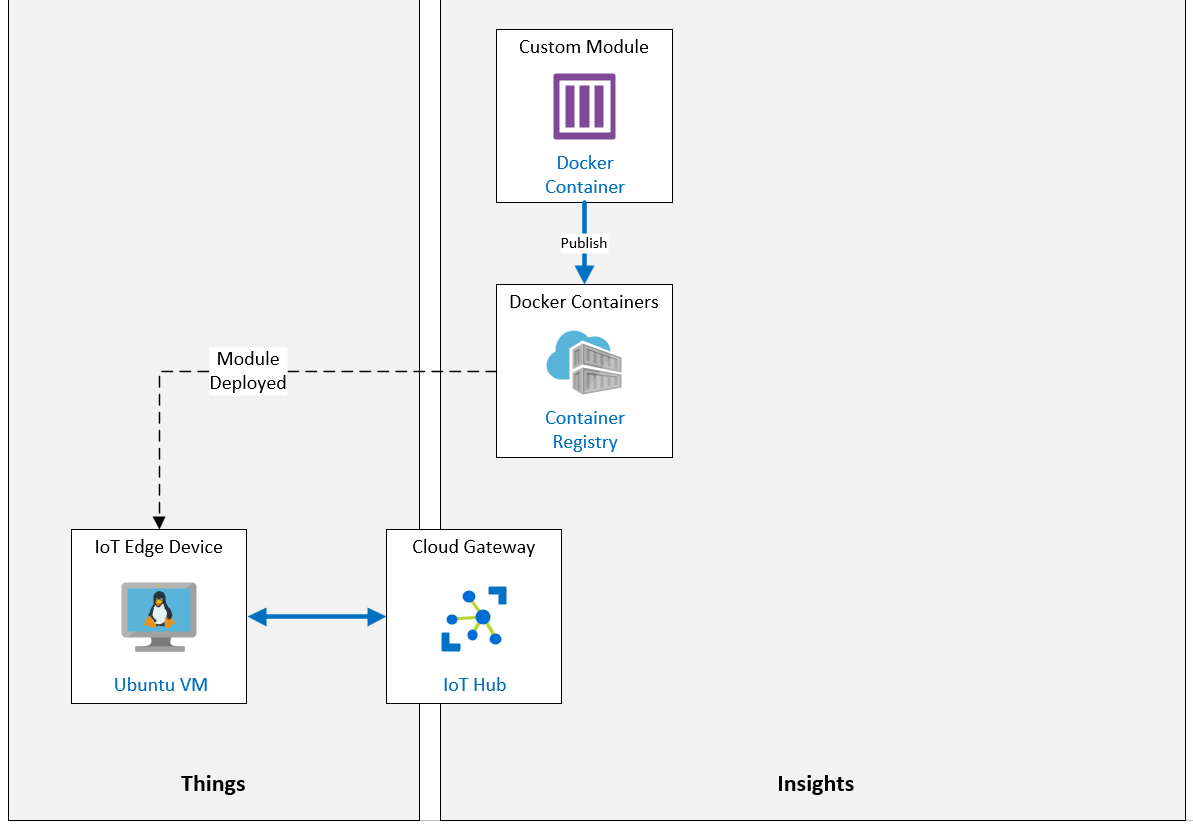
* [Exercise 2: Install Azure IoT EdgeHub Dev Tool](https://microsoftlearning.github.io/AZ-220-Microsoft-Azure-IoT-Developer/Instructions/Labs/LAB_AK_13-deploy-and-debug-custom-azure-iot-edge-module.html" \l "exercise-2-install-azure-iot-edgehub-dev-tool)
* [Exercise 3: Create an Azure Container Registry](https://microsoftlearning.github.io/AZ-220-Microsoft-Azure-IoT-Developer/Instructions/Labs/LAB_AK_13-deploy-and-debug-custom-azure-iot-edge-module.html#exercise-3-create-an-azure-container-registry)
* [Exercise 4: Create Custom Edge Module in C#](https://microsoftlearning.github.io/AZ-220-Microsoft-Azure-IoT-Developer/Instructions/Labs/LAB_AK_13-deploy-and-debug-custom-azure-iot-edge-module.html#exercise-4-create-custom-edge-module-in-c)
* [Exercise 5: Debug in Attach Mode with IoT Edge Simulator](https://microsoftlearning.github.io/AZ-220-Microsoft-Azure-IoT-Developer/Instructions/Labs/LAB_AK_13-deploy-and-debug-custom-azure-iot-edge-module.html#exercise-5-debug-in-attach-mode-with-iot-edge-simulator)
* [Exercise 6: Deploy the IoT Edge Solution](https://microsoftlearning.github.io/AZ-220-Microsoft-Azure-IoT-Developer/Instructions/Labs/LAB_AK_13-deploy-and-debug-custom-azure-iot-edge-module.html#exercise-6-deploy-the-iot-edge-solution)

You need to **create and deploy a custom IoT Edge module** that **counts** the **number of packages detected by the other module**.

* **Visual Studio Code** with the following extensions installed:
  + [Azure IoT Tools](https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools) by Microsoft
  + [C#](https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp) by Microsoft
  + [Docker](https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker)
* **Docker Community Edition** installed on development machine, with Docker Client version 18.03.0 or later
  + [Download Docker Desktop for Mac and Windows](https://www.docker.com/products/docker-desktop)

**Important**: Due to the January 13, 2020 removal of Azure Container Registry support for any TLS versions before TLS version 1.2, you must be running Docker Client 18.03.0 or later.

[](https://microsoftlearning.github.io/AZ-220-Microsoft-Azure-IoT-Developer/Instructions/Labs/media/LAB_AK_13-architecture.png)

1. Create the Container Registry
2. Create and customize an Edge module
3. Deploy modules to Edge 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-testing-0001 |

Exercise 2: Install Azure IoT EdgeHub Dev Tool

1. Verify that you have Python 3.8 installed in your development environment.
2. At the command prompt, to install the package manager for Python (Pip), enter

curl [https://bootstrap.pypa.io/get-pip.py -o get-pip.py](https://bootstrap.pypa.io/get-pip.py%20-o%20get-pip.py)

python get-pip.py

**Pip** is **required** to install the **Azure IoT EdgeHub Dev Tool** on your development machine.

1. To install the Azure IoT EdgeHub Dev Tool, enter the following command:

Python –m pip install iotedgehubdev --user

**Python environment** and **installed these tools**, you are ready to **create an Azure Container Registry** which will be used to **store** our **custom IoT Edge module.**

Exercise 3: Create an Azure Container Registry

**Azure Container Registry provides storage of private Docker images** for **container deployments**. The service is a managed, private Docker registry service based on the open-source Docker Registry 2.0. **Azure Container Registry** is used to **store** and **manage** your **private Docker container images.**

In this exercise, you will use the Azure portal to create a new Azure Container Registry resource.

Task 1: Create the container registry

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.

1. On the Azure portal menu, click **+ Create a resource**.
2. On the **New** blade, in the **Search the Marketplace** textbox, type **container registry** and then press **Enter**.
3. On the **Marketplace** blade, click **Container Registry**.
4. On the **Container Registry** blade, click **Create**.
5. On the **Create container registry** blade, under **Subscription**, ensure that the subscription you are using for this course is selected.
6. In the **Resource group** dropdown, click **rg-az220**.
7. On the **Create container registry** blade, under **Registry name**, enter a globally unique name.

To provide a globally unique name, enter **acraz220testing0001**.

For example: **acraz220trainingcah191204**

The **name of your Azure Container Registry** must be **globally unique** because it is a publicly accessible resource that you must be able to access from any IP connected device.

Consider the following when you specify a unique name for your new Azure Container Registry:

* + As mentioned above, the name of the registry must be unique across all of Azure. This is true because the value assigned to the name will be used in the domain name assigned to the service. Since Azure enables you to connect from anywhere in the world to your registry, it makes sense that all container registries must be accessible from the Internet using the resulting domain name.
  + The registry name cannot be changed once the Azure Container Registry has been created. If you do need to change the name, you’ll need to create a new Container Registry, re-deploy your container images, and delete your old Container Registry.

**Note**: Azure will ensure that the name you enter is unique. If the name that you enter is not unique, Azure will display an asterisk at the end of the name field as a warning. You can append the name suggested above with 01 or 02 as necessary to achieve a globally unique name.

1. In the **Location** dropdown, choose the same Azure region that was used for the resource group.
2. Under **Availability zones**, leave the checkbox **unchecked**.

Availability zones is a high-availability offering that provides resiliency and high availability to a container registry in a specific region.

1. In the **SKU** dropdown, ensure that **Standard** is selected.

Azure Container Registry is available in multiple service tiers, known as SKUs. These SKUs provide predictable pricing and several options for aligning to the capacity and usage patterns of your private Docker registry in Azure.

1. At the bottom of the blade, click **Review + create**.

The settings you have entered will be validated.

1. To complete the creation of the Container Registry, at the bottom of the blade, click **Create**.

Task 2: Connect Docker to the container registry

1. On your dashboard, refresh your Resources tile, and then click **acraz220training{your-id}**.
2. On the left side navigation menu, under **Settings**, click **Access keys**.
3. Under **Admin user**, click **Enable**.

This option will enable you to Docker login to the Azure Container Registry service using the registry name as the username and admin user access key as the password.

1. Make a record of the following values:
   * **Login server**
   * acraz220testing0001.azurecr.io
   * **Username**
   * acraz220testing0001
   * **password**
   * =N3d79SbHuzXL=FUmYcZAK93riiKnA=l
   * BaawdLrh7kaWtr7rhDW6IS/vienvViJy

By default, the admin Username will match the ACR name - **acraz220training{your-id}**

This information will enable you to authenticate to the new registry, which is required to perform Docker operations in the upcoming steps.

1. Open a command prompt, and then enter the following command:

CodeCopy

docker login <loginserver>

docker login acraz220testing0001.azurecr.io

Replace <loginserver> with the name you recorded, and enter the username and password you recorded when prompted. For example:

CodeCopy

docker login az220acrcah191204.azurecr.io

This command will record your credentials in the local Docker client configuration file ($HOME/.docker/config.json) or your operating system’s secure credential storage mechanism (depending on the Docker configuration) for future use by the Docker toolset.

Now that you have created the Azure Container Registry and authenticated your local machine against it, you can create a custom IoT Edge Module container that will be stored in the registry.

Exercise 4: **Create Custom Edge Module in C#**

In this exercise, you will create an Azure IoT Edge Solution that contains a custom Azure IoT Edge Module written in C#.

Task 1: Create the solution

1. Open **Visual Studio Code.**
2. On the **View** menu, to open the Visual Studio Command Palette, click **Command Palette**.
3. At the command prompt, type **Azure IoT Edge: New** and then click **Azure IoT Edge: New IoT Edge Solution**.
4. Browse to the folder where you want to create the new solutions, and then click **Select folder**.
5. When prompted for a solution name, enter **EdgeSolution**

This name will be used as the directory name for the new **IoT Edge Solution** that will be created.

1. When prompted to select a module template, click **C# Module**.

This will define C# as the development language for the custom IoT Edge Module added to the solution.

1. When prompted for the name of the custom IoT Edge Module, enter **ObjectCountingModule**

This will be the name of the new IoT Edge Module that is being created.

1. When prompted for name of the Docker image repository for the module, update the placeholder value as follows:

Replace the localhost:5000 part of the default localhost:5000/objectcountingmodule repository location with the name of the Azure Container Registry server - similar to acraz220training{your-id}.azurecr.io

This will be the Docker repository where the IoT Edge Module docker image will be published.

The Docker image repository location follows the format shown below:

CodeCopy

<acr-name>.azurecr.io/<module-name>

Be sure to replace the placeholders with the appropriate values:

* + <acr-name>: Replace with the name of the Azure Container Registry service.
  + <module-name>: Replace with the name of the custom Azure IoT Edge Module that’s being created.

**Note**: The default Docker image repository in Visual Studio Code is set to localhost:5000/<your module name>. If you were to use a local Docker registry for testing, then **localhost** is fine.

**Important**: Make sure to remove any reference to port 5000 from your ACR references! That port is used for a local Docker repository but it not used in the ACR case.

1. Wait for Visual Studio Code to create the solution.

Once the new **IoT Edge Solution** has been created, Visual Studio Code will open the solution.

**Note**: If Visual Studio Code prompts you to load required resources or C# extension, click **Yes**

**Note**: If Visual Studio Code prompts you to configure the **.env** file, select **Yes** and proceed to Task 2 below.

Task 2: Configure the solution

1. Take a moment to review the contents of the **Explorer** pane.

Notice the files and directories that were created as part of the new IoT Edge Solution.

1. In the **Explorer** pane, to open the .env file, click **.env**.

The .env file is located in the root directory of the IoT Edge Solution. This is where the username and password are configured for accessing your Docker registry.

**Note:** Visual Studio Code may have already opened this file for you.

The username and password are stored in this file using the following format:

CodeCopy

CONTAINER\_REGISTRY\_USERNAME\_<registry-name>=<registry-username>

CONTAINER\_REGISTRY\_PASSWORD\_<registry-name>=<registry-password>

The placeholders above are defined as follows:

* + <registry-name>: The name of your Docker registry.
  + <registry-username>: The username to use for accessing your Docker registry.
  + <registry-password>: The password to use for accessing your Docker registry.

Within your version of .env file, notice that the <registry-name> has already been added to the configuration values. The value that has been added should match the name of the Docker registry that you specified when creating the IoT Edge Solution.

**Note**: You may wonder why you ran docker login before when you’re supplying the same credentials here. At the time when this lab was written, the Visual Studio Code tools do not automatically perform the docker login step with these credentials; they are only used to supply the credentials to the Edge Agent later as part of the deployment template.

1. Within the .env file, replace the placeholder values with the username and password values that you saved earlier.

Replace the <registry-username> placeholder with the **Registry name** (*aka Username*) of the Azure Container Registry that was previously created. Replace the <registry-password> placeholder with the **password** for the Azure Container Registry.

**Note**: The Azure Container Registry **Username** and **password** values can be found by accessing the **Access keys** pane for the **Azure Container Registry** service within the Azure portal, if you did not record them earlier.

1. Save the updated **.env** file.
2. In the **Explorer** pane, to open the deployment.template.json file, click **deployment.template.json**.

The deployment.template.json file is located in the root IoT Edge Solution directory. This file is the *deployment manifest* for the IoT Edge Solution. The deployment manifest **tells an IoT Edge device** (or a group of devices) which **modules to install and how to configure them.** The deployment manifest includes the ***desired properties*** for each module twin. IoT edge devices report back the ***reported properties*** for each module.

Two modules are required in every deployment manifest; $edgeAgent and $edgeHub. These modules are part of the IoT Edge runtime that manages the IoT Edge devices and the modules running on it.

1. **$edgeAgent** element:
   * **systemModules** - This defines Docker images to use for
   * **modules** - This defines the various modules that will be deployed and run on the IoT Edge device (or a group of devices).
2. Notice that within the**modules** section for the $edgeAgent, there are two modules defined:
   * **ObjectCountingModule**: This is the custom IoT Edge Module that is being created as part of this new IoT Edge Solution.
   * **SimulatedTemperatureSensor**: This defines the Simulated Temperature Sensor module to be deployed to the IoT Edge device.
3. Notice the $edgeHub section of the deployment manifest.

This section **defines the desired properties** (via properties.desired element) that includes the **message routes for communicating** messages between the **IoT Edge Modules** and finally **to Azure IoT Hub service**.

CodeCopy

"$edgeHub": {

"properties.desired": {

"schemaVersion": "1.0",

"routes": {

"ObjectCountingModuleToIoTHub": "FROM /messages/modules/ObjectCountingModule/outputs/\* INTO $upstream",

"sensorToObjectCountingModule": "FROM /messages/modules/SimulatedTemperatureSensor/outputs/temperatureOutput INTO BrokeredEndpoint(\"/modules/ObjectCountingModule/inputs/input1\")"

},

...

}

}

The sensorToObjectCountingModule route is configured to route messages from the SimulatedTemperatureSensor (via /messages/modules/SimulatedTemplaratureSensor/outputs/temperatureOutput) module to the custom ObjectCountingModule module (via BrokeredEndpoint(\"/modules/ObjectCountingModule/inputs/input1\")").

The ObjectCountingModuleToIoTHub route is configured to route messages that are sent out from the custom ObjectCountingModule module (via /messages/modules/SimulatedTemperatureSensor/outputs/temperatureOutput) to the Azure IoT Hub service (via $upstream).

1. In Visual Studio Code, on the **View** menu, click **Command Palette**
2. At the command prompt, type **Azure IoT Edge: Set Default** and then click **Azure IoT Edge: Set Default Target Platform for Edge Solution**.
3. To select the target platform, click **amd64**.

This target platform needs to be set to the hardware platform architecture of the IoT Edge Device.

**Note**: Since you are using the **IoT Edge on Ubuntu** Linux VM, the amd64 option is the appropriate choice. For a Windows VM, use windows-amd64, and for modules that will be running on an ARM CPU architecture, you would choose the arm32v7 option.

Task 3: Review the module code

1. In the **Explorer** pane, to expand the /modules/ObjectCountingModule directory, click **modules**.

Notice that this directory contains the source code files for the new IoT Edge Module being developed.

1. In the **Explorer** pane, to open the /modules/ObjectCountingModule/Program.cs file, click **Program.cs**.

This file contains the template source code for the newly created custom IoT Edge Module. This code provides a starting point for creating custom IoT Edge Modules.

1. In the Program.cs file, locate the static async Task Init() method, and then take a minute to review the code.

This method initializes the ModuleClient for handling messages sent to the module, and sets up the callback to receive messages. Read the code comments within the code for this method and notice what each section of code does.

1. Locate the static async Task<MessageResponse> PipeMessage( method, and then take a minute to review the code.

This method is called whenever the module is sent a message from the EdgeHub. The current state of the source code within this method receives messages sent to this module and pipes them out to the module output, without any change. Read through the code within this method and notice what it does.

Also, within the PipeMessage method, notice the following lines of code and what they do:

The following line of code within the method increments a counter that counts the number of messages sent to the module:

C#Copy

int counterValue = Interlocked.Increment(ref counter);

The following lines of code within the method write out to the Module’s Console a message that contains the total number of messages received by the Module, along with the current message’s body as JSON.

C#Copy

byte[] messageBytes = message.GetBytes();

string messageString = Encoding.UTF8.GetString(messageBytes);

Console.WriteLine($"Received message: {counterValue}, Body: [{messageString}]");

We have now created and configured a sample custom module. Next, we will debug it in the IoT Edge Simulator.

Exercise 5: Debug in Attach Mode with IoT Edge Simulator

In this exercise, you will build and **run a custom IoT Edge Module solution** using the IoT Edge Simulator from **within Visual Studio Code.**

Task 1: Create a test IoT Edge device

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.

1. On your Resource group tile, click **iot-az220-testing-0001**
2. On the left hand navigation menu, under **Settings**, click **Shared access policies**.
3. In the list of policies, click **iothubowner**.

**Important**: The Edge Simulator requires a privileged role for configuration. You would not use such a privileged role for normal use cases.

1. In the **iothubowner** pane, **copy** the value for **Connection string–primary key**.

Record this value, as you will need it below.

1. On the left hand navigation menu, under **Automatic Device Management**, click **IoT Edge**.

This pane allows you to manage the IoT Edge devices connected to the IoT Hub.

1. At the top of the pane, click **Add an IoT Edge device**.
2. On the **Create a device** blade, under **Device ID**, enter **SimulatedDevice**

This is the device identity used for authentication and access control.

1. Under **Authentication type**, ensure that **Symmetric key** is selected.
2. Leave the **Auto-generate keys** box checked.

This will have IoT Hub automatically generate the Symmetric keys for authenticating the device.

1. Leave the other settings at their defaults, and then click **Save**.

Task 2: Configure the test module

1. Switch to the **Visual Studio Code** instance containing your IoT Edge solution.
2. In the **Explorer** pane, right-click **deployment.debug.template.json**, and then click **Build and Run IoT Edge Solution in Simulator**.

This file is the debugging deployment manifest file. It is located in the root directory of the IoT Edge Solution.

When the process begins, you will see a dialog open in the lower right corner of the windows that says, **Please setup iotedgehubdev first before starting simulator**.

1. When you see the prompt to **setup iotedgehubdev**, click **Setup**.
2. When prompted for the **IoT Hub Connection String**, enter the **Connection string–primary key** you noted earlier**. enter**
3. When prompted to **Select an IoT Edge Device**, click **SimulatedDevice**.

**Note**: If you get an **Unauthorized** error in the lower-right-hand corner, run the Azure IoT Hub: Set IoT Hub Connection String command from the Command Palette to reconfigure your simulator connection string, then run Azure IoT Edge: Setup IoT Edge Simulator from the command palette and try to select your device again.

**Note**: It is possible that you will be prompted for your Admin password on your local machine (in the Visual Studio Code **TERMINAL** window), particularly on Linux or macOS. Enter your password at the prompt and press **Enter**. The reason it might ask for your password is that the setup command for iotedgehubdev is being run using sudo as it requires elevated privileges.

Once the IoT Edge Simulator is set up successfully, a **Setup IoT Edge Simulator successfully** message will be displayed in the Visual Studio Code TERMINAL.

Now when you build and run the module in the IoT Edge Simulator, it will run as expected.

Task 3: Build and run the module

1. In the **Explorer** pane, right-click **deployment.debug.template.json**, and then click **Build and Run IoT Edge Solution in Simulator**.

**Note**: If you are on Windows and see a message in the Terminal that reads, in part, open //./pipe/docker\_engine: The system cannot find the file specified., Docker is likely not started, or running correctly. A Docker restart or even a full computer restart might be necessary.

**Note**: If you see a message that reads, in part, image operating system "linux" cannot be used on this platform, change your Docker configuration to support Linux containers. (Ask your instructor for assistance if necessary.)

**Note**: A build may take some time depending on what Docker images you have on your machine already and the speed of your Internet connection. The build includes downloading Docker images if not present and updating container instances if necessary.

1. Observe the build process reporting in your TERMINAL window.

It can take quite a few minutes to download and build everything that is required to simulate the IoT Edge device and run your modules, so be patient.

Notice that once the **IoT Edge Simulator** is running, the Modules that you built will begin sending message output that is reported to the TERMINAL window:

CodeCopy

SimulatedTemperatureSensor | 12/09/2019 15:05:08> Sending message: 4, Body: [{"machine":{"temperature":23.023276334173641,"pressure":1.2304998355387693},"ambient":{"temperature":20.56235126408858,"humidity":24},"timeCreated":"2019-12-09T15:05:08.4596891Z"}]

ObjectCountingModule | Received message: 4, Body: [{"machine":{"temperature":23.023276334173641,"pressure":1.2304998355387693},"ambient":{"temperature":20.56235126408858,"humidity":24},"timeCreated":"2019-12-09T15:05:08.4596891Z"}]

ObjectCountingModule | Received message sent

SimulatedTemperatureSensor | 12/09/2019 15:05:13> Sending message: 5, Body: [{"machine":{"temperature":23.925331861560853,"pressure":1.3332656551145274},"ambient":{"temperature":20.69443827876562,"humidity":24},"timeCreated":"2019-12-09T15:05:13.4856557Z"}]

ObjectCountingModule | Received message: 5, Body: [{"machine":{"temperature":23.925331861560853,"pressure":1.3332656551145274},"ambient":{"temperature":20.69443827876562,"humidity":24},"timeCreated":"2019-12-09T15:05:13.4856557Z"}]

ObjectCountingModule | Received message sent

Notice the output from the **ObjectCountingModule** contains the text Received message: # where # is the total message count that has been received by the custom **ObjectCountingModule** IoT Edge Module that was created.

1. With the IoT Edge Simulator still running, open the Azure portal, and then open the Cloud Shell.
2. At the Cloud Shell command prompt, to monitor the messages being sent to Azure IoT Hub from the SimulatedDevice running in the IoT Edge Simulator on your local machine, enter the following command:

CodeCopy

**az iot hub monitor-events --hub-name "iot-az220-testing-0001"**

Be sure to replace the iot-az220-training-{your-id} value in the above command with the name of your Azure IoT Hub service.

1. Observe the output displayed in the Cloud Shell.

With everything still running, notice the output of the previous command in the Cloud Shell will display a JSON representation of the messages being received by the Azure IoT Hub.

The output should look similar to the following:

CodeCopy

{

"event": {

"origin": "SimulatedDevice",

"payload": "{\"machine\":{\"temperature\":88.003809452058647,\"pressure\":8.6333453806142764},\"ambient\":{\"temperature\":21.090260561364826,\"humidity\":24},\"timeCreated\":\"2019-12-09T15:16:32.402965Z\"}"

}

}

{

"event": {

"origin": "SimulatedDevice",

"payload": "{\"machine\":{\"temperature\":88.564600328362815,\"pressure\":8.6972329488008278},\"ambient\":{\"temperature\":20.942187817041848,\"humidity\":25},\"timeCreated\":\"2019-12-09T15:16:37.4355705Z\"}"

}

}

1. Compare the output in the Visual Studio Code TERMINAL with the output in the Cloud Shell.

Notice that even though there are 2 IoT Edge Modules running on the IoT Edge Device that generate messages, there is still only a single copy of each message getting sent to Azure IoT Hub. The IoT Edge Device has a message pipeline defined where messages from the SimulatedTemperatureSensor are piped to the ObjectCountingModule which then sends messages out to the Azure IoT Hub.

1. To stop monitoring Azure IoT Hub events, press **Ctrl + C** within the Azure Cloud Shell.

Task 4: Debug the module

1. Switch to your Visual Studio Code window.
2. On the left side toolbar, to open the Visual Studio Code debugger view, click **Run**.

The Run button is the forth button down from the top, and includes a bug-like shape on the icon.

1. At the top of the **Run** pane, in the dropdown, ensure that **ObjectCountingModule Remote Debug (.NET Core)** is selected.
2. To the left of the dropdown, click **Start Debugging**.

You could also press **F5** to start debugging

1. When prompted to **Select the process to attach to**, click **dotnet ObjectCountingModule.dll**.
2. On the left side toolbar, to change to the file Explorer view, click **Explorer**
3. In the EXPLORER pane, to open the /modules/ObjectCountingModule/Program.cs source code file, click **Program.cs**.
4. In the code editor, locate the static async Task<MessageResponse> PipeMessage( method.
5. Select the static async Task<MessageResponse> PipeMessage( code line, and then, to set a breakpoint, press **F9**.

Visual Studio Code enables you to set a breakpoint within your code by clicking on the line and pressing the **F9** key.

1. Notice that execution stops at the breakpoint that is set, and the editor highlights that specific line of code.
2. To open the Visual Studio Code debugger view, on the left toolbar, click **Run**.
3. Notice the variables listed in the left panel.
4. To resume execution, click **Continue**.

You can also press **F5** to resume.

1. Notice that each time the breakpoint it hit, execution stops.
2. To stop debugging, click the **Disconnect** button, or press **Shift + F5**.
3. To stop the IoT Edge Simulator, open the **Command Palette**, and then select the **Azure IoT Edge: Stop IoT Edge Simulator** option.

Now that the module has been created and tested in the IoT Edge simulator, it is time to deploy it to the cloud.

Exercise 6: Deploy the IoT Edge Solution

In this exercise, you will **build and publish the custom IoT Edge Module into the Azure** Container Registry (ACR) service. Once published to ACR, the custom module will then be made available to be deployed to any IoT Edge Device.

Task 1: Publish module to the Azure Container Registry

1. Open the Visual Studio Code window containing your EdgeSolution project.
2. In the **Explorer** view, to open the .env file, click **.env**.

The .env file is located in the root directory of the IoT Edge Solution.

1. Ensure that the credentials for the Azure Container Registry have been set.

When set correctly, the CONTAINER\_REGISTRY\_USERNAME\_<acr-name> key will have it’s value set to the Azure Container Registry service name, and the CONTAINER\_REGISTRY\_PASSWORD\_<acr-name> key will have it’s value set to the **Password** for the Azure Container Registry service. Keep in mind, the <acr-name> placeholders in the keys will be set to the ACR service name (is all lowercase) automatically when the IoT Edge Solution was created.

**The resulting .env** file contents will look similar to the following:

CodeCopy

**CONTAINER\_REGISTRY\_USERNAME\_acraz220trainingcah191204=acraz220trainingcah191204**

**CONTAINER\_REGISTRY\_PASSWORD\_acraz220trainingcah191204=Q8YErJFCtqSe9C7BWdHOKEXk+C6uKSuK**

1. In the **Explorer** view, right-click **deployment.template.json**, and then click **Build and Push IoT Edge Solution**.

The status of the Build and Push IoT Edge Solution operation is displayed within the Visual Studio Code **TERMINAL** window. Once the process completes, the custom ObjectCountingModule IoT Edge Module will have been built, and then the Docker image for the IoT Edge Module will be published to the Azure Container Registry service.

1. **Switch to your Azure portal window.**

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.

1. On your Resource group tile, to open your Azure Container Registry (ACR) service, click **acraz220training{your-id}**.
2. On the left side navigation menu, under **Services**, click **Repositories**.
3. On the **Repositories** pane, notice that the objectcountingmodule repository now exists within the ACR service.

This was created when the custom ObjectCountingModule IoT Edge Module was published from within Visual Studio Code.

**Note**: If the repository is not present, review the output from the Push action and ensure that you did not leave references to the :5000 port number with your ACR references; you can do an **Edit**, **Find in Files** to confirm this. You should also validate your credentials in the .env file and validate that you performed the docker login step earlier in the lab.

1. Under **Repositories**, click **objectcountingmodule**.
2. On the **objectcountingmodule** blade, under **Tags**, notice that there is a tag named **0.0.1-amd64**.
3. To open a details pane for this Tag, click **0.0.1-amd64**.

Notice the properties listed, including *Repository*, *Tag*, *Tag creation date*, *Tag last updated date*, and other properties displaying information about the Tag.

1. Save a copy of the values for the **Repository** and **Tag** properties.

To copy the values, you can click on the **Copy to clipboard** button located to the right of the displayed values.

You will need the Repository and Tag names to pull down this specific version of the Docker image for the custom IoT Edge Module to run in an IoT Edge Device.

The format of the Docker image Repository and Tag names combined will be in the following format:

CodeCopy

<repository-name>:<tag>

Here’s an example of a full Docker image name for the objectcountingmodule IoT Edge Module:

CodeCopy

objectcountingmodule:0.0.1-amd64

docker pull acraz220testing0001.azurecr.io/objectcountingmodule:0.0.1-amd64

**Task 2: Configure an IoT Edge device to use the module**

1. Navigate to your Azure IoT Hub resource.

With the custom objectcountingmodule IoT Edge Module published to Azure Container Registry (ACR), the next step is to **create a new IoT Edge Device within IoT Hub** and configure it to run the new custom IoT Edge Module.

1. On the **iot-az220-training-{your-id}** blade, on the left side navigation menu under **Automatic Device Management**, click **IoT Edge**.
2. On the **IoT Edge** pane, at the top of the pane, click **Add an IoT Edge device**.
3. On the **Create a device** blade, under **Device ID**, enter **objectcountingdevice**
4. Under **Authentication type**, ensure that **Symmetric key** is selected, and ensure that the **Auto-generate keys** checkbox is selected.

For this unit, we’ll keep the IoT Edge Module registration simple by choosing *Symmetric key* authentication. With the *Auto-generate keys* option selected, the IoT Hub with automatically generate authentication keys for this device.

1. At the bottom of the blade, click **Save**.
2. On the **IoT Edge** pane, under **Device ID**, click **objectcountingdevice**.
3. At the top of the **objectcountingdevice** blade, click **Set Modules**.
4. On the **Set modules on device: objectcountingdevice** blade, under **Container Registry Settings**, enter the following values:
   * **Name**: Enter the **Registry name** of the **Azure Container Registry** (e.g. acraz220trainingcah191204)
   * **Address**: Enter the **Login server** (or DNS name) of the Azure Container Registry service (ex: acraz220trainingcah191204.azurecr.io)
   * **User Name**: Enter the **Username** for the Azure Container Registry service
   * **Password**: Enter the **password** for the Azure Container Registry service

**Note**: The Azure Container Registry (ACR) service *Registry name*, *Login server*, *Username*, and *Password* can be found on the **Access keys** pane for the service.

1. On the **Set modules on device: objectcountingdevice** blade, under **IoT Edge Modules**, click **Add**, and then click **IoT Edge Module**.
2. On the **Add IoT Edge Module** pane, under **IoT Edge Module Name**, enter **objectcountingmodule**
3. Under **Module Settings**, to enter the full URI and tagged name of the Docker image for the custom IoT Edge Module, use the following format:

CodeCopy

<container-registry-login-server>/<repository-name>:<tag>

Be sure to replace the placeholders within the above **Image URI** format with the appropriate values:

* + <container-registry-login-server> \* The **Login server**, or DNS name, for the Azure Container Registry service.
  + <repository-name> \* The **Repository name** for the Custom IoT Edge Module’s Docker image, that was copied previously.
  + <tag> \* The **Tag** for the Custom IoT Edge Module’s Docker image, that was copied previously.

The resulting **Image URI** to be entered into the field will be similar to the following:

CodeCopy

acraz220trainingcah191204.azurecr.io/objectcountingmodule:0.0.1-amd64

1. Leave the rest of the settings at their defaults, and then click **Add**.
2. On the **Set modules on device: objectcountingdevice** blade, at the bottom of the blade, click **Next: Routes >**.
3. On the **Routes** tab, review the default route settings.

The editor will display the configured default route for the IoT Edge Device. At this time, it should be configured with a route that sends all messages from all modules to Azure IoT Hub:

* + Name: **route**
  + Value**: FROM /messages/\* INTO $upstream**

1. To the right of the default route, click **Remove route**.
2. Add the following two routes:

| NAME | VALUE |
| --- | --- |
| AllMessagesToObjectCountingModule | FROM /\* INTO BrokeredEndpoint("/modules/objectcountingmodule/inputs/input1") |
| ObjectCountingModuleToIoTHub | FROM /messages/modules/objectcountingmodule/outputs/\* INTO $upstream |

1. Review the value assigned to the **AllMessagesToObjectCountingModule** route.

This route specifies the **Source** value of /\*. This applies the route to all device-to-cloud messages or twin change notifications from any module or leaf device.

This route specifies the **Destination** value of BrokeredEndpoint("/modules/objectcountingmodule/inputs/input1"). This sends all messages from the Source of this route to the objectcountingmodule IoT Edge Module’s input.

1. Review the value assigned to the **ObjectCountingModuleToIoTHub** route.

This route specifies the **Source** value of /messages/modules/objectcountingmodule/outputs/\*. This applies the route to all messages output from the objectcountingmodule IoT Edge Module.

This route specifies the **Destination** value of $upstream. This sends all messages from the Source of this route to the Azure IoT Hub service within Microsoft Azure.

**Note**: For more information on the configuration of Message Routing with Azure IoT Hub and IoT Edge Module, reference the following links:

* + [Learn how to deploy modules and establish routes in IoT Edge](https://docs.microsoft.com/en-us/azure/iot-edge/module-composition)
  + [IoT Hub message routing query syntax](https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-routing-query-syntax)

1. At the bottom of the blade, click **Next: Review + create >**.
2. Review the Deployment Manifest for the device, and then click **Create**.

This completes the development of the objectcountingmodule custom IoT Edge Module. Now that an IoT Edge Device is registered, the modules specified and the routes configured, the objectcountingmodule is ready to be deployed once the associated IoT Edge Device is connected to the Azure IoT Hub as shown in previous labs.

[MicrosoftLearning/AZ-220-Microsoft-Azure-IoT-Developer](https://github.com/MicrosoftLearning/AZ-220-Microsoft-Azure-IoT-Developer" \t "_blank)