Changed:
{"telemetryDelay":2,"$version":2} Reported Twin Properties:
{"telemetryDelay":2}
```

10. Notice the simulated device sensor telemetry messages are now being sent to Azure IoT Hub every 2 seconds.

```
text 12/9/2019 5:48:07 PM > Sending message:
{"temperature":33.89822140284731,"humidity":78.34939097908763,
,"pressure":1024.9467544610131,"latitude":40.020042418755764,
"longitude":-98.41923808825841} 12/9/2019 5:48:09 PM >
Sending message:
{"temperature":27.475786026323114,"humidity":64.4175510594703,
,"pressure":1020.6866468579678,"latitude":40.2089999240047,"l
ongitude":-98.26223221770334} 12/9/2019 5:48:11 PM > Sending
message:
{"temperature":34.63600901637041,"humidity":60.95207713588703,
,"pressure":1013.6262313688063,"latitude":40.25499096898331,"
longitude":-98.51199886959347}
```

11. Within the **Terminal** pane, to exit the simulated device app, press **Ctrl-C**.
12. Switch to each of your Visual Studio Code windows and use the **Terminal** prompt to close the simulated device apps.
13. Switch the Azure portal window.
14. Close the **Device twin** blade.
15. On the **sensor-thl-2000** blade, click **Device Twin**.
16. Scroll down to locate the JSON for the `properties.reported` object.

This contains the state reported by the device.

17. Notice that the `telemetryDelay` property exists here as well, and that it has also been set to 2.

There is also a `$metadata` value that shows you when the reported values were last updated.

18. Again close the **Device twin** blade.

19. Close the **sensor-thl-2000** blade, and then navigate back to your Azure portal Dashboard.

## **Exercise 5: Deprovision a single device from the Group Enrollment**

There are many reasons why you might need to deprovision just a portion of the devices that are registered as part of a group enrollment. For example, a device may no longer be needed, a newer version of the device may have become available, or it may have been broken or compromised.

To deprovision a single device from an enrollment group, you must do two things:

- Create a disabled individual enrollment for the device's leaf (device) certificate.

This revokes access to the provisioning service for that device while still permitting access for other devices that have the enrollment group's signing certificate in their chain. Note that you should not delete the disabled individual enrollment for the device, as doing so would allow the device to re-enroll through the enrollment group.

- Disable or delete the device from the IoT hub's identity registry.

If your solution includes multiple IoT hubs, you should use the list of provisioned devices for the enrollment group to find the IoT hub that the device was provisioned to (so that you can disable or delete the device). In this case you have a single IoT hub, so you don't need to look up which IoT hub was used.

In this exercise, you will deprovision a single device from an enrollment group.

### **Task 1: Create a disabled individual enrollment for the device.**

In this task, you will use the **sensor-thl-2004** device for the individual enrollment.

1. If necessary, log in to your Azure portal using your Azure account credentials.

If you have more than one Azure account, be sure that you are logged in with the account that is tied to the subscription that you will be using for this course.

2. On the Azure portal toolbar, click **Cloud Shell**

The Azure portal toolbar runs across the top of the portal window. The Cloud Shell button is the 6th in from the right.

3. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

4. At the Cloud Shell command prompt, to navigate to the `~/certificates` directory, enter the following command:

```
sh cd ~/certificates
```

5. To download the .pem device certificates from the Cloud Shell to your local machine, enter the following commands:

```
sh download ~/certificates/certs/sensor-thl-2004-  
device.cert.pem
```

6. Switch to your Azure Dashboard.
7. On your Resources tile, click **dps-az220-training-{your-id}**.
8. On the left-side menu of your DPS blade, under **Settings**, click **Manage enrollments**.
9. On the **Manage enrollments** pane, click **Add individual enrollment**.
10. On the **Add Enrollment** blade, under **Mechanism**, ensure that **X.509** is selected.
11. Under **Primary Certificate .pem or .cer file**, click **Open**.
12. In the **Open** dialog, navigate to the downloads folder.
13. In the downloads folder, click **sensor-thl-2004-device.cert.pem**, and then click **Open**.
14. On the **Add Enrollment** blade, under **IoT Hub Device ID**, enter **sensor-thl-2004**
15. Under **Enable entry**, click **Disable**.
16. At the top of the blade, click **Save**.

17. Navigate back to your Azure Dashboard.

## **Task 2: Deregister the device from IoT hub**

1. On the Resources tile, click **iot-az220-training-{your-id}**.
2. On the left-side menu of your IoT hub blade, under **Explorers**, click **IoT devices**.
3. On the **IoT devices** pane, under **DEVICE ID**, locate the **sensor-thl-2004** device.
4. To the left of **sensor-thl-2004**, click the checkbox.
5. At the top of the **IoT devices** pane, click **Delete**, and then click **Yes**.

## **Task 3: Confirm that the device is deprovisioned**

1. Switch to the Visual Studio Code window containing your ContainerDevice2004 code project.  
  
If you closed Visual Studio Code after the previous exercise, use Visual Studio Code to open the ContainerDevice2004 folder.
2. On the **View** menu, click **Terminal**.
3. Ensure that the command prompt is located at the **ContainerDevice2004** folder location.
4. To begin running the simulated device app, enter the following command:  
  

```
cmd/sh dotnet run
```
5. Notice the exceptions listed when the device attempts to provision.

When a device attempts to connect and authenticate with Device Provisioning Service, the service first looks for an individual enrollment that matches the device's credentials. The service then searches enrollment groups to determine whether the device can be provisioned. If the service finds a disabled individual enrollment for the device, it prevents the device from connecting. The service prevents the connection even if an enabled enrollment group for an intermediate or root CA in the device's certificate chain exists.

When the application attempts to use the configured X.509 certificate to connect to DPS, DPS reports DeviceRegistrationResult.Status is NOT 'Assigned'.

```
txt Found certificate:
13F32448E03F451E897B681758BAC593A60BFBFA CN=sensor-thl-2004;
PrivateKey: True Using certificate
13F32448E03F451E897B681758BAC593A60BFBFA CN=sensor-thl-2004
ProvisioningClient AssignedHub: ; DeviceID: Unhandled
exception. System.Exception: DeviceRegistrationResult.Status
is NOT 'Assigned' at
ContainerDevice.Program.ProvisionDevice(ProvisioningDeviceCli
ent provisioningDeviceClient, SecurityProviderX509Certificate
security) in C:\Users\howdc\Allfiles\Labs\06-Automatic
Enrollment of Devices in
DPS\Starter\ContainerDevice2004\Program.cs:line 107 at
ContainerDevice.Program.Main(String[] args) in
C:\Users\howdc\Allfiles\Labs\06-Automatic Enrollment of
Devices in DPS\Starter\ContainerDevice2004\Program.cs:line 49
at ContainerDevice.Program.<Main>(String[] args)
```

If you were to go back into your Azure portal and either enable the individual enrollment or delete the individual enrollment, the device will once again be able to authenticate with DPS and connect to IoT hub. If the individual enrollment is deleted, the device is automatically added back to the group enrollment.

## Exercise 6: Deprovision the Group Enrollment

In this exercise, you will deprovision the full enrollment group. Again, this includes disenrolling from Device Provisioning Service and deregistering the devices from IoT Hub.

### Task 1: Disenroll the enrollment group from the DPS

In this task, you will delete your Enrollment Group, which will remove the enrolled devices.

1. Navigate to your Azure Dashboard.
2. On your Resources tile, click **dps-az220-training-{your-id}**.
3. On the left-side menu of your DPS blade, under **Settings**, click **Manage enrollments**.
4. In the list of **Enrollment Groups**, under **GROUP NAME**, click **eg-test-simulated-devices**.

5. On the **Enrollment Group Details** blade, scroll down to locate the **Enable entry** field, and then click **Disable**.

Disabling the Group Enrollment within DPS allows you to temporarily disable devices within this Enrollment Group. This provides a temporary list of the X.509 certificate that should not be used by these devices.

6. At the top of the blade, click **Save**.

If you run one of your simulated devices now, you will see an error message similar to what you saw with the disabled individual enrollment.

To permanently delete the Enrollment Group, you must delete the enrollment group from DPS.

7. On the **Manage enrollments** pane, under **GROUP NAME**, select the check box to the left of **eg-test-simulated-devices**.

If the check box to the left of **simulated-devices** was already checked, leave it checked.

8. At the top of the **Manage enrollments** pane, click **Delete**.

9. When prompted to confirm the action to **Remove enrollment**, click **Yes**.

Once deleted, the Group Enrollment is completely removed from DPS, and would need to be recreated to add it back.

**Note:** If you delete an enrollment group for a certificate, devices that have the certificate in their certificate chain might still be able to enroll if a different, enabled enrollment group still exists for the root certificate or another intermediate certificate higher up in their certificate chain.

10. Navigate back to your Azure portal Dashboard

## **Task 2: Deregister the devices from the IoT Hub**

Once the enrollment group has been removed from the Device Provisioning Service (DPS), the device registration will still exist within Azure IoT Hub. To fully deprovision the devices, you will need to remove that registration as well.

1. On your Resources tile, click **iot-az220-training-{your-id}**.
2. On the left-side menu of your IoT hub blade, under **Explorers**, click **IoT devices**.

3. Notice that the device IDs for **sensor-thl-2000** and the other group enrolled devices still exist within the Azure IoT Hub device registry.
4. To remove the sensor-thl-2000 device, select the check box to the left of **sensor-thl-2000**, then click **Delete**.
5. When prompted with "*Are you certain you wish to delete selected device(s)*", click **Yes**.
6. Repeat the steps 4-5 above to remove the following devices:
  - sensor-thl-2001
  - sensor-thl-2002
  - sensor-thl-2003

### **Task 3: Confirm that your devices have been deprovisioned**

With the group enrollment deleted from the Device Provisioning Service, and the device deleted from the Azure IoT Hub device registry, the device(s) have been fully removed from the solution.

1. Switch to the Visual Studio Code window containing your ContainerDevice code project.

If you closed Visual Studio Code after the previous exercise, use Visual Studio Code to open the lab 6 Starter folder.

2. On the Visual Studio Code **View** menu, click **Terminal**.
3. Ensure that the command prompt is located at the **ContainerDevice** folder location.
4. To begin running the simulated device app, enter the following command:

```
cmd/sh dotnet run
```

5. Notice the exceptions listed when the device attempts to provision.

Now that the group enrollment and registered device have been deleted, the simulated device will no longer be able to provision or connect. When the application attempts to use the configured X.509 certificate to connect to DPS, it will return a `ProvisioningTransportException` error message.

```
txt Found certificate:
AFF851ED016CA5AEB71E5749BCBE3415F8CF4F37 CN=sensor-thl-2000;
```



PrivateKey: True Using certificate  
AFF851ED016CA5AEB71E5749BCBE3415F8CF4F37 CN=sensor-thl-2000  
RegistrationID = sensor-thl-2000 ProvisioningClient  
RegisterAsync . . . Unhandled exception.  
Microsoft.Azure.Devices.Provisioning.Client.ProvisioningTransportException: {"errorCode":401002,"trackingId":"df969401-c766-49a4-bab7-e769cd3cb585","message":"Unauthorized","timestampUtc":"2019-12-20T21:30:46.6730046Z"} at  
Microsoft.Azure.Devices.Provisioning.Client.Transport.ProvisioningTransportHandlerAmqp.ValidateOutcome(Outcome outcome) at  
Microsoft.Azure.Devices.Provisioning.Client.Transport.ProvisioningTransportHandlerAmqp.RegisterDeviceAsync(AmqpClientConnection client, String correlationId, DeviceRegistration deviceRegistration) at  
Microsoft.Azure.Devices.Provisioning.Client.Transport.ProvisioningTransportHandlerAmqp.RegisterAsync(ProvisioningTransport RegisterMessage message, CancellationToken cancellationToken)

**You have completed the registration, configuration, and deprovisioning as part of the IoT devices life cycle with Device Provisioning Service.**

lab: title: 'Lab 07: Device Message Routing' module: 'Module 4: Message Processing and Analytics'

---

# Device Message Routing

## Lab Scenario

Contoso Management is impressed with your implementation of automatic device enrollment using DPS. They are now interested in having you develop an IoT-based solution related to product packaging and shipping.

The cost associated with packaging and shipping cheese is significant. To maximize cost efficiency, Contoso operates an on-premises packaging facility. The workflow is straightforward - cheese is cut and packaged, packages are assembled into shipping containers, containers are delivered to specific bins associated with their destination. A conveyor belt system is used to move the product through this process. The metric for success is the number of packages leaving the conveyor belt system during a given time period (typically a work shift).

The conveyor belt system is a critical link in this process and is visually monitored to ensure that the workflow is progressing at maximum efficiency. The system has three operator controlled speeds: stopped, slow, and fast. Naturally, the number of packages being delivered at the low speed is less than at the higher speed. However, there are a number of other factors to consider:

- the vibration level of the conveyor belt system is much lower at the slow speed
- high vibration levels can cause packages to fall from the conveyor
- high vibration levels are known to accelerate wear-and-tear of the system
- when vibration levels exceed a threshold limit, the conveyor belt must be stopped to allow for inspection (to avoid more serious failures)

In addition to maximizing throughput, your automated IoT solution will implement a form of preventive maintenance based on vibration levels, which will be used to detect early warning signs before serious system damage occurs.

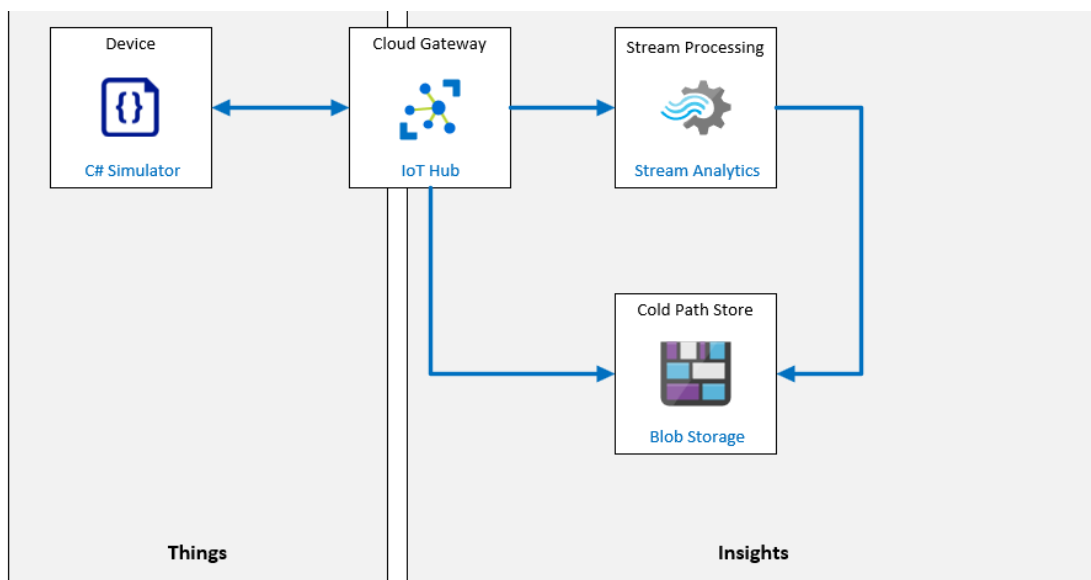
**Note: Preventive maintenance** (sometimes called preventative maintenance or predictive maintenance) is an equipment maintenance program that schedules maintenance activities to be performed while the equipment is operating normally. The intent of this approach is to avoid unexpected breakdowns that often incur costly disruptions.

It's not always easy for an operator to visually detect abnormal vibration levels. For this reason, you are looking into an Azure IoT solution that will help to measure vibration levels and data anomalies. Vibration sensors will be attached to the conveyor belt at various locations, and you will use IoT devices to send telemetry to IoT hub. The IoT hub will use Azure Stream Analytics, and a built-in Machine Learning (ML) model, to alert you to vibration anomalies in real time. You also plan to archive all of the telemetry data so that in-house machine learning models can be developed in the future.

You decide to prototype the solution using simulated telemetry from a single IoT device.

To simulate the vibration data in a realistic manner, you work with an engineer from Operations to understand a little bit about what causes the vibrations. It turns out there are a number of different types of vibration that contribute to the overall vibration level. For example, a "force vibration" could be introduced by a broken guide wheel or an especially heavy load placed improperly on the conveyor belt. There's also an "increasing vibration", that can be introduced when a system design limit (such as speed or weight) is exceeded. The Engineering team agrees to help you develop the code for a simulated IoT device that will produce an acceptable representation of vibration data (including anomalies).

The following resources will be created:



## In This Lab

In this lab, you will begin by reviewing the lab prerequisites and you will run a script if needed to ensure that your Azure subscription includes the required resources. You will then create a simulated device that sends vibration telemetry to your IoT hub. With your simulated data arriving at IoT hub, you will implement an IoT Hub Message Route and Azure Stream Analytics job that can be used to archive data. The lab includes the following exercises:

- Verify Lab Prerequisites
- A script will be used to create any missing resources and a new device identity (sensor-v-3000) for this lab
- Write Code to generate Vibration Telemetry
- Create a Message Route to Azure Blob Storage
- Logging Route Azure Stream Analytics Job

# Lab Instructions

## Exercise 1: Verify Lab Prerequisites

This lab assumes that the following Azure resources are available:

Resource Type	Resource Name
Resource Group	rg-az220
IoT Hub	iot-az220-training-{your-id}
Device ID	sensor-v-3000

**Important:** Run the setup script to create the required device.

To create any missing resources and the new device you will need to run the **lab07-setup.azcli** script as instructed below before moving on to Exercise 2. The script file is included in the GitHub repository that you cloned locally as part of the dev environment configuration (lab 3).

The **lab07-setup.azcli** script is written to run in a **bash** shell environment - the easiest way to execute this is in the Azure Cloud Shell.

1. Using a browser, open the [Azure Cloud Shell](#) and login with the Azure subscription you are using for this course.

If you are prompted about setting up storage for Cloud Shell, accept the defaults.

2. Verify that the Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

3. On the Cloud Shell toolbar, click **Upload/Download files** (fourth button from the right).
4. In the dropdown, click **Upload**.
5. In the file selection dialog, navigate to the folder location of the GitHub lab files that you downloaded when you configured your development environment.

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 07-Device Message Routing
    - Setup

The lab07-setup.azcli script file is located in the Setup folder for lab 7.

6. Select the **lab07-setup.azcli** file, and then click **Open**.

A notification will appear when the file upload has completed.

7. To verify that the correct file has uploaded in Azure Cloud Shell, enter the following command:

```
bash ls
```

The `ls` command lists the content of the current directory. You should see the lab07-setup.azcli file listed.

8. To create a directory for this lab that contains the setup script and then move into that directory, enter the following Bash commands:

```
bash mkdir lab7 mv lab07-setup.azcli lab7 cd lab7
```

9. To ensure that **lab07-setup.azcli** has the execute permission, enter the following command:

```
bash chmod +x lab07-setup.azcli
```

10. On the Cloud Shell toolbar, to enable access to the lab07-setup.azcli file, click **Open Editor** (second button from the right - { }).
11. In the **FILES** list, to expand the lab7 folder and open the script file, click **lab7**, and then click **lab07-setup.azcli**.

The editor will now show the contents of the **lab07-setup.azcli** file.

12. In the editor, update the {your-id} and {your-location} assigned values.

Referencing the sample below as an example, you need to set {your-id} to the Unique ID you created at the start of this course - i.e. **cah191211**,

and set {your-location} to the location that makes sense for your resources.

```
```bash
```



# !/bin/bash

## Change these values!

```
YourID="{your-id}" Location="{your-location}" ``
```

**Note:** The `{your-location}` variable should be set to the short name for the region where you are deploying all of your resources. You can see a list of the available locations and their short-names (the **Name** column) by entering this command:

```
``bash az account list-locations -o Table
```

```
DisplayName Latitude Longitude Name
```

---

```
East Asia 22.267 114.188 eastasia Southeast Asia 1.283 103.833  
southeastasia Central US 41.5908 -93.6208 centralus East US 37.3719  
-79.8164 eastus East US 2 36.6681 -78.3889 eastus2 ``
```

13. In the top-right of the editor window, to save the changes made to the file and close the editor, click **...**, and then click **Close Editor**.

If prompted to save, click **Save** and the editor will close.

**Note:** You can use **CTRL+S** to save at any time and **CTRL+Q** to close the editor.

14. To create the resources required for this lab, enter the following command:

```
bash ./lab07-setup.azcli
```

This script can take a few minutes to run. You will see output as each step completes.

The script will first create a resource group named **rg-az220** and an IoT Hub named **iot-az220-training-{your-id}**. If they already exist, a corresponding message will be displayed. The script will then add a device with an ID of **sensor-v-3000** to the IoT hub and display the device connection string.

15. Notice that, once the script has completed, the connection string for the device is displayed.

The connection string starts with "HostName="

16. Copy the connection string into a text document, and note that it is for the **sensor-v-3000** device.

Once you have saved the connection string to a location where you can find it easily, you will be ready to continue with the lab.

## **Exercise 2: Write Code to generate Vibration Telemetry**

Both long term and real-time data analysis are required to automate the monitoring of Contoso's conveyor belt system and enable predictive maintenance. Since no historical data exists, your first step will be to generate simulated data that mimics vibration data and data anomalies in a realistic manner. Contoso engineers have developed an algorithm to simulate vibration over time and embedded the algorithm within a code class that you will implement. The engineers have agreed to support any future updates required to adjust the algorithms.

During your initial prototype phase, you will implement a single IoT device that generates telemetry data. In addition to the vibration data, your device will create some additional values (packages delivered, ambient temperature, and similar metrics) that will be sent to Blob storage. This additional data simulates the data that will be used to develop machine learning modules for predictive maintenance.

In this exercise, you will:

- load the simulated device project
- update the connection string for your simulated device and review the project code
- test your simulated device connection and telemetry communications
- ensure that telemetry is arriving at your IoT hub

### **Task 1: Open your simulated device project**

1. Open **Visual Studio Code**.
2. On the **File** menu, click **Open Folder**.
3. In the **Open Folder** dialog, navigate to the **07-Device Message Routing** folder.

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
  - Labs
    - 07-Device Message Routing
      - Starter
        - VibrationDevice

4. Navigate to the **Starter** folder for Lab 7.

5. Click **VibrationDevice**, and then click **Select Folder**.

You should see the following files listed in the EXPLORER pane of Visual Studio Code:

- Program.cs
- VibrationDevice.csproj

**Note:** If you are prompted to load required assets, you can do that now.

6. In the **EXPLORER** pane, click **Program.cs**.

A cursory glance will reveal that the **VibrationDevice** application is very similar to those used in the preceding labs. This version of the application uses symmetric Key authentication, sends both telemetry and logging messages to the IoT Hub, and has a more complex sensor implementation.


7. On the **Terminal** menu, click **New Terminal**.

Examine the directory path indicated as part of the command prompt to ensure that you are in the correct location. You do not want to start building this project within the folder structure of a previous lab project.

8. At the terminal command prompt, to verify that the application builds without errors, enter the following command:

```
cmd dotnet build
```

The output will be similar to:

```text  dotnet build Microsoft (R) Build Engine version 16.5.0+d4cbfca49 for .NET Core Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 39.27 ms for D:\Az220-Code\AllFiles\Labs\07-Device Message  
Routing\Starter\VibrationDevice\VibrationDevice.csproj.  
VibrationDevice -> D:\Az220-Code\AllFiles\Labs\07-Device Message  
Routing\Starter\VibrationDevice\bin\Debug\netcoreapp3.1\VibrationDevice.dll

Build succeeded. 0 Warning(s) 0 Error(s)

Time Elapsed 00:00:01.16 ```

In the next task, you will configure the connection string and review the application.

## Task 2: Configure connection and review code

The simulated device app that you will build in this task simulates an IoT device that is monitoring the conveyor belt. The app will simulate sensor readings and report vibration sensor data every two seconds.

1. Ensure that you have the **Program.cs** file opened in Visual Studio Code.
2. Near the top of the **Program** class, locate the declaration of the `deviceConnectionString` variable:

```
csharp private readonly static string deviceConnectionString  
= "<your device connection string>";
```

3. Replace `<your device connection string>` with the device connection string that you saved earlier.

**Note:** This is the only change that you are required to make to this code.

4. On the **File** menu, click **Save**.
5. Take a minute to review the structure of the project.

Notice that the application structure is similar to that of your previous simulated device projects.

- Using statements

- Namespace definition
- Program class - responsible for connecting to Azure IoT and sending telemetry
- ConveyorBeltSimulator class - (replaces EnvironmentSensor) rather than just generating telemetry, this class also simulates a running conveyor belt
- ConsoleHelper - a new class that encapsulates writing different colored text to the console

## 6. Take a minute to review the **Main** method.

```
```csharp private static void Main(string[] args) {
    ConsoleHelper.WriteColorMessage("Vibration sensor device app.\n",
    ConsoleColor.Yellow);

    // Connect to the IoT hub using the MQTT protocol.
    deviceClient =
    DeviceClient.CreateFromConnectionString(deviceConnectionString,
    TransportType.Mqtt);

    SendDeviceToCloudMessagesAsync();
    Console.ReadLine();

} ```
```

Notice how straightforward it is to create an instance of **DeviceClient** using the **deviceConnectionString** variable. Since the `deviceClient` object is declared outside of **Main** (at the Program level in the code above), it is global and therefore available inside the methods that communicate with IoT hub.

## 7. Take a minute to review the **SendDeviceToCloudMessagesAsync** method.

```
```csharp private static async void SendDeviceToCloudMessagesAsync()
{ var conveyor = new ConveyorBeltSimulator(intervalInMilliseconds);

    // Simulate the vibration telemetry of a conveyor belt.
    while (true)
    {
        var vibration = conveyor.ReadVibration();

        await CreateTelemetryMessage(conveyor, vibration);

        await CreateLoggingMessage(conveyor, vibration);

        await Task.Delay(intervalInMilliseconds);
    }
}
```

```
} ``
```

First off, notice that this method is being used to establish the infinite program loop, first taking a vibration reading and then sending messages at a defined time interval.

A closer look reveals that the `ConveyorBeltSimulator` class is used to create a `ConveyorBeltSimulator` instance named `conveyor`. The `conveyor` object is first used to capture a vibration reading which is placed into a local `vibration` variable, and is then passed to the two create message methods along with the `vibration` value that was captured at the start of the interval.

#### 8. Take a minute to review the **CreateTelemetryMessage** method.

```
``csharp private static async Task
CreateTelemetryMessage(ConveyorBeltSimulator conveyor, double
vibration) { var telemetryDataPoint = new { vibration = vibration, }; var
telemetryMessageString =
JsonConvert.SerializeObject(telemetryDataPoint); var telemetryMessage
= new Message(Encoding.ASCII.GetBytes(telemetryMessageString));

// Add a custom application property to the message. This is
used to route the message.
telemetryMessage.Properties.Add("sensorID", "VSTel");

// Send an alert if the belt has been stopped for more than
five seconds.
telemetryMessage.Properties.Add("beltAlert",
(conveyor.BeltStoppedSeconds > 5) ? "true" : "false");

Console.WriteLine($"Telemetry data:
{telemetryMessageString}");

// Send the telemetry message.
await deviceClient.SendEventAsync(telemetryMessage);
ConsoleHelper.WriteGreenMessage($"Telemetry sent
{DateTime.Now.ToShortTimeString()}");

} ``
```

As in earlier labs, this method creates a JSON message string and uses the **Message** class to send the message, along with additional properties. Notice the **sensorID** property - this will be used to route the **VSTel** values appropriately at the IoT Hub. Also notice the **beltAlert** property - this is set to true if the conveyor belt has stopped for more than 5 seconds.

As usual, the message is sent via the **SendEventAsync** method of the device client.

9. Take a minute to review the **CreateLoggingMessage** method.

```
```csharp private static async Task
CreateLoggingMessage(ConveyorBeltSimulator conveyor, double
vibration) { // Create the logging JSON message. var loggingDataPoint =
new { vibration = Math.Round(vibration, 2), packages =
conveyor.PackageCount, speed = conveyor.BeltSpeed.ToString(), temp =
Math.Round(conveyor.Temperature, 2), }; var loggingMessageString =
JsonConvert.SerializeObject(loggingDataPoint); var loggingMessage =
new Message(Encoding.ASCII.GetBytes(loggingMessageString));

// Add a custom application property to the message. This is
used to route the message.
loggingMessage.Properties.Add("sensorID", "VSLog");

// Send an alert if the belt has been stopped for more than
five seconds.
loggingMessage.Properties.Add("beltAlert",
(conveyor.BeltStoppedSeconds > 5) ? "true" : "false");

Console.WriteLine($"Log data: {loggingMessageString}");

// Send the logging message.
await deviceClient.SendEventAsync(loggingMessage);
ConsoleHelper.WriteGreenMessage("Log data sent\n");

} ```
```

Notice that this method is very similar to the **CreateTelemetryMessage** method. Here are the key items to note:

- The **loggingDataPoint** contains more information than the telemetry object. It is common to include as much information as possible for logging purposes to assist in any fault diagnosis activities or more detailed analytics in the future.
- The logging message includes the **sensorID** property, this time set to **VSLog**. Again, as noted above, this will be used to route the **VSLog** values appropriately at the IoT Hub.

10. Optionally, take a moment to review the **ConveyorBeltSimulator** class and the **ConsoleHelper** class.

You don't actually need to understand how either of these classes work to achieve the full value of this lab, but they both support the outcome in their own way. The **ConveyorBeltSimulator** class simulates the

operation of a conveyor belt, modeling a number of speeds and related states to generate vibration data. The **ConsoleHelper** class is used to write different colored text to the console to highlight different data and values.

### Task 3: Test your code to send telemetry

1. At the Terminal command prompt, to run the app, enter the following command:

```
bash dotnet run
```

This command will run the **Program.cs** file in the current folder.

1. Console output should be displayed that is similar to the following:

```
``text Vibration sensor device app.
```

```
Telemetry data: {"vibration":0.0} Telemetry sent 10:29 AM Log data:
{"vibration":0.0,"packages":0,"speed":"stopped","temp":60.22} Log data
sent
```

```
Telemetry data: {"vibration":0.0} Telemetry sent 10:29 AM Log data:
{"vibration":0.0,"packages":0,"speed":"stopped","temp":59.78} Log data
sent ``
```

**Note:** In the Terminal window, green text is used to show things are working as they should and red text when bad stuff is happening. If you receive error messages, start by checking your device connection string.

2. Leave this app running for the next task.

If you won't be continuing to the next task, you can enter **Ctrl-C** in the Terminal window to stop the app. You can start it again later by using the `dotnet run` command.

### Task 4: Verify the IoT Hub is Receiving Telemetry

In this task, you will use the Azure portal to verify that your IoT Hub is receiving telemetry.

1. Open the [Azure Portal](#).
2. On your Resources tile, click **iot-az220-training-{your-id}**.



3. On the **Overview** pane, scroll down to view the metrics tiles.
4. Adjacent to **Show data for last**, change the time range to one hour.

The **Device to cloud messages** tile should be plotting some current activity. If no activity is shown, wait a short while, as there's some latency.

With your device pumping out telemetry, and your hub receiving it, the next step is to route the messages to their correct endpoints.

### Exercise 3: Create a Message Route to Azure Blob Storage

IoT solutions often require that incoming message data be sent to multiple endpoint locations, either dependent upon the type of data or for business reasons. Azure IoT hub provides the *message routing* feature to enable you to direct incoming data to locations required by your solution.

The architecture of our system requires data be sent to two destinations: a storage location for archiving data, and a location for more immediate analysis.

Contoso's vibration monitoring scenario requires you to create two message routes:

- the first route will be to an Azure Blob storage location for data archiving
- the second route will be to an Azure Stream Analytics job for real-time analysis

Message routes should be built and tested one at a time, so this exercise will focus on the storage route. This route will be referred to as the "logging" route, and it involves digging a few levels deep into the creation of Azure resources.

One important feature of message routing is the ability to filter incoming data before routing to an endpoint. The filter, written as a SQL query, directs output through a route only when certain conditions are met.

One of the easiest ways to filter data is to evaluate a message property. You may recall adding message properties to your device messages in the previous exercise. The code that you added looked like the following:

```
csharp ... telemetryMessage.Properties.Add("sensorID", "VSTel");  
... loggingMessage.Properties.Add("sensorID", "VSLog");
```

You can now embed a SQL query within your message route that uses `sensorID` as a criteria for the route. In this case, when the value assigned to

`sensorID` is `VSLog` (vibration sensor log), the message is intended for the storage archive.

In this exercise, you will create and test the logging route.

### Task 1: Define the message routing endpoint

1. In the [Azure Portal](#), ensure that your IoT hub blade is open.
2. On the left-hand menu, under **Messaging**, click **Message routing**.
3. On the **Message routing** pane, ensure that the **Routes** tab is selected.
4. To add a new route, click **+ Add**.

The **Add a route** blade should now be displayed.

5. On the **Add a route** blade, under **Name**, enter **vibrationLoggingRoute**
6. To the right of **Endpoint**, click **+ Add endpoint**, and then, in the drop-down list, click **Storage**.

The **Add a storage endpoint** blade should now be displayed.

7. On the **Add a storage endpoint** blade, under **Endpoint name**, enter **vibrationLogEndpoint**
8. To display a list of Storage accounts associated with your subscription, click **Pick a container**.

A list of the storage accounts already present in the Azure Subscription is listed. At this point you could select an existing storage account and container, however, for this lab you will create a new one.

9. To begin creating a storage account, click **+ Storage account**.

The **Create storage account** blade should now be displayed.

10. On the **Create storage account** blade, under **Name**, enter **vibrationstore{your-id}**

For example: **vibrationstorecah191211**

**Note:** This field can only contain lower-case letters and numbers, must be between 3 and 24 characters, and must be unique.

11. In the **Account kind** dropdown, click **StorageV2 (general purpose v2)**.

12. Under **Performance**, ensure that **Standard** is selected.

This keeps costs down at the expense of overall performance.

13. Under **Replication**, ensure that **Locally-redundant storage (LRS)** is selected.

This keeps costs down at the expense of risk mitigation for disaster recovery. In production your solution may require a more robust replication strategy.

14. Under **Location**, select the region that you are using for the labs in this course.

15. To create the storage account endpoint, click **OK**.

16. Wait until the request is validated and the storage account deployment has completed.

Validation and creation can take a minute or two.

Once completed, the **Create storage account** blade will close and the **Storage accounts** blade will be displayed. The Storage accounts blade should have auto-updated to show the storage account that was just created.

## **Task 2: Define the storage account container**

1. On the **Storage accounts** blade, click **vibrationstore{your-id}**.

The **Containers** blade should appear. Since this is a new storage account, there are no containers listed.

2. To create a container, click **+ Container**.

The **New container** dialog should now be displayed.

3. On the **New container** dialog, under **Name**, enter **vibrationcontainer**

Again, only lower-case letters and numbers are accepted.

1. Under **Public access level**, ensure that **Private (no anonymous access)** is selected.

2. To create the container, click **Create**.

After a moment the **Lease state** for your container will update to display **Available**.

3. To choose this container for your solution, click **vibrationcontainer**, and then click **Select**.

You should be returned to the **Add a storage endpoint** blade. Notice that the **Azure Storage container** has been set to the URL for the storage account and container you just created.

4. Leave the **Batch frequency** and **Chunk size window** fields set to their default values of **100**.
5. Under **Encoding**, notice that there are two options and that **AVRO** is selected.

**Note:** By default IoT Hub writes the content in Avro format, which has both a message body property and a message property. The Avro format is not used for any other endpoints. Although the Avro format is great for data and message preservation, it's a challenge to use it to query data. In comparison, JSON or CSV format is much easier for querying data. IoT Hub now supports writing data to Blob storage in JSON as well as AVRO.

6. Take a moment to examine the value specified in **File name format** field.

The **File name format** field specifies the pattern used to write the data to files in storage. The various tokens are replaced with values as the file is created.

7. At the bottom of the blade, to create your storage endpoint, click **Create**.

Validation and subsequent creation will take a few moments. Once complete, you should be located back on the **Add a route** blade.

### Task 3: Define the routing query

1. On the **Add a route** blade, under **Data source**, ensure that **Device Telemetry Messages** is selected.
2. Under **Enable route**, ensure that **Enable** is selected.
3. Under **Routing query**, replace **true** with the query below:

```
sql sensorID = 'VSLog'
```

This query ensures that only messages with the `sensorID` application property set to `VSLog` will be routed to the storage endpoint.

4. To save this route, click **Save**.

Wait for the success message. Once completed, the route should be listed on the **Message routing** pane.

5. Navigate back to your Azure portal Dashboard.

#### Task 4: Verify Data Archival

1. Ensure that the device app you created in Visual Studio Code is still running.

If not, run it in the Visual Studio Code terminal using `dotnet run`.

2. On your Resources tile, to open your Storage account blade, click **vibrationstore{your-id}**.

If your Resources tile does not list your Storage account, click the **Refresh** button at the top of the resource group tile, and then follow the instruction above to open your storage account.

3. On the left-side menu of your **vibrationstore{your-id}** blade, click **Storage Explorer (preview)**.

You can use the Storage Explorer to verify that your data is being added to the storage account.

**Note:** The Storage Explorer is currently in preview mode, so its exact mode of operation may change.

4. In **Storage Explorer (preview)** pane, expand **BLOB CONTAINERS**, and then click **vibrationcontainer**.

To view the data, you will need to navigate down a hierarchy of folders. The first folder will be named for the IoT Hub.

5. In the right-hand pane, under **NAME**, double-click **iot-az220-training-{your-id}**, and then use double-clicks to navigate down into the hierarchy.

Under your IoT hub folder, you will see folders for the Partition, then numeric values for the Year, Month, and Day. The final folder represents the Hour, listed in UTC time. The Hour folder will contain a number of Block Blobs that contain your logging message data.

6. Double-click the Block Blob for the data with the earliest time stamp.

The URL link will open in a new browser tab. Although the data is not formatted in a way that is easy to read, you should be able to recognize it as your vibration messages.

7. Close browser tab containing your data, and then navigate back to your Azure portal Dashboard.

## Exercise 4: Logging Route Azure Stream Analytics Job

In this exercise, you will create a Stream Analytics job that outputs logging messages to Blob storage. You will then use Storage Explorer in the Azure Portal to view the stored data.

This will enable you to verify that your route includes the following settings:

- **Name** - vibrationLoggingRoute
- **Data Source** - DeviceMessages
- **Routing query** - sensorID = 'VSLog'
- **Endpoint** - vibrationLogEndpoint
- **Enabled** - true

**Note:** It may seem odd that in this lab you are routing data to storage, and then also sending your data to storage through Azure Stream Analytics. In a production scenario, you wouldn't have both paths long-term. Instead, it is likely that the second path that we're creating here would not exist. You will use it here, in a lab environment, as a way to validate that your routing is working as expected and to show a simple implementation of Azure Stream Analytics.

### Task 1: Create the Stream Analytics Job

1. On the Azure portal menu, click **+ Create a resource**.
2. On the **New** blade, in the **Search the Marketplace** textbox, type **stream analytics** and then click **Stream Analytics job**.
3. On the **Stream Analytics job** blade, click **Create**.

The **New Stream Analytics job** pane is displayed.

4. On the **New Stream Analytics job** pane, under **Name**, enter **vibrationJob**.
5. Under **Subscription**, choose the subscription you are using for the lab.
6. Under **Resource group**, select **rg-az220**.
7. Under **Location**, select the region that you are using for the labs in this course.
8. Under **Hosting environment**, ensure that **Cloud** is selected.

Edge hosting will be discussed later in the course.

9. Under **Streaming units**, reduce the number from **3** to **1**.

This lab does not require 3 units and this will reduce costs.

10. To create the Stream Analytics job, click **Create**.
11. Wait for the **Deployment succeeded** message, then open the new resource.

**Tip:** If you miss the message to go to the new resource, or need to find a resource at any time, select **Home/All resources**. Enter enough of the resource name for it to appear in the list of resources.

12. Take a moment to examine your new Stream Analytics job.

Notice that you have an empty job, showing no inputs or outputs, and a skeleton query. The next step is to populate these entries.

13. On the left-side menu under **Job topology**, click **Inputs**.

The **Inputs** pane will be displayed.

14. On the **Inputs** pane, click **+ Add stream input**, and then click **IoT Hub**.

The **IoT Hub - New input** pane will be displayed.

15. On the **IoT Hub - New input** pane, under **Input alias**, enter `vibrationInput`.

16. Ensure that **Select IoT Hub from your subscriptions** is selected.

17. Under **Subscription**, ensure that the subscription you used to create the IoT Hub earlier is selected.
18. Under **IoT Hub**, ensure that your **iot-az220-training-{your-id}** IoT hub is selected.
19. Under **Endpoint**, ensure that **Messaging** is selected.
20. Under **Shared access policy name**, ensure that **iothubowner** is selected.

**Note:** The **Shared access policy key** is populated and read-only.

21. Under **Consumer group**, ensure that **\$Default** is selected.
22. Under **Event serialization format**, ensure that **JSON** is selected.
23. Under **Encoding**, ensure that **UTF-8** is selected.

You may need to scroll down to see some of the fields.

24. Under **Event compression type**, ensure **None** is selected.
25. To save the new input, click **Save**, and then wait for the input to be created.

The **Inputs** list should be updated to show the new input.

26. To create an output, on the left-side menu under **Job topology**, click **Outputs**.

The **Outputs** pane is displayed.

27. On the **Outputs** pane, click **+ Add**, and then click **Blob storage/Data Lake Storage Gen2**.

The **Blob storage/Data Lake Storage Gen2 - New output** pane is displayed.

28. On the **Blob storage/Data Lake Storage Gen2 - New output** pane, under **Output alias**, enter `vibrationOutput`.
29. Ensure that **Select storage from your subscriptions** is selected.
30. Under **Subscription**, select the subscription you are using for this lab.
31. Under **Storage account**, click **vibrationstore{your-id}**.



**Note:** The **Storage account key** is automatically populated and read-only.

32. Under **Container**, ensure that **Use existing** is selected and that **vibrationcontainer** is selected from the dropdown list.
33. Leave the **Path pattern** blank.
34. Leave the **Date format** and **Time format** at their defaults.
35. Under **Event serialization format**, ensure that **JSON** is selected.
36. Under **Encoding**, ensure that **UTF-8** is selected.
37. Under **Format**, ensure that **Line separated** is selected.

**Note:** This setting stores each record as a JSON object on each line and, taken as a whole, results in a file that is an invalid JSON record. The other option, **Array**, ensures that the entire document is formatted as a JSON array where each record is an item in the array. This allows the entire file to be parsed as valid JSON.

38. Leave **Minimum rows** blank.
39. Under **Maximum time**, leave **Hours** and **Minutes** blank.
40. Under **Authentication mode**, ensure that **Connection string** is selected.
41. To create the output, click **Save**, and then wait for the output to be created.

The **Outputs** list will be updated with the new output.

42. To edit the query, on the left-side menu under **Job topology**, click **Query**.
43. In the query editor pane, replace the existing query with the query below:  

```
sql SELECT * INTO vibrationOutput FROM vibrationInput
```
44. Directly above the query editor pane, click **Save Query**.
45. On the left-side menu, click **Overview**.

## **Task 2: Test the Logging Route**

Now for the fun part. Does the telemetry your device app is pumping out work its way along the route, and into the storage container?

1. Ensure that the device app you created in Visual Studio Code is still running.

If not, run it in the Visual Studio Code terminal using `dotnet run`.

2. On the **Overview** pane of your Stream Analytics job, click **Start**.
3. In the **Start job** pane, leave the **Job output start time** set to **Now**, and then click **Start**.

It can take a few moments for the job to start.

4. On the Azure portal menu, click **Dashboard**.
5. On your Resources tile, click **vibrationstore{your-id}**.

If your Storage account is not visible, use the **Refresh** button at the top of the resource group tile.

6. On the **Overview** pane of your Storage account, scroll down until you can see the **Monitoring** section.
7. Under **Monitoring**, adjacent to **Show data for last**, change the time range to **1 hour**.

You should see activity in the charts.

8. On the left-side menu, click **Storage Explorer (preview)**.

You can use Storage Explorer for additional reassurance that all of your data is getting to the storage account.

**Note:** The Storage Explorer is currently in preview mode, so its exact mode of operation may change.

9. In **Storage Explorer (preview)**, under **BLOB CONTAINERS**, click **vibrationcontainer**.

To view the data, you will need to navigate down a hierarchy of folders. The first folder will be named for the IoT Hub, the next will be a partition, then year, month, day and finally hour.

10. In the right-hand pane, under **Name**, double-click the folder for your IoT hub, and then use double-clicks to navigate down into the hierarchy until

you open the most recent hour folder.

Within the hour folder, you will see files named for the minute they were generated. This verifies that your data is reaching the storage location as intended.

11. Navigate back to your Dashboard.
12. On your Resources tile, click **vibrationJob**.
13. On the **vibrationJob** blade, click **Stop**, and then click **Yes**.

You've traced the activity from the device app, to the hub, down the route, and to the storage container. Great progress! You will continue this scenario stream analytics in the next module when you take a quick look at data visualization.

14. Switch to the Visual Studio Code window.
15. At the Terminal command prompt, to exit the device simulator app, press **CTRL-C**.

**> IMPORTANT: Do not remove these resources until you have completed the Data Visualization module of this course.**

lab: title: 'Lab 08: Visualize a Data Stream in Power BI' module: 'Module 5: Insights and Business Integration'

---

# Visualize a Data Stream in Power BI

**IMPORTANT:** This lab has several service prerequisites that are not related to the Azure subscription you were given for the course:

1. The ability to sign in to a "Work or School Account" (Azure Active Directory account)
2. You must know your account sign-in name, which may not match your e-mail address.
3. Access to Power BI, which could be through:
  1. An existing Power BI account
  2. The ability to sign up for Power BI - some organizations block this.

The first lab exercise will validate your ability to access Power BI. If you are not successful in the first exercise, you will not be able to complete the lab, as there is no quick workaround for blocked access to a work or school account.

## Lab Scenario

You have developed a simulated IoT device that generates vibration data and other telemetry outputs that are representative of the conveyor belt system used in Contoso's cheese packaging process. You have built and tested a logging route that sends data to Azure Blob storage. You will now start work on a new route within IoT hub that will send telemetry data to an Azure Event Hubs service.

The primary difference between Azure IoT Hub and Azure Event Hubs is that Event Hubs is designed for big data streaming, while IoT hub is optimized for an IoT solution. Both services support ingestion of data with low latency and high reliability. Since Azure Event Hubs provides an input to Stream Analytics in a manner that is similar to IoT hub, your choice of Event Hubs in this case allows you explore an additional Azure service option within your solution.

## Make a Call to a Built-in Machine Learning Model

In this lab, you will be calling a built-in Machine Learning (ML) function named `AnomalyDetection_SpikeAndDip`. The `AnomalyDetection_SpikeAndDip` function uses a sliding window to analyze data for anomalies. The sliding window could be, for example, the most recent two minutes of telemetry data. The window advances in near real-time with the flow of telemetry. Generally speaking, if the size of the sliding window is increased to include more data, the accuracy of anomaly detection will increase as well (however, the latency also increases, so a balance must be found).

The function establishes a "normal" range for the data and then uses it to identify anomalies and assign a rating. It works like this: as the function continues to monitor the flow of data, the algorithm establishes a normal range of values, then compares new values against those norms. The result is a score for each value, expressed as a percentage that determines the confidence level that the given value is anomalous. As you may expect, low confidence levels can be ignored, but you may wonder what percentage confidence value is acceptable. In your query, you will set this tipping point at 95%.

There are always complications, like when there are gaps in the data (the conveyor belt stops for a while, perhaps). The algorithm handles voids in the data by imputing values.

**Note:** In statistics, imputation is the process of replacing missing data with substituted values. You can learn more about imputations [here](#).

Spikes and dips in telemetry data are temporary anomalies. However, since you are simulating vibration data using sine waves, you can expect a period of "normal" values followed by a high or low value that triggers an anomaly alert. An operator may look for a cluster of anomalies occurring in a short time span, which would signal that something is wrong.

There are other built-in ML models, such as a model for detecting trends. We don't include these models as part of this module, but the student is encouraged to investigate further.

## **Visualize data using Power BI**

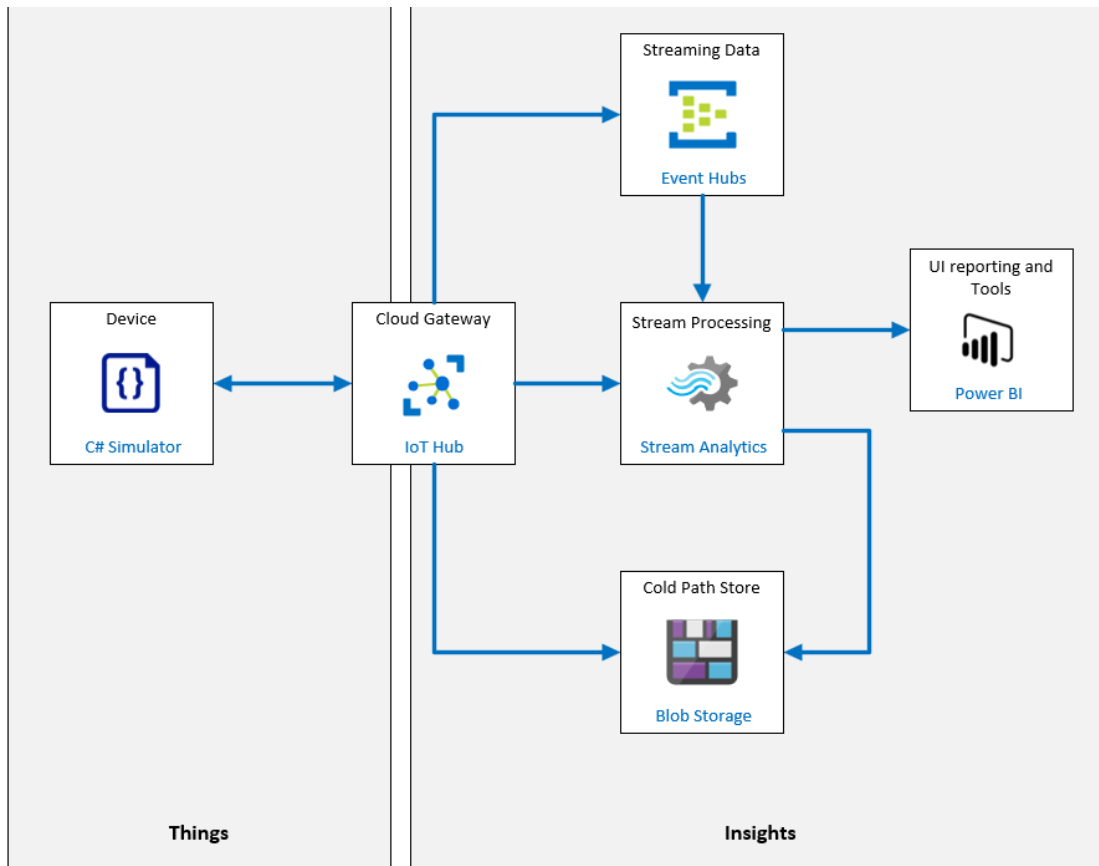
Visualizing numerical data, especially volumes of it, is a challenge in itself. How can we alert a human operator of the sequence of anomalies that infer something is wrong?

The solution we use in this module is to use some built-in functionality of Power BI along with the ability of Azure Stream Analytics to send data in a real-time format that Power BI can ingest.

We use the dashboard feature of Power BI to create a number of tiles. One tile contains the actual vibration measurement. Another tile is a gauge, showing from 0.0 to 1.0 the confidence level that the value is an anomaly. A third tile indicates if the 95% confidence level is reached. Finally, the fourth tile shows the number of anomalies detected over the past hour. By including time as the x-axis, this tile makes it clear if a clutch of anomalies were detected in short succession as they will be clustered together horizontally.

The fourth tile allows you to compare the anomalies with the red text in the telemetry console window. Is there a cluster of anomalies being detected when forced, or increasing, or both, vibrations are in action?

The following resources will be created:





## In This Lab

In this lab, you will complete the following activities:

- Verify that the lab prerequisites are met (that you have the required Azure resources)
- Sign-up for Power BI
- Generate Telemetry from Simulated Device
- Analyze Telemetry in Real-Time
- Create an Azure Event Hubs service
- Create a Real-time Message Route
- Add a Telemetry Route to IoT hub
- Create a Power BI dashboard to visualize data anomalies

Let's create the Event Hub, create the second route, update the SQL query, create a Power BI dashboard, and let it all run!

# Lab Instructions

## Exercise 1: Verify Lab Prerequisites

This lab assumes the following Azure resources are available:

Resource Type	Resource Name
Resource Group	rg-az220
IoT Hub	iot-az220-training-{your-id}
Device ID	sensor-v-3000
Storage Account Name	vibrationstore{your-id}
Storage Container	vibrationcontainer
Event Hub Namespace	vibrationNamespace{your-id}
Event Hub Name	vibrationeventhubinstance
Streaming Job	vibrationJob
Streaming Job Input	vibrationInput
Streaming Job Output	vibrationOutput
Streaming Job Transformation	VibrationJobTransformation

If these resources are not available, you will need to run the **lab08-setup.azcli** script as instructed below before moving on to Exercise 2. The script file is included in the GitHub repository that you cloned locally as part of the dev environment configuration (lab 3).

The **lab08-setup.azcli** script is written to run in a **bash** shell environment - the easiest way to execute this is in the Azure Cloud Shell.

1. Using a browser, open the [Azure Shell](#) and login with the Azure subscription you are using for this course.

If you are prompted about setting up storage for Cloud Shell, accept the defaults.

2. Verify that the Azure Cloud Shell is using **Bash**.

The dropdown in the top-left corner of the Azure Cloud Shell page is used to select the environment. Verify that the selected dropdown value is **Bash**.

3. On the Azure Shell toolbar, click **Upload/Download files** (fourth button from the right).

4. In the dropdown, click **Upload**.
5. In the file selection dialog, navigate to the folder location of the GitHub lab files that you downloaded when you configured your development environment.

In *Lab 3: Setup the Development Environment*, you cloned the GitHub repository containing lab resources by downloading a ZIP file and extracting the contents locally. The extracted folder structure includes the following folder path:

- Allfiles
- Labs
  - 08-Visualize a Data Stream in Power BI
  - Setup

The lab08-setup.azcli script file is located in the Setup folder for lab 7.

6. Select the **lab08-setup.zip** file, and then click **Open**.

A notification will appear when the file upload has completed.

7. To verify that the correct file has uploaded in Azure Cloud Shell, enter the following command:

```
bash ls
```

The `ls` command lists the content of the current directory. You should see the lab08-setup.azcli file listed.

8. To create a directory for this lab that contains the setup script and then move into that directory, enter the following Bash commands:

```
bash mkdir lab8 unzip lab08-setup.zip -d ./lab8 cd lab8
```

9. To ensure that **lab08-setup.azcli** and **Create-Job.ps1** have the execute permission, enter the following command:

```
bash chmod +x lab08-setup.azcli chmod +x Create-Job.ps1
```

10. On the Cloud Shell toolbar, to enable access to the lab08-setup.azcli file, click **Open Editor** (second button from the right - { }).
11. In the **FILES** list, to expand the lab8 folder and open the script file, click **lab8**, and then click **lab08-setup.azcli**.

The editor will now show the contents of the **lab08-setup.azcli** file.

12. In the editor, update the `{your-id}` and `{your-location}` assigned values.

Referencing the sample below as an example, you need to set `{your-id}` to the Unique ID you created at the start of this course - i.e. **cah191211**, and set `{your-location}` to the location that makes sense for your resources.

```
```bash
```

# !/bin/bash

## Change these values!

```
YourID="{your-id}" Location="{your-location}" ``
```

**Note:** The `{your-location}` variable should be set to the short name for the region where you are deploying all of your resources. You can see a list of the available locations and their short-names (the **Name** column) by entering this command:

```
``bash az account list-locations -o Table
```

```
DisplayName Latitude Longitude Name
```

---

```
East Asia 22.267 114.188 eastasia Southeast Asia 1.283 103.833  
southeastasia Central US 41.5908 -93.6208 centralus East US 37.3719  
-79.8164 eastus East US 2 36.6681 -78.3889 eastus2 ``
```

13. In the top-right of the editor window, to save the changes made to the file and close the editor, click **...**, and then click **Close Editor**.

If prompted to save, click **Save** and the editor will close.

**Note:** You can use **CTRL+S** to save at any time and **CTRL+Q** to close the editor.

14. To create the resources required for this lab, enter the following command:

```
bash ./lab08-setup.azcli
```

This script will take a few minutes to run. You will see output as each step completes:

```
``text Create resource group rg-az220 - Success Create IoT Hub iot-  
az220-training-dm200420 - Success Create device sensor-v-3000 -  
Success Create storage account vibrationstoredm200420 - Success Create  
storage container vibrationcontainer - Success Create IoT Hub endpoint  
vibrationLogEndpoint - Success Create IoT Hub route  
vibrationLoggingRoute - Success Create routing-endpoint
```

vibrationTelemetryEndpoint - Success Create iot hub route - Success  
Setup Azure Streaming Job - launching PowerShell

MOTD: Download scripts from PowerShell Gallery: Install-Script

# Table of Contents

You have completed the registration, configuration, and deprovisioning as part of the IoT devices life cycle with 162 Device Provisioning Service.

Want to know what is installed on the lab VM?	1
{{ activity.lab.title }}{% if activity.lab.type %} - {{ activity.lab.type }}{% endif %}	1
{{ activity.demo.title }}	1