Debug Azure IoT Edge modules using Visual Studio Code

Article • 05/09/2023

Applies to: IoT Edge 1.4

> **Important**
>
> IoT Edge 1.4 is the **supported release**. If you are on an earlier release, see **Update IoT Edge**.

This article shows you how to use Visual Studio Code to debug IoT Edge modules in multiple languages. On your development computer, you can use Visual Studio Code to attach and debug your module in a local or remote module container.

This article includes steps for two IoT Edge development tools.

- *Azure IoT Edge Dev Tool* command-line tool (CLI). This tool is preferred for development.
- *Azure IoT Edge tools for Visual Studio Code* extension. The extension is in maintenance mode .

Use the tool selector button at the beginning of this article to select the tool version.

Visual Studio Code supports writing IoT Edge modules in the following programming languages:

- C# and C# Azure Functions
- C
- Python
- Node.js
- Java

Azure IoT Edge supports the following device architectures:

- AMD64
- ARM32v7
- ARM64

For more information about supported operating systems, languages, and architectures, see Language and architecture support.

You can also use a Windows development computer and debug modules in a Linux container using IoT Edge for Linux on Windows (EFLOW). For more information about using EFLOW for developing modules, see Tutorial: Develop IoT Edge modules with Linux containers using IoT Edge for Linux on Windows.

If you aren't familiar with the debugging capabilities of Visual Studio Code, see Visual Studio Code debugging .

Prerequisites

You can use a computer or a virtual machine running Windows, macOS, or Linux as your development machine. On Windows computers, you can develop either Windows or Linux modules. To develop Linux modules, use a Windows computer that meets the requirements for Docker Desktop .

To install the required tools for development and debugging, complete the Develop Azure IoT Edge modules using Visual Studio Code tutorial.

Install Visual Studio Code

To debug your module on a device, you need:

- An active IoT Hub with at least one IoT Edge device.
- A physical IoT Edge device or a virtual device. To create a virtual device in Azure, follow the steps in the quickstart for Linux.
- A custom IoT Edge module. To create a custom module, follow the steps in the Develop Azure IoT Edge modules using Visual Studio Code tutorial.

Debug a module with the IoT Edge runtime

In each module folder, there are several Docker files for different container types. Use any of the files that end with the extension **.debug** to build your module for testing.

When you debug modules using this method, your modules are running on top of the IoT Edge runtime. The IoT Edge device and your Visual Studio Code can be on the same machine, or more typically, Visual Studio Code is on the development machine and the IoT

Edge runtime and modules are running on another physical machine. To debug from Visual Studio Code, you must:

- Set up your IoT Edge device, build your IoT Edge modules with the **.debug** Dockerfile, and then deploy to the IoT Edge device.
- Update `launch.json` so that Visual Studio Code can attach to the process in a container on the remote machine. You can find this file in the `.vscode` folder in your workspace, and it updates each time you add a new module that supports debugging.
- Use Remote SSH debugging to attach to the container on the remote machine.

Build and deploy your module to an IoT Edge device

In Visual Studio Code, open the *deployment.debug.template.json* deployment manifest file. The deployment manifest describes the modules to be configured on the targeted IoT Edge device. Before deployment, you need to update your Azure Container Registry credentials and your module images with the proper `createOptions` values. For more information about createOption values, see How to configure container create options for IoT Edge modules.

1. If you're using an Azure Container Registry to store your module image, add your credentials to the *edgeAgent* > *settings* > *registryCredentials* section in **deployment.debug.template.json**. Replace **myacr** with your own registry name in both places and provide your password and **Login server** address. For example:

   JSON

   ```
   "modulesContent": {
   "$edgeAgent": {
     "properties.desired": {
       "schemaVersion": "1.1",
       "runtime": {
         "type": "docker",
         "settings": {
           "minDockerVersion": "v1.25",
           "loggingOptions": "",
           "registryCredentials": {
             "myacr": {
               "username": "myacr",
               "password": "<your_azure_container_registry_password>",
               "address": "myacr.azurecr.io"
             }
           }
   ```

```
      }
    },
...
```

2. Add or replace the following stringified content to the *createOptions* value for each system (edgeHub and edgeAgent) and custom module (for example, filtermodule) listed. Change the values if necessary.

```JSON
"createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":
[{\"HostPort\":\"5671\"}],\"8883/tcp\":
[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}}}"
```

   For example, the *filtermodule* configuration should be similar to:

```JSON
"filtermodule": {
"version": "1.0",
"type": "docker",
"status": "running",
"restartPolicy": "always",
"settings": {
    "image": "myacr.azurecr.io/filtermodule:0.0.1-amd64",
    "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":
[{\"HostPort\":\"5671\"}],\"8883/tcp\":
[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}}}"
}
```

Sign in to Docker

Provide your container registry credentials to Docker so that it can push your container image to storage in the registry.

1. Sign in to Docker with the Azure Container Registry credentials that you saved after creating the registry.

```Bash
docker login -u <Azure Container Registry username> -p <Azure Container
Registry password> <Azure Container Registry login server>
```

You may receive a security warning recommending the use of `--password-stdin`. While that's a recommended best practice for production scenarios, it's outside the scope of this tutorial. For more information, see the docker login reference.

2. Sign in to the Azure Container Registry. You may need to Install Azure CLI to use the `az` command. This command asks for your user name and password found in your container registry in **Settings** > **Access keys**.

Azure CLI

```
az acr login -n <Azure Container Registry name>
```

> **Tip**
>
> If you get logged out at any point in this tutorial, repeat the Docker and Azure Container Registry sign in steps to continue.

Build module Docker image

Use the module's Dockerfile to build the module Docker image.

Bash

```
docker build --rm -f "<DockerFilePath>" -t <ImageNameAndTag> "<ContextPath>"
```

For example, to build the image for the local registry or an Azure Container Registry, use the following commands:

Bash

```
# Build the image for the local registry

docker build --rm -f "./modules/filtermodule/Dockerfile.amd64.debug" -t
localhost:5000/filtermodule:0.0.1-amd64 "./modules/filtermodule"

# Or build the image for an Azure Container Registry

docker build --rm -f "./modules/filtermodule/Dockerfile.amd64.debug" -t
myacr.azurecr.io/filtermodule:0.0.1-amd64 "./modules/filtermodule"
```

Push module Docker image

Push your module image to the local registry or a container registry.

```
docker push <ImageName>
```

For example:

Bash

```
# Push the Docker image to the local registry

docker push localhost:5000/filtermodule:0.0.1-amd64

# Or push the Docker image to an Azure Container Registry
az acr login --name myacr
docker push myacr.azurecr.io/filtermodule:0.0.1-amd64
```

Deploy the module to the IoT Edge device

Use the IoT Edge Azure CLI set-modules command to deploy the modules to the Azure IoT Hub. For example, to deploy the modules defined in the *deployment.debug.template.json* file to IoT Hub *my-iot-hub* for the IoT Edge device *my-device*, use the following command:

Azure CLI

```
az iot edge set-modules --hub-name my-iot-hub --device-id my-device --content
./deployment.debug.template.json --login "HostName=my-iot-hub.azure-
devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=<SharedAccessKey>"
```

Tip

You can find your IoT Hub shared access key in the Azure portal in your IoT Hub > **Security settings** > **Shared access policies** > **iothubowner**.

Debug your module

To debug modules on a remote device, you can use Remote SSH debugging in Visual Studio Code.

To enable Visual Studio Code remote debugging, install the Remote Development extension . For more information about Visual Studio Code remote debugging, see Visual Studio Code Remote Development .

For details on how to use Remote SSH debugging in Visual Studio Code, see Remote Development using SSH

In the Visual Studio Code Debug view, select the debug configuration file for your module. By default, the **.debug** Dockerfile, module's container `createOptions` settings, and the `launch.json` file use *localhost*.

Select **Start Debugging** or select **F5**. Select the process to attach to. In the Visual Studio Code Debug view, you see variables in the left panel.

Debug using Docker Remote SSH

The Docker and Moby engines support SSH connections to containers allowing you to debug in Visual Studio Code connected to a remote device. You need to meet the following prerequisites before you can use this feature.

Configure Docker SSH tunneling

1. Follow the steps in Docker SSH tunneling to configure SSH tunneling on your development computer. SSH tunneling requires public/private key pair authentication and a Docker context defining the remote device endpoint.

2. Connecting to Docker requires root-level privileges. Follow the steps in Manage docker as a non-root user to allow connection to the Docker daemon on the remote device. When you finish debugging, you may want to remove your user from the Docker group.

3. In Visual Studio Code, use the Command Palette (Ctrl+Shift+P) to issue the *Docker Context: Use* command to activate the Docker context pointing to the remote machine. This command causes both Visual Studio Code and Docker CLI to use the remote machine context.

   > **Tip**

All Docker commands use the current context. Remember to change context back to *default* when you are done debugging.

4. To verify the remote Docker context is active, list the running containers on the remote device:

```
Bash

docker ps
```

The output should list the containers running on the remote device similar:

```
Output

PS C:\> docker ps
CONTAINER ID   IMAGE
COMMAND                  CREATED        STATUS         PORTS
NAMES
a317b8058786   myacr.azurecr.io/filtermodule:0.0.1-amd64
"dotnet filtermodule…"   24 hours ago   Up 6 minutes
filtermodule
d4d949f8dfb9   mcr.microsoft.com/azureiotedge-hub:1.4
"/bin/sh -c 'echo \"$…"   24 hours ago   Up 6 minutes   0.0.0.0:443-
>443/tcp, :::443->443/tcp, 0.0.0.0:5671->5671/tcp, :::5671->5671/tcp,
0.0.0.0:8883->8883/tcp, :::8883->8883/tcp, 1883/tcp   edgeHub
1f0da9cfe8e8   mcr.microsoft.com/azureiotedge-simulated-temperature-
sensor:1.0   "/bin/sh -c 'echo \"$…"   24 hours ago   Up 6 minutes
                                       tempSensor
66078969d843   mcr.microsoft.com/azureiotedge-agent:1.4
"/bin/sh -c 'exec /a…"   24 hours ago   Up 6 minutes
                                       edgeAgent
```

5. In the *.vscode* directory, add a new configuration to **launch.json** by opening the file in Visual Studio Code. Select **Add configuration** then choose the matching remote attach template for your module. For example, the following configuration is for .NET Core. Change the value for the *-H* parameter in *PipeArgs* to your device DNS name or IP address.

```
JSON

"configurations": [
{
  "name": "Remote Debug IoT Edge Module (.NET Core)",
  "type": "coreclr",
```

```
    "request": "attach",
    "processId": "${command:pickRemoteProcess}",
    "pipeTransport": {
      "pipeProgram": "docker",
      "pipeArgs": [
        "-H",
        "ssh://user@my-device-vm.eastus.cloudapp.azure.com:22",
        "exec",
        "-i",
        "filtermodule",
        "sh",
        "-c"
      ],
      "debuggerPath": "~/vsdbg/vsdbg",
      "pipeCwd": "${workspaceFolder}",
      "quoteArgs": true
    },
    "sourceFileMap": {
      "/app": "${workspaceFolder}/modules/filtermodule"
    },
    "justMyCode": true
  },
```

Remotely debug your module

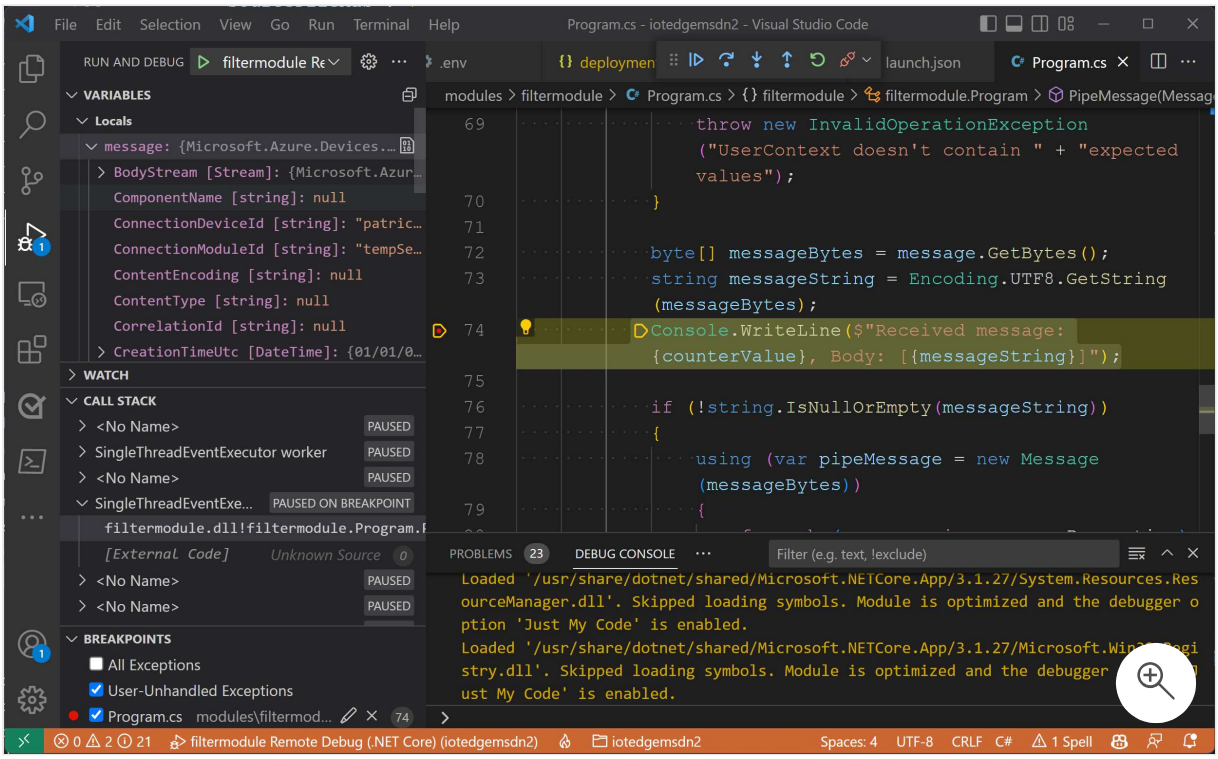
1. In Visual Studio Code Debug view, select the debug configuration *Remote Debug IoT Edge Module (.NET Core)*.

2. Select **Start Debugging** or select **F5**. Select the process to attach to.

3. In the Visual Studio Code Debug view, you see the variables in the left panel.

4. In Visual Studio Code, set breakpoints in your custom module.

5. When a breakpoint is hit, you can inspect variables, step through code, and debug your module.



> **Note**
>
> The preceding example shows how to debug IoT Edge modules on remote containers. The example adds a remote Docker context and changes to the Docker privileges on the remote device. After you finish debugging your modules, set your Docker context to *default* and remove privileges from your user account.

See this IoT Developer blog entry for an example using a Raspberry Pi device.

Next steps

After you've built your module, learn how to deploy Azure IoT Edge modules from Visual Studio Code.

To develop modules for your IoT Edge devices, understand and use Azure IoT Hub SDKs.