Building and Executing Trusted Execution Environment (TEE) based applications on Azure (and on the Edge)

A starter guide for developers

Version 1.2, April 2020

For the latest information about Azure, please see
https://azure.microsoft.com/en-us/overview/

For the latest information on Azure Confidential Computing (ACC), please see
https://azure.microsoft.com/en-us/solutions/confidential-compute/

For the latest information about open source on Azure, please see
https://azure.microsoft.com/en-us/overview/choose-azure-opensource/

For the latest information on the Open Enclave (OE) SDK, please see
https://openenclave.io/sdk/

This page is intentionally left blank.

Table of contents

NOTICE 3
ABOUT THIS GUIDE 4
GUIDE ELEMENTS 9
GUIDE PREREQUISITES 10
Installing OpenSSH on Windows 10 10
Generating your RSA Key pairs with OpenSSH 11
MODULE 1: SETTING UP A CONFIDENTIAL COMPUTING VM IN AZURE 13
OVERVIEW 13
STEP-BY-STEP DIRECTIONS 15
Deploying a v1 DC-series VM on Azure 15
Deploying a v2 DCsv2-series VM on Azure 20
Connecting to your DC-series VM 26
Cloning and building the Open Enclave SDK 28
Using the Open Enclave SDK 29
MODULE 2: DEVELOPING TEE-BASED APPLICATION IN AZURE 34
OVERVIEW 34
IMPORTANT CONCEPTS 35
Terminology 35
Enclave interface definition 36
Data marshalling 37
STEP-BY-STEP DIRECTIONS 38
Building a TEE-based Linux application on Intel SGX 38
MODULE 3: DEVELOPING TEE-BASED APPLICATION FOR THE EDGE 47
OVERVIEW 47
IMPORTANT CONCEPT 49
Azure IoT Platform for the "Intelligent Cloud, Intelligent Cloud" 49
STEP-BY-STEP DIRECTIONS 50
Building a TEE-based Linux application on a simulated ARM TrustZone environment 51
Building a TEE-based Linux module on an Edge ARM TrustZone device 62
APPENDIX. PREREQUISITES AND ADDITIONAL CONFIGURATION 86
SETTING UP A CORE AZURE IOT ENVIRONMENT 86

Creating an Azure Container Registry 86
Creating an Azure IoT Hub 88
Registering an Azure IoT Edge device to your Azure IoT hub 91
Installing and starting the Azure IoT Edge runtime on your device 93

Notice

This guide for developers is intended to illustrate a new way for companies to build and execute so-called Trusted Execution Environment (TEE) based applications using the Microsoft Open Enclave SDK (OESDK) in C and C++. The OESDK is available in open source at https://openenclave.io/sdk/.

MICROSOFT DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, IN RELATION WITH THE INFORMATION CONTAINED IN THIS WHITE PAPER. The white paper is provided "AS IS" without warranty of any kind and is not to be construed as a commitment on the part of Microsoft.

Microsoft cannot guarantee the veracity of the information presented. The information in this guide, including but not limited to internet website and URL references, is subject to change at any time without notice. Furthermore, the opinions expressed in this guide represent the current vision of Microsoft France on the issues cited at the date of publication of this guide and are subject to change at any time without notice.

All intellectual and industrial property rights (copyrights, patents, trademarks, logos), including exploitation rights, rights of reproduction, and extraction on any medium, of all or part of the data and all of the elements appearing in this paper, as well as the rights of representation, rights of modification, adaptation, or translation, are reserved exclusively to Microsoft France. This includes, in particular, downloadable documents, graphics, iconographics, photographic, digital, or audiovisual representations, subject to the pre-existing rights of third parties authorizing the digital reproduction and/or integration in this paper, by Microsoft France, of their works of any kind.

The partial or complete reproduction of the aforementioned elements and in general the reproduction of all or part of the work on any electronic medium is formally prohibited without the prior written consent of Microsoft France.

Publication: August 2019 (updated April 2020)
Version 1.2

© 2019 Microsoft France. All rights reserved

About this guide

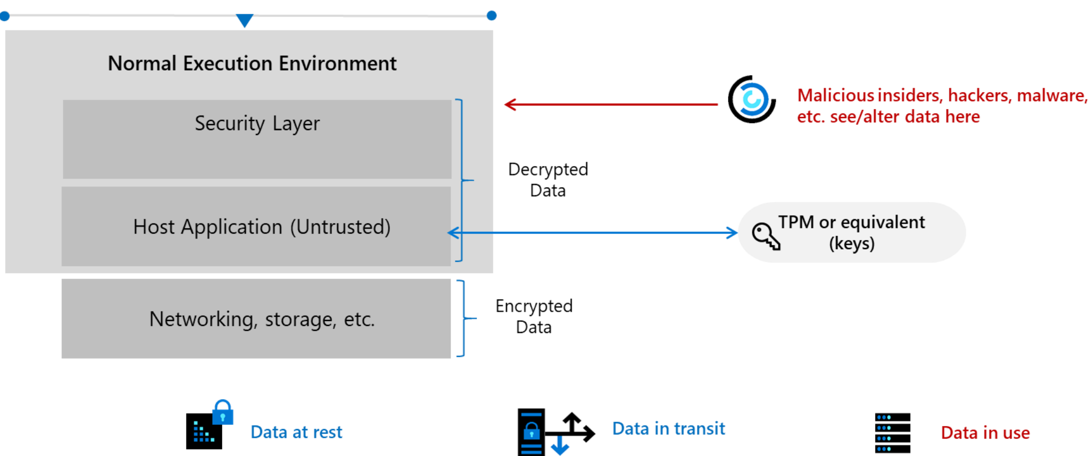
Welcome to the **Building and Executing Trusted Execution Environment (TEE) based applications on Azure** starter guide for developers.

This document is part of a series of guides that covers confidential computing in the Cloud, and the Edge, and considerations that pertain to it from a development perspective and/or an infrastructure one. (This series of guides is available at https://aka.ms/CCDevGuides.)

Data can be uploaded encrypted (i.e. **in transit**) to the cloud. Furthermore, in most situations, not to say in all of them, data is stored encrypted (i.e. **at rest**) in the cloud and decrypted on the fly when used or computed by a program. This is both a usual and an adapted way to proceed for the most common data.

But sometimes, protecting data at rest and data in transit is not enough. Data may be indeed too sensible to appear in clear in memory (i.e. **in use**), even if the (virtual) machine and the workload processing data can be considered hardened respectively secured.

Financial data processing constitutes one typical illustration but is far from being the only one.



The specificities of your workload may require protecting data in use confidentiality and integrity from malicious insiders with administrative privilege or direct access, safeguarding against hackers and malware that exploit bugs in the operating system, application, or hypervisor, protecting against third-party access without consent, etc.

By extension, you may also consider the situation where multiple data sources from different organizations that do not necessarily trust each other, or are even competitor, must be combined.

For example, multiples organizations, such as health facilities/institutions and pharmaceutic industries, may have to joint their effort and combine their own respective private patient/health data sources to build, train, evaluate a deep/machine learning model for a better algorithmic outcome without sacrificing data confidentiality: organizations do not see each other's data sets.

The resulting solution, known as a privacy-preserving multi-party machine learning[1], should allow to fusion sensitive data sources across different organizations while not revealing data to participants or the cloud platform.

[1] OBLIVIOUS MULTI-PARTY MACHINE LEARNING ON TRUSTED PROCESSORS: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/paper.pdf

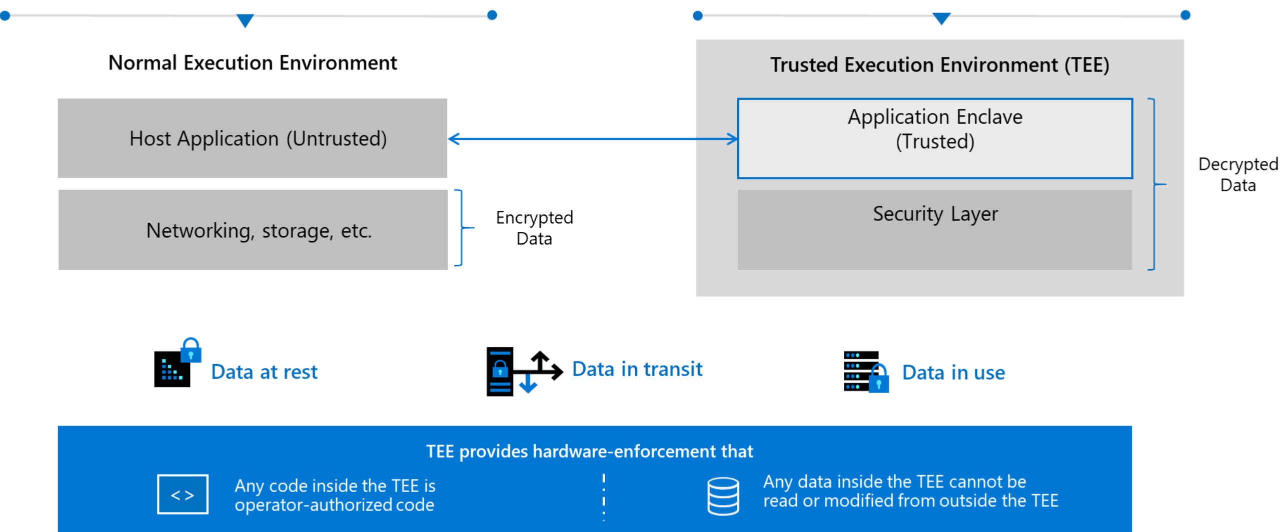
> **Note** For more information on privacy-preserving multi-party machine learning, see the various presentations of the one-day workshop NIPS 2016 Workshop Private Multi-Party Machine Learning[2].

In some cases, your sensitive content is the code and not the data. To secure sensitive IP, you may require protect confidentiality and integrity of your code while it's in use.

Increasing popularity of use cases like the above ones has led to secure compute workloads within the confines of Trusted Execution Environments[3] (TEEs).

This concept called Confidential Computing is an ongoing **effort to protect data and/or code throughout its lifecycle at rest, in transit and now in use**.

With the use of TEEs or simply enclaves, you can build applications that protect workloads during computation.



A TEE-based application partitions itself into two components 1) an untrusted component (called the host application) and 2) a trusted component, i.e. a TEE or enclave:

1. The host component runs unmodified on the untrusted operating system (OS), while the trusted component runs within the enclave. It's a normal user mode application that loads an enclave into its address space before starting to interact with an enclave.
2. The enclave is a secured container provided by a TEE implementation whose memory (text and data) is protected from entities outside the enclave, including the host application, privileged users, and even the hardware: a user (remotely) connected to the machine (even a trusted administrator or the operating system (OS)) can't see what is running and processing inside this enclave.

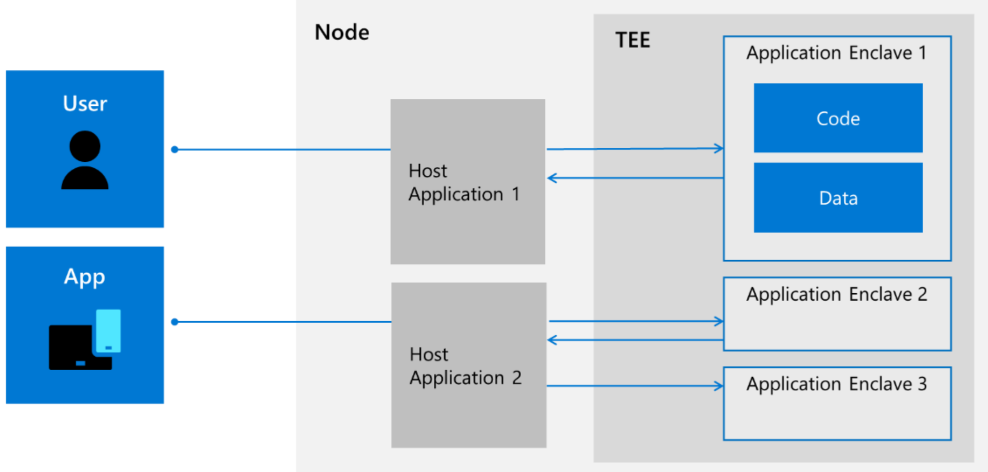
   These protections allow enclaves to perform secure computations with assurances that secrets will not be compromised. Thus, all functionality that needs to be run in an enclave should be compiled into the enclave binary. The enclave may run in an untrusted environment with the expectation that secrets will not be compromised.

[2] NIPS 2016 Workshop Private Multi-Party Machine Learning: https://pmpml.github.io/PMPML16/

[3] Trusted execution environment: https://en.wikipedia.org/wiki/Trusted_execution_environment

One can obtain a remote attestation of the enclave's identity, call the enclave's exposed functions' interface, but cannot access the code itself within the enclave, the defined variables (and therefore data). In this context, data stays encrypted all the way long from the user point of view: in transit, at rest, and in use. Same considerations may apply for the code itself.

To sump-up, code and date are isolated in encrypted enclaves, preventing snooping or tampering even by the OS or trusted administrators.



TEE-based applications root trust in any secure silicon TEE built on such enclaving technologies like Intel Software Extension Guard[4] (SGX) - The Intel SGX instruction extension was introduced with 7th Generation Intel Core processor platforms and Intel Xeon processor E3 v5 for data center servers back in 2015 -, ARM TrustZone[5] (TZ), and embedded Secure Elements using Windows or Linux OSs.

Note For more information on the two above TEE technologies, see article SGX AND TRUSTZONE[6].

But all of this is still really a low-level work and developing applications above that is really difficult and requires both advanced security expertise and specifics skills.

In this context, Microsoft Research, together with partners, has embarked and invested in a way that simplifies TEE-based application development for all audiences from hardcore hardware security experts to edge and cloud software applications developers, regardless of the underlying enclaving technologies. The effort results in the Microsoft Open Enclave SDK[7] (OESDK), an open source framework available on GitHub over a year ago under an open source license.

The OESDK aims at creating a single unified enclaving abstraction for developer to build applications once that run across multiple TEE architectures, and thus was designed to:

- Make it easy to write and debug code that runs inside TEEs.
- Allow the development of code that's portable between TEEs.

---

[4] Intel Software Extension Guard: https://software.intel.com/en-us/sgx

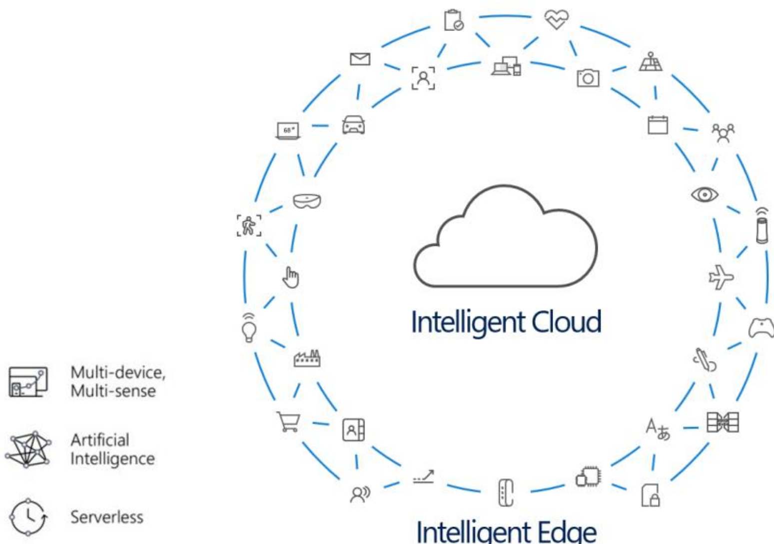
[5] Layered Security for Your Next SoC: https://www.arm.com/products/silicon-ip-security

[6] SGX AND TRUSTZONE: https://github.com/openenclave/openenclave/blob/feature.new_platforms/new_platforms/docs/sgx_trustzone_arch.md

[7] Open Enclave SDK: https://Open Enclave.io/sdk/

- Provide a flexible plugin model to support different runtimes and cryptographic libraries.
- Have a high degree of compatibility with existing code.

As such, the OESDK provides an open source consistent API surface across enclave technologies and platforms from the cloud to the edge[8].



Note Beyond the ongoing shift to the "Intelligent Cloud", one can observe the inevitable rise of the "Intelligent Edge". The enormity of data and the evolution of intelligence in (Industrial) Internet of Things ((I)IoT) is inspiring new computational models that distribute compute from cloud to edge and synthesize data to improve business outcomes. As the (I)IoT continues to increasingly accelerate and businesses realize the immense benefits, the next breakthrough capability upon us - or even already here.

Enabling intelligence on edge devices means enabling analytics and insights to happen closer to the source of the (sensitive) data, saving organizations money and simplifying their solutions. The result is rapid growth in patterns such as deploying proprietary algorithms and data including Artificial Intelligence (AI), generating valuable insights including monetization avenues in, and creating direct actions including those controlling downstream critical infrastructure from (I)IoT devices.

Increased functionalities and computing power available on the edge are already changing the way organizations design and build products, from intelligent construction site video surveillance[9], to oil rig maintenance tracking[10]. But such patterns are potentially susceptible to exposure of algorithms, data, valuable insights, and tampering of actions unless the computational environment is protected by Confidential Computing.

Confidential compute resources are different for the "Intelligent Cloud" to the "Intelligent Edge" but the development experience for confidential compute workloads targeting the cloud and edge platforms need not be different. In a single development environment, developers can create trusted applications for deployment in

---

[8] The future of computing: intelligent cloud and intelligent edge: https://azure.microsoft.com/en-us/overview/future-of-cloud

[9] INTELLIGENT VIDEO SURVEILLANCE: AN UNTAPPED SOURCE OF VALUE: https://blog.deepomatic.com/en/intelligent-video-surveillance-an-untapped-source-of-value/4098/

[10] EDGE TECHNOLOGIES GIVE OIL, GAS OPERATORS EXTRA COMPUTING POWER: https://www.rigzone.com/news/edge_technologies_give_oil_gas_operators_extra_computing_power-30-aug-2018-156789-article/

cloud enclaves like Azure Confidential Computing[11] (ACC) and in TEE enabled Internet of Things (IoT) Edge devices such as the ones running Azure IoT Edge[12] bits.

> **Note** For more information on Azure Confidential Computing, see blog post Introducing Azure confidential computing[13] and the webcast Azure Confidential Computing updates with Mark Russinovich | Best of Microsoft Ignite 2018[14].

> **Note** For more information on confidential computing with Azure IoT Edge, see blog post Simplifying confidential computing: Azure IoT Edge security with enclaves – Public preview[15] and the webcast Deep Dive: Confidential Computing in IoT using Open Enclave SDK[16].

In other words, this means that you as a developer can use the same APIs across multiple enclaves, greatly reducing the complexity of following best practices and encouraging organizations to integrate (host) applications with enclaves.

As TEE technology matures and as different implementations arise, the OESDK is committed to supporting an API set that allows developers to build once and deploy on multiple technology platforms, different environments from cloud to hybrid to edge, and for both Linux and Windows.

> **Important note** As of this writing, and available with this version is the ability to write enclave applications for cloud workloads targeting TEE technology based on Intel SGX hardware technology with a Linux host application
>
> Preview support is also provided for new TEE platforms, namely ARM TrustZone with a Linux host application, and intel SGX with a Windows host application via the Intel SGX SDK. Support for a Windows host application on ARM TrustZone and native Open Enclave support for a Windows host application on Intel SGX will be added in the future.

This broad applicability across different enclave technologies greatly simplifies the work developers must do to protect sensitive data. Furthermore, with accessibility by all security expertise as topmost goal, this integration is laden with features to truly simplify and shorten the journey from idea to at-scale production deployment of secure (intelligent edge) TEE-based applications.

> **Note** Microsoft has recently joined partners and the Linux Foundation to create Confidential Computing Consortium[17] that will be dedicated to defining and accelerating the adoption of confidential computing.
>
>
>
>
> Microsoft will be contributing the OESDK to the Confidential Computing Consortium to develop a broader industry collaboration and ensure a truly open development approach. “The Open Enclave SDK is already a popular tool for developers working on Trusted Execution Environments, one of the most promising areas for protecting data in use,” said Mark Russinovich, chief technical officer, Microsoft. “We hope this contribution to the Consortium can put the tools in

---

[11] Azure Confidential Computing: https://azure.microsoft.com/solutions/confidential-compute/

[12] Azure IoT Edge: https://azure.microsoft.com/en-us/services/iot-edge/

[13] Introducing Azure confidential computing: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/

[14] Azure Confidential Computing updates with Mark Russinovich | Best of Microsoft Ignite 2018: https://www.youtube.com/watch?v=Qu6sP0XDMU8

[15] Simplifying confidential computing: Azure IoT Edge security with enclaves – Public preview: https://azure.microsoft.com/en-us/blog/simplifying-confidential-computing-azure-iot-edge-security-with-enclaves-public-preview/

[16] Deep Dive: Confidential Computing in IoT using Open Enclave SDK: https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Confidential-Computing-in-IoT-using-Open-Enclave-SDK

[17] Confidential Computing Consortium: https://confidentialcomputing.io/

even more developers' hands and accelerate the development and adoption of applications that will improve trust and security across cloud and edge computing."[18]

In this starter guide, and as its title suggest, we will cover the basics of TEE-based application development.

You will learn how to create and deploy this new kind of applications on top of Azure Confidential Computing (ACC), using the OESDK.

For that purposes, you're invited to follow a short series of modules, each of them illustrating a specific aspect of the TEE-based application development.

Each module within the guide builds on the previous. You're free to stop at any module you want, but our advice is to go through all the modules.

At the end of the starter guide, you will be able to:

- Understand the Azure Confidential Computing (ACC) offering,
- Instantiate a Confidential Computing (CC) or other Linux VMs well-suited for trusted applications development,
- Setup a full-pledged development environment with Visual Studio and Visual Studio Code,
- Create new trusted applications or containers using the Open Enclave SDK (OESDK) in C or C++ in the Cloud and in the Edge.

Guide elements

In the starter guide modules, you will see the following elements:

- **Step-by-step directions**. Click-through instructions - along with relevant snapshots - or links to online documentation for completing each procedure or part.
- **Important concepts**. An explanation of some of the concepts important to the procedures in the module, and what happens behind the scenes.
- **Sample applications, and files**. A downloadable or cloneable version of the project containing the code that you will use in this guide, and other files you will need. **Please go to https://github.com/openenclave/ on GitHub to download or clone all necessary assets.**

[18] NEW CROSS-INDUSTRY EFFORT TO ADVANCE COMPUTATIONAL TRUST AND SECURITY FOR NEXT-GENERATION CLOUD AND EDGE COMPUTING: https://www.linuxfoundation.org/press-release/2019/08/new-cross-industry-effort-to-advance-computational-trust-and-security-for-next-generation-cloud-and-edge-computing/

Guide prerequisites

To successfully leverage the provided code in this starter guide, you will need:

- A Microsoft account[19].
- An Azure subscription. If you don't have an Azure subscription, create a free account[20] before you begin.
- A windows 10 local machine.
- A code editor of your choice, such as Visual Studio[21] or Visual Studio Code[22], with C++ for Linux and Open Enclave installed. The related installation and configuration will be further covered later in this guide.
- A terminal console for your Windows 10 local machine, which allows you to remotely connect to a virtual machine (VM) in SSH, such as PuTTY[23], Git for Windows[24] (2.10 or later).

Important note With Git, ensure that long paths are enabled: `git config --global core.longpaths true`.

As far as the latter is concerned, recent versions of Windows 10 provide OpenSSH client commands to create and manage SSH keys and make SSH connections from a command prompt.

Note For more information, see blogpost WHAT'S NEW FOR THE COMMAND LINE IN WINDOWS 10 VERSION 1803[25].

Installing OpenSSH on Windows 10

The OpenSSH Client and OpenSSH Server are separately installable components in Windows 10 1809 and above.

Note For information about the OpenSSH availability on Windows 10, see here.

To install OpenSSH on your Windows 10 local machine, perform the following steps.

1. Open an elevated PowerShell console.
2. Run the following command:

```
PS C:\> Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```

[19] Microsoft Account: https://account.microsoft.com/account?lang=en-us

[20] Create your Azure free account today: https://azure.microsoft.com/en-us/free/?WT.mc_id=A261C142F

[21] Visual Studio: https://visualstudio.microsoft.com/

[22] Visual Studio Code: https://code.visualstudio.com/

[23] PuTTY: https://www.chiark.greenend.org.uk/~sgtatham/putty/

[24] Git for Windows: https://git-for-windows.github.io/

[25] WHAT'S NEW FOR THE COMMAND LINE IN WINDOWS 10 VERSION 1803: https://blogs.msdn.microsoft.com/commandline/2018/03/07/windows10v1803/

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0


Path          :
Online        : True
RestartNeeded : False



PS C:\WINDOWS\system32>
```

Once the installation completes, you can use the OpenSSH client from PowerShell or the Windows 10 command shell.

Generating your RSA Key pairs with OpenSSH

OpenSSH includes different tools and more specifically the `ssh-keygen` command for generating secure RSA key pairs, that can be in turn used for key authentication with SSH.

RSA Key pairs refer to the public and private key files that are used by certain authentication protocols.

To generate your RSA Key pairs, perform the following steps.

1. Open an elevated PowerShell console.
2. Run the following command:

```
PS C:\> ssh-keygen
```

You can just hit ENTER to generate them, but you can also specify your own filename if you want. At this point, you'll be prompted to use a passphrase to encrypt your private key files. The passphrase works with the key file to provide 2-factor authentication. For this example, we are leaving the passphrase empty.

```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\philber/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:\Users\philber/.ssh/id_rsa.
Your public key has been saved in C:\Users\philber/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:zHdApZpi0+mzKynqCVEzJxWgdmSkiaTfqQqi9SLIuCQ europe\philber@PHILBER001
The key's randomart image is:
+---[RSA 3072]----+
| .o=o.    ...    |
|+.=.     . .     |
|++=..     o      |
|.o.* . + + .     |
|. . o + S . .    |
| . . . + . .     |
|E.o    .o        |
|X=.o. o  o       |
|+++o.. .o.       |
+----[SHA256]-----+
PS C:\WINDOWS\system32>
```

Note SSH public-key authentication uses asymmetric cryptographic algorithms to generate two key files – one "private" and the other "public". The **private key** file is the equivalent of a password and should protected under all circumstances. If someone acquires your private key, they can log in as you to any SSH server you have access to. The **public key** is what is placed on the SSH server and may be shared without compromising the private key.

When using key authentication with an SSH server, the SSH server and client compare the public key for username provided against the private key. If the public key cannot be validated against the client-side private key, authentication fails.

By default, the files are saved in the following folder *%USERPROFILE%\.ssh:*

File	Description
%USERPROFILE%\.ssh\id_rsa	Contains the RSA private key
%USERPROFILE%\.ssh\id_rsa.pub	Contains the RSA public key.

Module 1: Setting up a Confidential Computing VM in Azure

Overview

This first module of this starter guide will illustrate how to deploy a Confidential Compute (CC) VM to later leverage the Open Enclave SDK (OESDK) to develop in C and C++ Trusted Execution Environment (TEE) based applications.

In the Azure Platform, the OESDK must be indeed installed on top of a Confidential Compute DC-series[26] or DCsv2-series[27] [28] virtual machine (VM).

For the Azure Confidential Computing (ACC) offering currently in public preview, v1 DC-series and v2 DCsv2-series VMs are indeed (as of this writing) the (only) two types of VMs in Azure that can support Trusted Execution Environment (TEEs), thanks to the latest generation of the Intel XEON processor with the Intel Software Guard Extensions (Intel SGX) technology.

As such, both DC-series and DCsv2-series VMs are generation 2 VMs[29] that, besides the supports the Intel SGX technology, use the new UEFI-based boot architecture rather than the BIOS-based architecture used by generation 1 VMs, along with additional features that are not available in generation 1 VMs, such as increased memory, and virtualized persistent memory (vPMEM).

These series' instances enable customers to build secure enclave-based applications to protect their code and the confidentiality and integrity of their data while it's processed, and thus in use in the public cloud. Example use cases include confidential multiparty data sharing, fraud detection, anti-money laundering, blockchain, confidential usage analytics, intelligence analysis and confidential machine learning.

The DC-series corresponds to the initially introduced family of CC VMs in Azure, that are backed by the 3.7GHz Intel XEON E-2176G[30] processor with SGX technology, and with the Intel Turbo Boost Technology can go up to 4.7GHz. This family is currently available in East US and West Europe regions only.

---

[26] PREVIOUS GENERATIONS OF VIRTUAL MACHINE SIZES: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-previous-gen?toc=/azure/virtual-machines/linux/toc.json&bc=/azure/virtual-machines/linux/breadcrumb/toc.json#preview-dc-series

[27] GENERAL PURPOSE VIRTUAL MACHINE SIZES: https://docs.microsoft.com/en-us/azure/virtual-machines/dcv2-series

[28] AZURE LAUNCHES DC-SERIES CONFIDENTIAL COMPUTE VM PREVIEW: https://www.petri.com/azure-launches-dc-series-confidential-compute-vm-preview

[29] SUPPORT FOR GENERATION 2 VMS ON AZURE: https://docs.microsoft.com/en-us/azure/virtual-machines/linux/generation-2#creating-a-generation-2-vm

[30] Intel XEON E-2176G: https://www.intel.com/content/www/us/en/products/processors/xeon/e-processors/e-2176g.html

Two VM sizing options are available for the v1 DC-series:

1. Standard_DC2s with 2 vCPUs and 8 GB of memory,
2. Standard_DC4s with 4 vCPUs and 16 GB of memory.

These sizes are still supported but will not receive additional capacity.

V1 DC-Series VMs in Preview and deploys an older version of the DC-Series VMs. They aren't going to be generally available and will remain in preview until deprecation.

For the most up-to-date technology and confidential computing VM, you will need to use DCsv2-series instead that correspond to an Azure Confidential Compute (Virtual Machine) V2 deployment.

The DCsv2-series is indeed a new family of CC VMs in Azure that are backed by the latest generation of the Intel XEON E-2288G processor[31] with the Intel SGX technology. With the above-mentioned Intel Turbo Boost Technology, these machines can go up to 5.0GHz. This family is currently available in UK South and Canada Central only.

DCsv2-series VMs allow for a greater selection of VM sizes, higher EPC (Enclave Page Cache), and a higher level of support.

As such, the following four sizing options VM sizing options are available for the DC-Series:

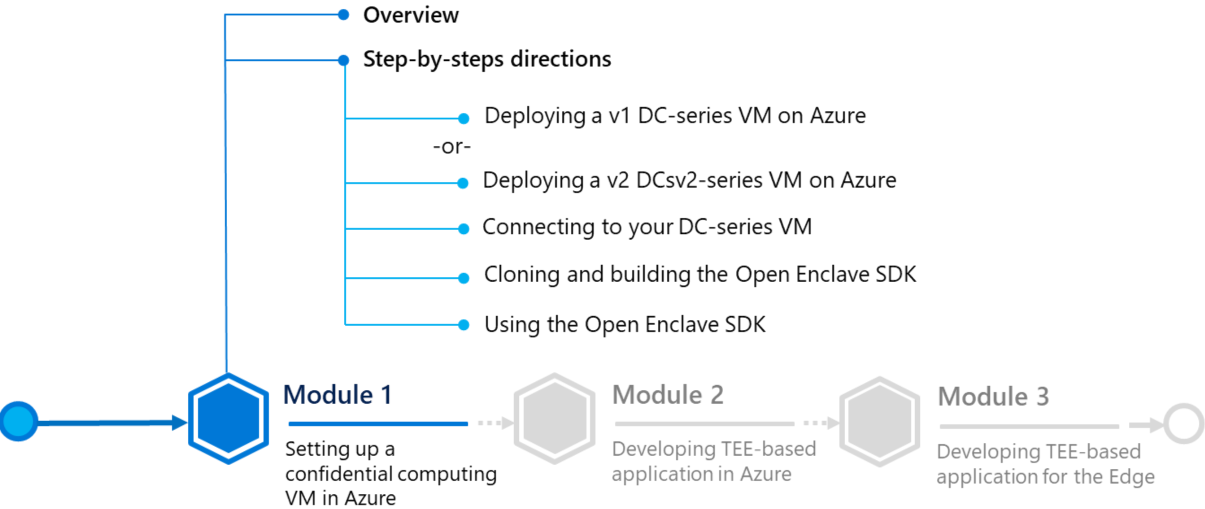
1. Standard_DC1s_v2 with 1 vCPU and 4 GB of memory,
2. Standard_DC2s_v2 with 2 vCPUs and 8 GB of memory,
3. Standard_DC4s_v2 with 4 vCPUs and 16 GB of memory,
4. Standard_DC8s_v2 with 8 vCPUs and 32 GB of memory.

Currently, three operating systems are supported for the above families of VMs:

1. Windows Server 2016 Datacenter,
2. Ubuntu Server 16.04 LTS,
3. Ubuntu Server 18.04 TLS.

> **Note** Additional OS offerings may be supported once the ACC program transitions from public preview to general availability (GA).

[31] Intel XEON E-2288G: https://www.intel.com/content/www/us/en/products/processors/xeon/e-processors/e-2288g.html



Step-by-step directions

This module covers the following four activities:

1. Deploying a v1 DC-series VM on Azure.

-or-

2. Deploying a v2 DCsv2-series VM on Azure.
3. Connecting to your DC-series VM.
4. Cloning and building the Open Enclave SDK.
5. Using the Open Enclave SDK.

Each activity is described in order in the next sections.

Deploying a v1 DC-series VM on Azure

The older v1 DC-series VMs are not listed by default in your **Virtual Machines** tab in the Azure portal. They can instead can be found in the directory in the Azure Marketplace[32] or by searching "*Confidential Compute*" in the search bar in Azure.

To (still) deploy a v1 DC-series VM in your Azure subscription, perform the following steps:

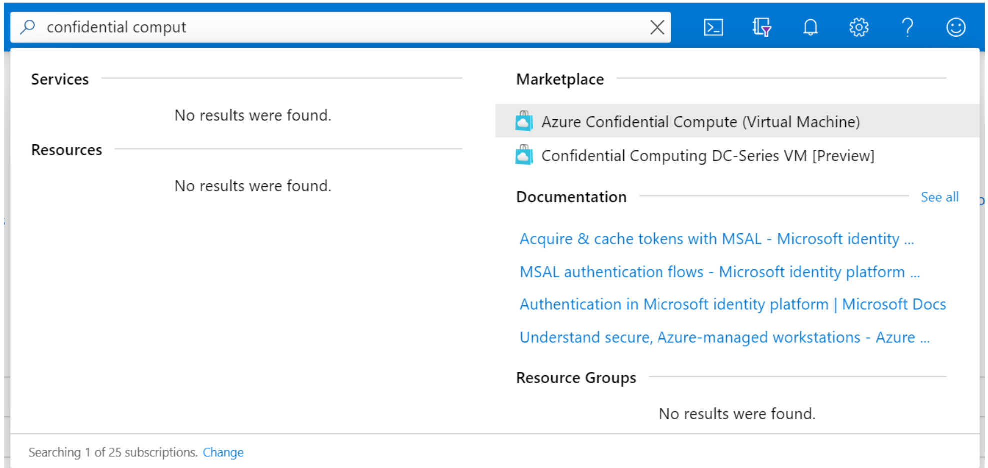
Note	For more information, see article GET STARTED WITH MICROSOFT AZURE* CONFIDENTIAL COMPUTING[33].

1. Open a browser session and go to the Azure portal at https://portal.azure.com.
2. Sign in with your Azure account.

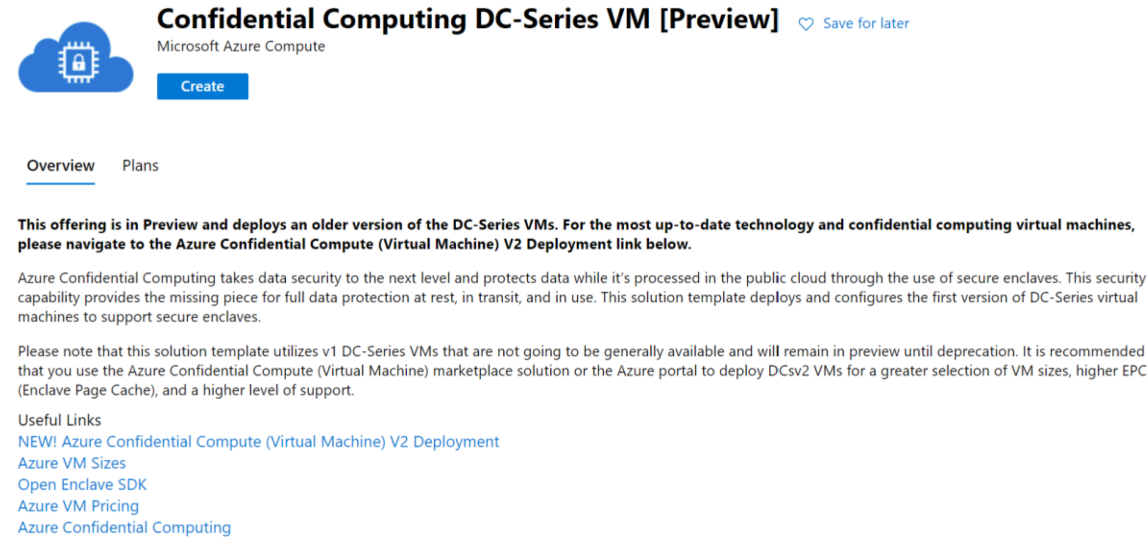
[32] Confidential Compute VM Deployment: https://azuremarketplace.microsoft.com/marketplace/apps/microsoft-azure-compute.confidentialcompute

[33] GET STARTED WITH MICROSOFT AZURE* CONFIDENTIAL COMPUTING: https://software.intel.com/en-us/articles/get-started-with-azure-confidential-computing

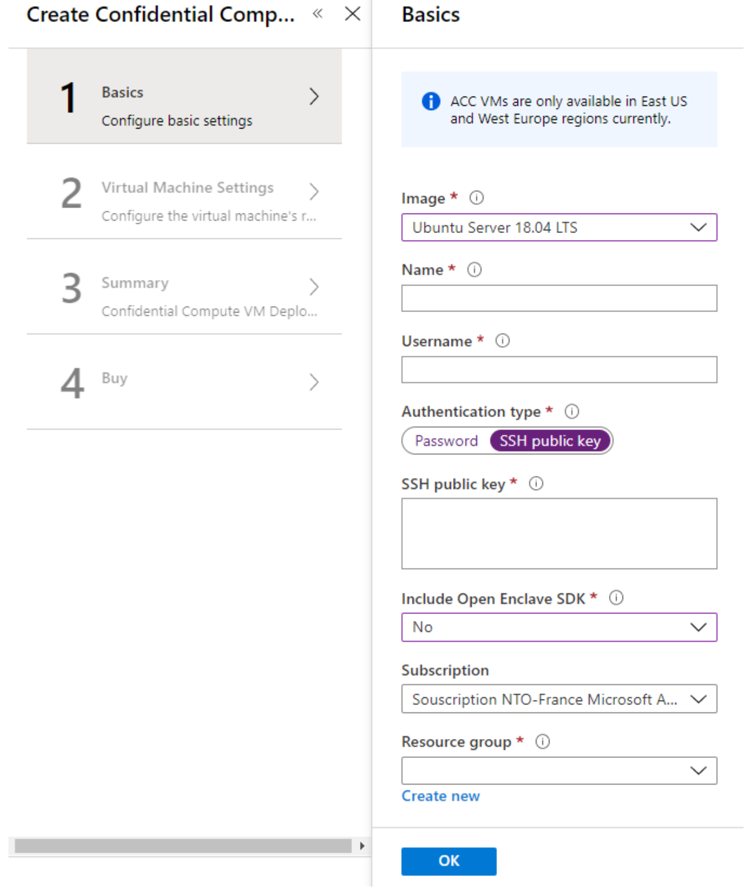
3. First search for "*Confidential Comput*" in the search bar in the Azure portal.



4. Click on **Confidential Computing DC-series VM [Preview]** under **Marketplace**.to select a DC-Series VM. (**Azure Confidential Compute (Virtual Machine)** corresponds to the template for creating a DCsv2-series VM). You will be then re-directed to the Confidential Compute VM Deployment wizard.



5. Click **Create**.



Create Confidential Comp...
1 Basics
Configure basic settings
2 Virtual Machine Settings
Configure the virtual machine's r...
3 Summary
Confidential Compute VM Deplo...
4 Buy
Basics
ACC VMs are only available in East US and West Europe regions currently.
Image *
Ubuntu Server 18.04 LTS
Name *
Username *
Authentication type *
Password
SSH public key
SSH public key *
Include Open Enclave SDK *
No
Subscription
Souscription NTO-France Microsoft A...
Resource group *
Create new
OK

6. Specify the required settings.

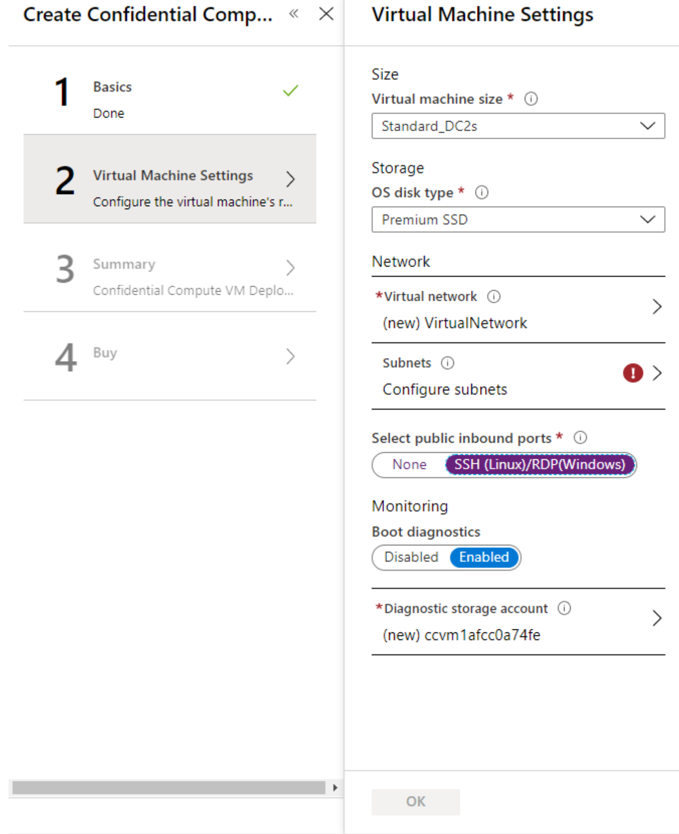
Setting	Description
Image	Select **Ubuntu Server 18.04 LTS**.
Name	Provide the VM a hostname (as a resource, which will be displayed in Azure). For example, **acc-vm1** in our illustration.
Username	Specify a username for the privileged user account of the VM. For example, **azureadmin** in our illustration.
Authentication type	Select **SSH public key** for stronger authentication to later remotely connect to your VM.
SSH public key	Specify a RSA public key[34] in the single-line format beginning with "*ssh-rsa*" - you can use instead the multi-line PEM format -. On a Windows command prompt, run the following command to retrieve the content of your SSH public key and copy it to the clipboard. `C:\> type %USERPROFILE%\.ssh\id_rsa.pub | clip` And then, paste the content in this field.
Include Open Enclave SDK	Ensure that the Open Enclave SDK (OESDK) will NOT be included with this VM deployment. **No** should be selected. **Important note** The version of the Open Enclave SDK that comes with the VM is currently outdated. You will have to install it manually later on. For more information on how to install the Open Enclave SDK, see article INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04)[35].
Subscription	Select your own subscription. If you have more than one, select the most appropriate subscription.
Resource group	For public preview, the wizard will only allow deployment to an empty resource group. Create one during VM deployment as follows: 1. Under **Resource group**, click on **Create new**. 2. In the dialog box, name the new resource group and click on **OK**.

[34] PUBLIC KEY AUTHENTICATION FOR SSH: https://www.ssh.com/ssh/public-key-authentication

[35] INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04): https://github.com/openenclave/openenclave/blob/v0.6.x/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md

Location	Select the Microsoft Azure data enter location to which you want to deploy. Choose between **East US** and **West Europe** as this particular type of VM is only available in these locations. Any selection other than these two locations will fail validation checks.

Click on **OK** to continue to the **Virtual Machine Settings** page.



7. This second page of the form is about more specific settings for the DC-series VM, which include VM size, storage type, and virtual network details. Fill in the **Virtual Machine Settings**.

Setting	Description
Size	Azure Confidential Computing (ACC) public preview DC_series VMs come in two sizes. Let **Standard_DC2s** selected by default.
Storage	Select your preferred storage type. **Premium SSD** is the default.
Virtual Network	Configure the (new) virtual network (VNet) where your VM will reside. For simplicity, this starter guide will use the default settings.
Subnets	Configure the subnet. Again, this starter guide will use the defaults provided. Go to the configuration sub-menu (click on the red exclamation point) and click on **OK** to accept the default values Click on the option, then click on **OK** below the newly created subnet.
Inbound ports	Select **SSH (Linux)/RDP (Windows)** as you're going to use SSH.

	Important note The ports will be open for all public inbound traffic from the Internet, posing a serious security issue. For a production environment, it is recommended that you leave selected **none**; after the VM has been created and deployed, configure the VM's networking inbound port rules to open the required port for a specific IP Address range or enable the Microsoft Azure Security Center Just-in-time VM access.
Boot diagnostics	Leave **Enabled**.
Diagnostic storage account	Leave the default parameter untouched.

8. Click on **OK** to continue. Validation of your configuration settings will occur on the **Summary** page.
9. If validation has passed, review your configuration and click on **OK** to continue.
10. Before creating and deploying your newly configured VM, carefully read the terms of use and understand any costs associated with the use of Microsoft Azure resources. When you are ready to deploy the VM, click on **Create**. Your DC-series VM will be deployed on Azure.

    The completion process will take approximately 10 minutes, at which time you will see a new message in the Microsoft Azure portal notifications tab.

Deploying a v2 DCsv2-series VM on Azure

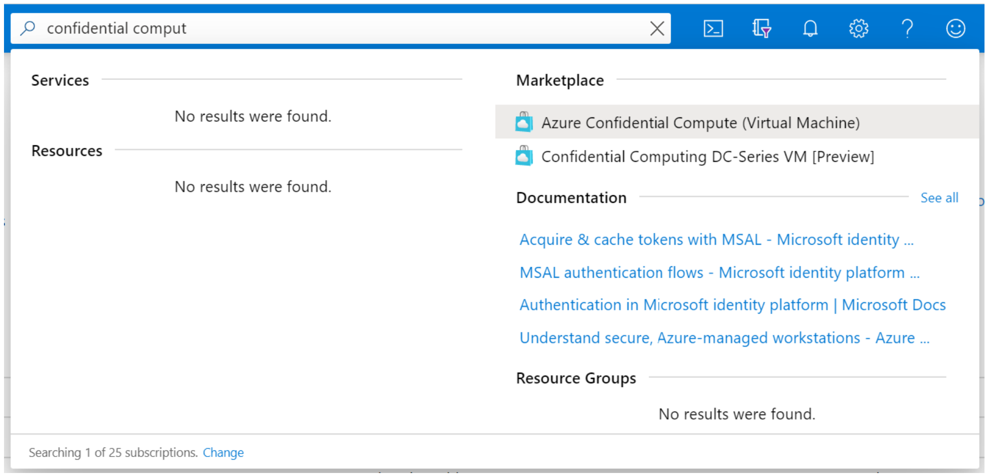
Let's now see how to deploy the newest family of VMs that enable confidential computing features in Azure. It just takes you with a few configurations and a single-click deployment with the provided template to create a DCsV2-series VM.

Like the v1 DC-series, DCsv2-series are not (yet) listed by default in your **Virtual Machines** tab in the Azure portal. They can instead can be found in the directory in the Azure Marketplace[36] or by searching "*Confidential Comput* in the search bar in Azure.

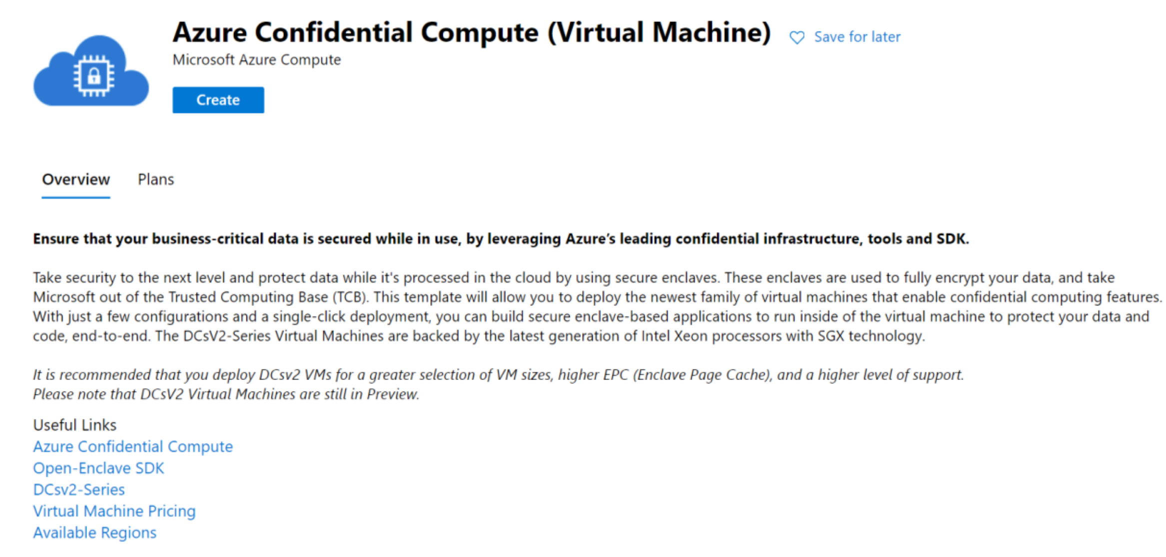
To deploy a DCsv2-series VM in your Azure subscription, perform the following steps:

1. Open a browser session and go to the Azure portal at https://portal.azure.com.
2. Sign in with your Azure account.
3. First search for "*Confidential Comput*" in the search bar in the Azure portal.

[36] Confidential Compute VM Deployment: https://azuremarketplace.microsoft.com/marketplace/apps/microsoft-azure-compute.confidentialcompute



4. Click this time on **Azure Confidential Compute (Virtual Machine)** under **Marketplace** to select a DCsv2-Series VM. (**Confidential Computing DC-series VM [Preview]** corresponds to the template for creating a v1 DC-series VM). You will be then re-directed to the Confidential Compute VM Deployment wizard.



5. Click **Create**.

Create Azure Confidential Compute (Virtual Machine)

Basics Virtual Machine Settings Review + create

Configure and deploy a DCsv2-Virtual Machine in available regions on a compatible image. This series of virtual machines are backed by the latest generation of Intel Xeon processor with SGX to enable confidential computing secure enclaves to protect your data while it's in use.

Note: This solution template only deploys a virtual machine, the operating system, and any associated resources. The virtual machine will install no additional software. Pricing is based on the operating system, size of the virtual machine, and region selected.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Souscription NTO-France Microsoft Azure

Resource group *

Create new

Instance details

Region * (Europe) UK South

Please provide the configuration values for your application. Please note that this configuration requires Specialty SKU DCv2-Series VMs that are currently only available in select regions.

See available DCsv2-Series regions

Image * Ubuntu Server 18.04 (Gen 2)

Virtual Machine name * accvm

Username *

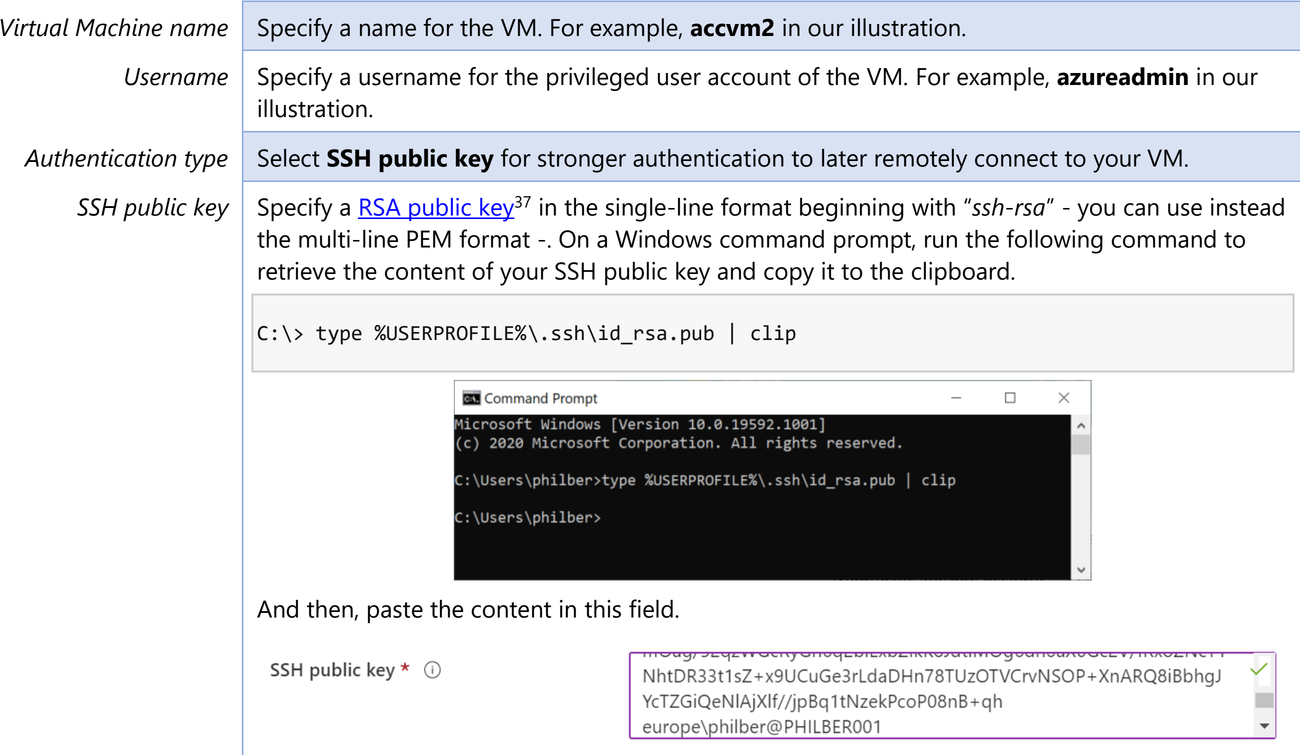
Authentication type * Password SSH Public Key

SSH public key *

Review + create < Previous Next : Virtual Machine Settings >

6. Specify the required settings.

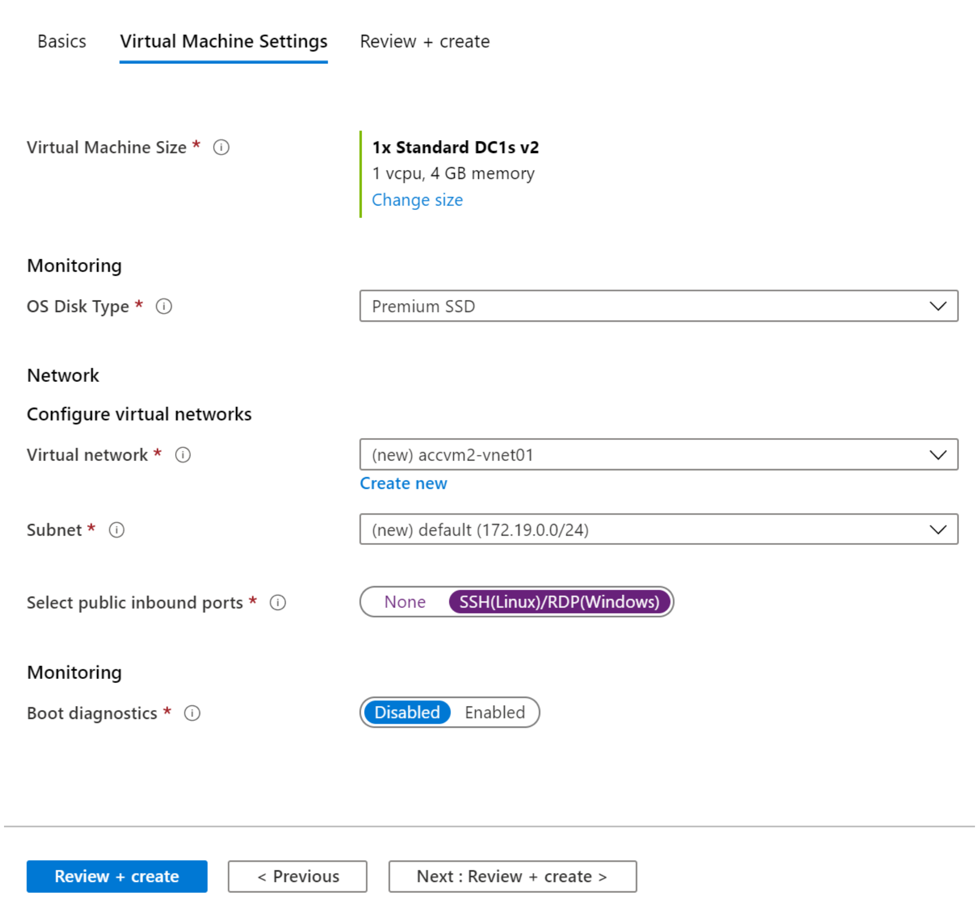
Setting	Description
Subscription	Select your own subscription. If you have more than one, select the most appropriate subscription.
Resource group	For public preview, the wizard will only allow deployment to an empty resource group. Create one during VM deployment as follows: 3. Under **Resource group**, click on **Create new**. 4. In the dialog box, name the new resource group and click on **OK**.
Region	Select the Microsoft Azure data enter location to which you want to deploy. Choose between **UK South** and **Canada Central** as this particular type of VM is only currently available in these locations. Any selection other than these two locations will fail validation checks.
Image	Select **Ubuntu Server 18.04 (Gen 2)**.



Virtual Machine name	Specify a name for the VM. For example, **accvm2** in our illustration.
Username	Specify a username for the privileged user account of the VM. For example, **azureadmin** in our illustration.
Authentication type	Select **SSH public key** for stronger authentication to later remotely connect to your VM.
SSH public key	Specify a RSA public key[37] in the single-line format beginning with "*ssh-rsa*" - you can use instead the multi-line PEM format -. On a Windows command prompt, run the following command to retrieve the content of your SSH public key and copy it to the clipboard. `C:\> type %USERPROFILE%\.ssh\id_rsa.pub | clip` And then, paste the content in this field.

7. Click on **Next : Virtual Machine Settings >** to continue to the **Virtual Machine Settings** page.

[37] PUBLIC KEY AUTHENTICATION FOR SSH: https://www.ssh.com/ssh/public-key-authentication

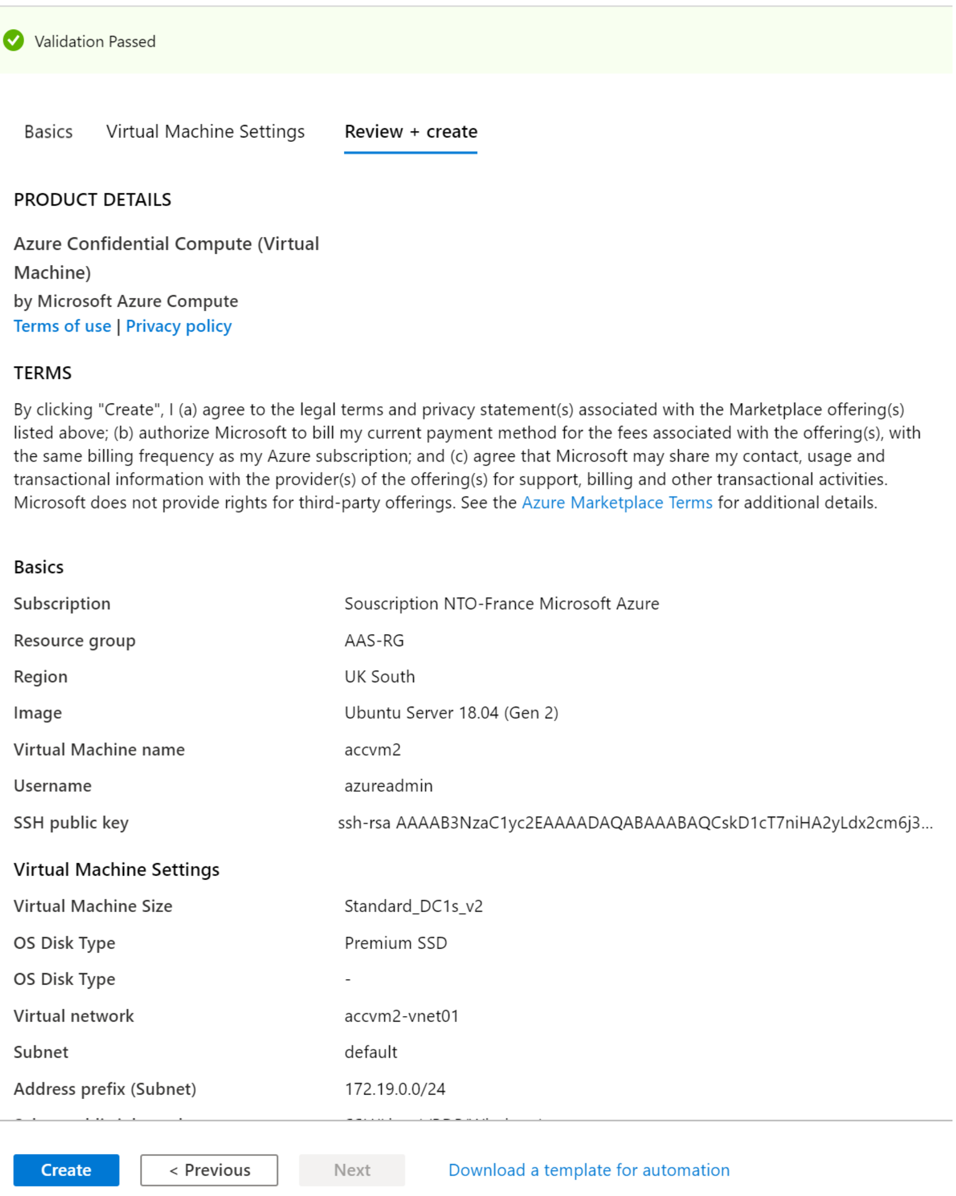


8. This second page of the form is about more specific settings for the DC-series VM, which include VM size, storage type, and virtual network details. Fill in the **Virtual Machine Settings**.

Setting	Description
Virtual Machine Size	Azure Confidential Computing (ACC) DCsv2_series VMs come in four sizes. For the sake of this guide, keep the default or select **DC2s_v2**.
OS Disk Type	Select your preferred storage type. **Premium SSD** is the default.
Virtual network	Configure the (new) virtual network (VNet) where your VM will reside. For simplicity, this starter guide will use the default settings.
Subnets	Configure the subnet. Again, this starter guide will use the defaults provided. Go to the configuration sub-menu (click on the red exclamation point) and click on **OK** to accept the default values Click on the option, then click on **OK** below the newly created subnet.
Select public inbound ports	Select **SSH (Linux)/RDP (Windows)** as you're going to use SSH. **Important note** The ports will be open for all public inbound traffic from the Internet, posing a serious security issue. For a production environment, it is recommended that you leave selected **none**; after the VM has been created and deployed, configure the VM's networking inbound port rules to open the

	required port for a specific IP Address range or enable the Microsoft Azure Security Center Just-in-time VM access.
Boot diagnostics	Leave **Disabled**.

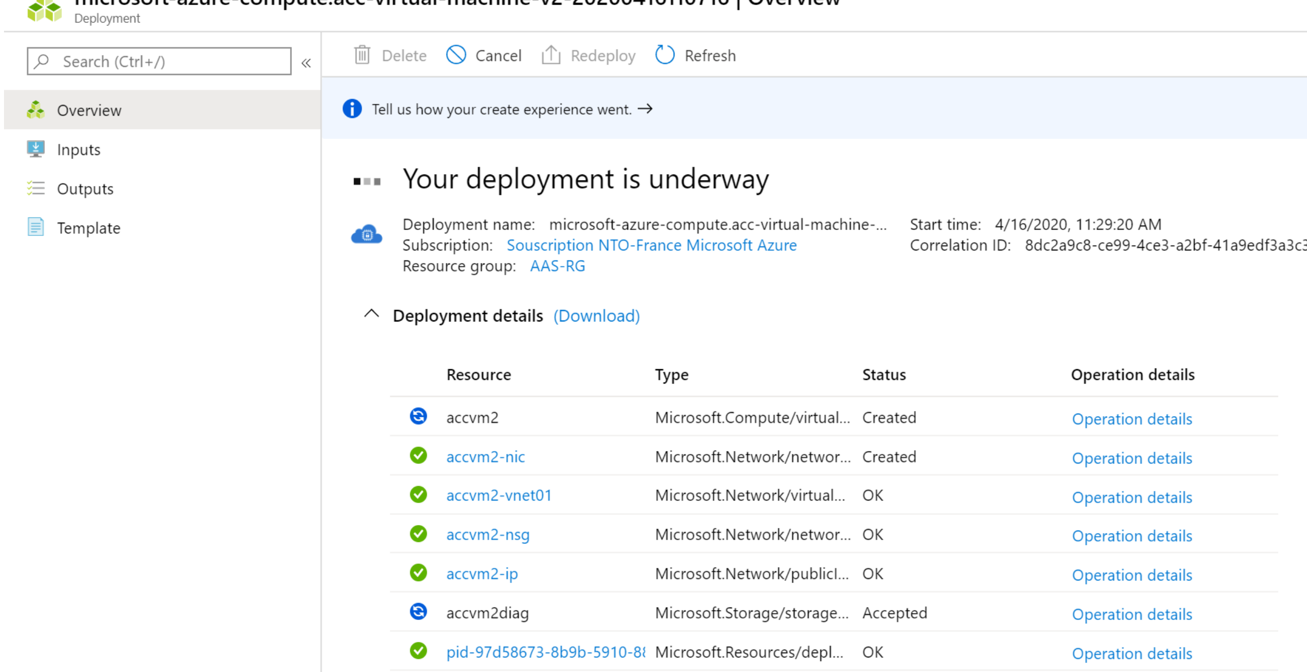
9. Click on **Next : Review + create** to continue. Validation of your configuration settings will occur on the **Summary** page.



10. If validation has passed, review your configuration.

11. Before creating and deploying your newly configured VM, carefully read the terms of use and understand any costs associated with the use of Microsoft Azure resources. When you are ready to deploy the VM, click on **Create**. Your DCsv2-series VM will be deployed on Azure.

The completion process will take approximately 10 minutes, at which time you will see a new message in the Microsoft Azure portal notifications tab.



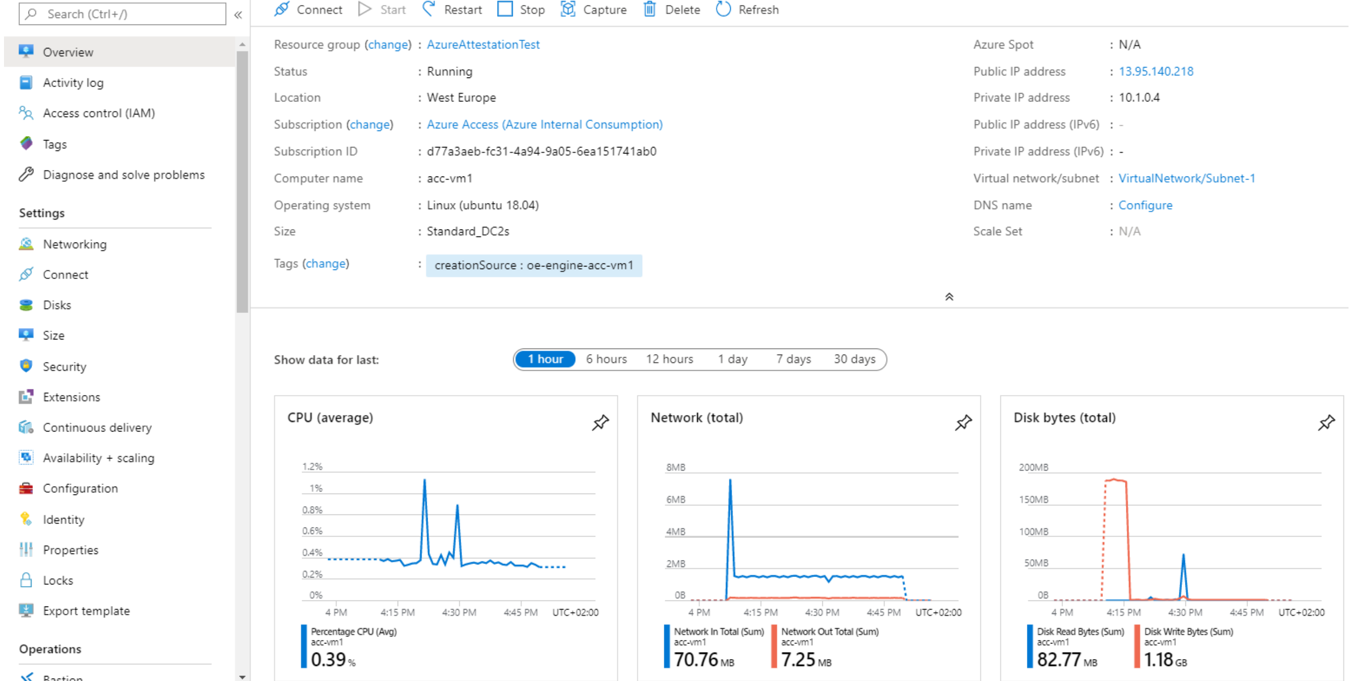
Connecting to your DC-series VM

Once your DC-series VM is online, by using a SSH client of your choice, such as OpenSSH, PuTTY, etc. you can test your remote connection to the newly created (v1 DC-series or v2 DCsv2-series VM) VM using the administrator credentials provided above. The public IP address of the VM can be found on the VM Networking page.

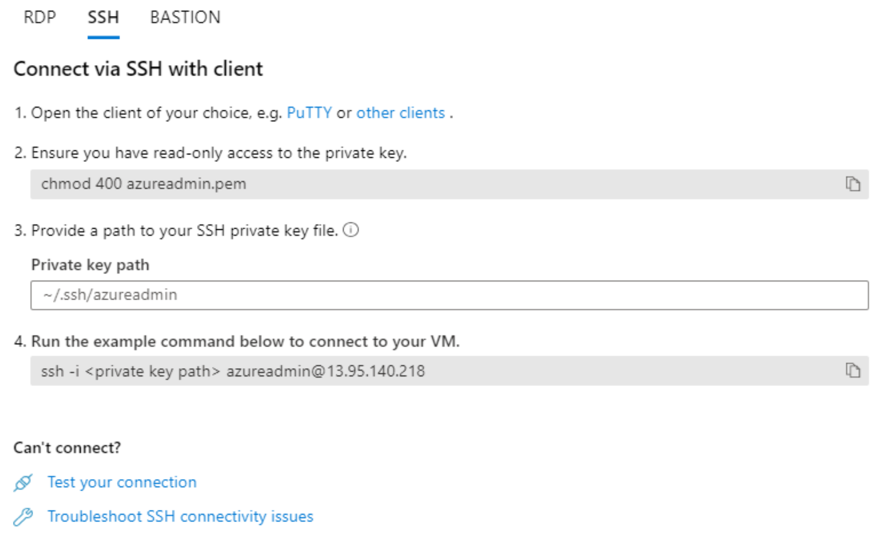
> **Note** Depending on your configuration, you may need to configure a proxy in the SSH client to connect to the virtual machine.

To connect to your VM using OpenSSH, perform the following steps:

1. From the Azure portal, search for your VM and click on it to display its menu.



2. Click on **Connect**, select SSH, and then make a note of the public IP address and the SSH connection string.



In our illustration, the IP address of the DC_series VM is 13.95.140.218, and the SSH connection string is azureadmin@13.95.140.218.

3. Open a PowerShell console, and the SSH to your VM:

```
PS C:\> ssh azureadmin@13.95.140.218
```

4. When prompted, type "*yes*". Optionally specify your passphrase if any for your private key.

Et voila! You are now connected to your DC-series VM.

```
azureadmin@acc-vm1: ~
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\philber> ssh azureadmin@13.95.140.218
The authenticity of host '13.95.140.218 (13.95.140.218)' can't be established.
ECDSA key fingerprint is SHA256:RGoxs7oWYVoG9KTDl1tJpM9mQ2nvNex/xtpWgaK9nAs.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '13.95.140.218' (ECDSA) to the list of known hosts.
Enter passphrase for key 'C:\Users\philber/.ssh/id_rsa':
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.0.0-1035-azure x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue Apr  7 15:02:51 UTC 2020

  System load:  0.09              Processes:           111
  Usage of /:   14.7% of 28.90GB   Users logged in:     0
  Memory usage: 3%                IP address for eth0: 10.1.0.4
  Swap usage:   0%

 * Kubernetes 1.18 GA is now available! See https://microk8s.io for docs or
   install it with:

     sudo snap install microk8s --channel=1.18 --classic

 * Multipass 1.1 adds proxy support for developers behind enterprise
   firewalls. Rapid prototyping for cloud operations just got easier.

     https://multipass.run/

17 packages can be updated.
0 updates are security updates.


Last login: Tue Apr  7 14:20:36 2020 from 93.3.247.151
azureadmin@acc-vm1:~$
```

At this stage, your DC-series VM is ready.

You can now clone and build the Open Enclave SDK (OESDK) from its GitHub repo. Let's see how to instantiate it on top of your DC_series VM: an older v1 DC-series or a v2 DCsv2-series.

Cloning and building the Open Enclave SDK

See [Install the Open Enclave SDK (Ubuntu 18.04)](#)

Perform the following steps:

1. From the Bash terminal, configure the Intel and Microsoft APT repositories:

```
$ echo 'deb [arch=amd64] https://download.01.org/intel-sgx/sgx_repo/ubuntu bionic main' | sudo tee
/etc/apt/sources.list.d/intel-sgx.list
$ wget -qO - https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-deb.key | sudo apt-key add -

$ echo "deb http://apt.llvm.org/bionic/ llvm-toolchain-bionic-7 main" | sudo tee
/etc/apt/sources.list.d/llvm-toolchain-bionic-7.list
$ wget -qO - https://apt.llvm.org/llvm-snapshot.gpg.key | sudo apt-key add -
```

```
$ echo "deb [arch=amd64] https://packages.microsoft.com/ubuntu/18.04/prod bionic main" | sudo tee
/etc/apt/sources.list.d/msprod.list
$ wget -qO - https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add –
```

2. Install the Intel SGX DCAP driver:

```
$ sudo apt update
$ sudo apt -y install dkms
$ wget https://download.01.org/intel-sgx/sgx-
dcap/1.5/linux/distro/ubuntuServer18.04/sgx_linux_x64_driver_1.21.bin -O sgx_linux_x64_driver.bin
$ chmod +x sgx_linux_x64_driver.bin
$ sudo ./sgx_linux_x64_driver.bin
```

Note This step also installs the az-dcap-client package which is necessary for performing remote attestation in Azure. A general implementation for using Intel DCAP outside the Azure environment is coming soon.

3. Install the Intel and Open Enclave packages and dependencies:

```
$ sudo apt -y install clang-7 libssl-dev gdb libsgx-enclave-common libsgx-enclave-common-dev
libprotobuf10 libsgx-dcap-ql libsgx-dcap-ql-dev az-dcap-client open-enclave
```

Note This may not be the latest Intel SGX DCAP driver. Please check with Intel's SGX site if a more recent SGX DCAP driver exists.

4. Install CMake

```
$ sudo snap install cmake --classic
$ cmake –version
```

You can now start studying and compiling the Open Enclave SDK sample applications. Let's see quickly how to begin with.

Using the Open Enclave SDK

As covered in the introduction, Open Enclave SDK helps you build TEE-based applications and provides you a series of sample applications that demonstrate how to develop enclave applications using Open Enclave APIs.

In your newly created DC-series VM, the Open Enclave SDK is installed to its default directory */opt/openenclave*, which contains the following folders:

Path	Description
bin	All the developer tools for developing, debugging and signing TEE-based applications using the Open Enclave SDK.
include/openenclave	Open Enclave runtime headers for use in your application enclave (*enclave.h*) and its host application (*host.h*)
include/openenclave/3rdparty	Headers for libc, libcxx and mbedlts libraries for use inside the enclave.

lib/openenclave/cmake	Open Enclave SDK CMake package for integration with your CMake projects. For example, and as illustrated in the module, Visual Studio 2017/2019 supports CMake projects for cross-platform builds and there are extensions for Visual Studio Code.
lib/openenclave/enclave	Libraries for linking into the enclave, including the libc, libcxx and mbedtls libraries for Open Enclave.
lib/openenclave/host	Library for linking into the host application process of the enclave.
lib/openenclave/debugger	Libraries used by the gdb plug-in for debugging enclaves.
share/pkgconfig	Pkg-config files for header and library includes when building TEE-based applications using the Open Enclave SDK (OESDK).
share/openenclave/samples	Sample applications' code showing how to use the Open Enclave SDK.

Note For more information, see article USING THE OPEN ENCLAVE SDK[38].

As far as the sample applications are concerned, it's advised to go through them in the order listed hereafter to progressively familiarize yourself with the Open Enclave SDK (OESDK).

azureadmin@acc-vm1: /opt/openenclave/share/openenclave/samples

```
azureadmin@acc-vm1:/opt/openenclave$ azureadmin@acc-vm1:/opt/openenclave$ cd share/openenclave//samples/
azureadmin@acc-vm1:/opt/openenclave/share/openenclave/samples$ ls -l
total 36
-rw-r--r-- 1 root root 6866 Mar 24 13:48 README.md
drwxr-xr-x 7 root root 4096 Mar 31 15:13 attested_tls
drwxr-xr-x 7 root root 4096 Mar 31 15:13 data-sealing
drwxr-xr-x 4 root root 4096 Mar 31 15:13 file-encryptor
drwxr-xr-x 4 root root 4096 Mar 31 15:13 helloworld
drwxr-xr-x 7 root root 4096 Mar 31 15:13 local_attestation
drwxr-xr-x 7 root root 4096 Mar 31 15:13 remote_attestation
drwxr-xr-x 4 root root 4096 Mar 31 15:13 switchless
azureadmin@acc-vm1:/opt/openenclave/share/openenclave/samples$
```

Sample Application	Description
HelloWorld[39]	Minimum code needed for an Open Enclave application. Help understand the basic components a TEE-based application with the Open Enclave SDK.
File-Encryptor[40]	Show how to encrypt and decrypt data inside an enclave.
Data-Sealing[41]	Introduce the Open Enclave sealing and unsealing features.
Remote Attestation[42]	Explain how the Open Enclave attestation works.

[38] USING THE OPEN ENCLAVE SDK: https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/using_oe_sdk.md

[39] HelloWorld sample: https://github.com/openenclave/openenclave/blob/master/samples/helloworld/README.md

[40] File-Encryptor sample: https://github.com/openenclave/openenclave/blob/master/samples/file-encryptor/README.md

[41] Data-Sealing sample: https://github.com/openenclave/openenclave/blob/master/samples/data-sealing/README.md

[42] Remote Attestation sample: https://github.com/openenclave/openenclave/blob/master/samples/remote_attestation/README.md

	Demonstrate an implementation of such a remote attestation between two enclaves running on different machines.
Local Attestation[43]	Explain the concept of Open Enclave local attestation. Demonstrate an implementation of local attestation between two enclaves on the same machine.
Attested TLS[44]	Explain what an Attested TLS channel is. See article WHAT IS AN ATTESTED TLS CHANNEL[45]. Demonstrate an implementation for how to establish an Attested TLS channel between i) two enclaves, and ii) one non-enclave client and an enclave.

Note For a detailed explanation of each sample application, see article OPEN ENCLAVE SDK SAMPLES[46].

All the above sample applications that come with the Open Enclave SDK installation share a similar directory structure with underneath a *host* folder for the host application and an *enclave* folder for the enclave itself, along with build instructions for two different build systems: one using GNU Make and pkg-config, the other using CMake.

However, writing files under the */opt* folder, where the Open Enclave is installed, is not allowed unless the command is running in the context of the superuser, i.e. sudo.

To build the above sample applications and avoid this sudo requirement, perform the following steps:

1. Connect to your DC-series VM as per previous activity.
2. You may want to first copy the sample applications to a user directory of your choice then build and run on those local copy. Copy them to your home directory, for example in a folder *mysamples*:

```
$ sudo cp -r /opt/openenclave/share/openenclave/samples ~/mysamples
```

3. Change the owner of the sample applications' code directory from root to your account:

```
$ sudo chown -R azureadmin ~/mysamples/
```

In our illustration, the username specified when creating the VM was **azureadmin**.

4. Before building any sample application code, you first need to source the file *openenclaverc* to setup environment variables for the Open Enclave SDK for ease of development:
   - Open Enclave SDK *pkgconfig* folder to PKG_CONFIG_PATH,
   - Open Enclave SDK *bin* folder to PATH.

```
# Copyright (c) Microsoft Corporation. All rights reserved.
# Licensed under the MIT License.

# Update PKG_CONFIG_PATH.
$ export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/openenclave/share/pkgconfig
```

[43] Local Attestation sample: https://github.com/openenclave/openenclave/blob/master/samples/local_attestation/README.md

[44] Attested TLS sample: https://github.com/openenclave/openenclave/blob/master/samples/attested_tls/README.md

[45] WHAT IS AN ATTESTED TLS CHANNEL: https://github.com/openenclave/openenclave/blob/master/samples/attested_tls/AttestedTLSREADME.md#what-is-an-attested-tls-channel

[46] OPEN ENCLAVE SDK SAMPLES: https://github.com/openenclave/openenclave/blob/master/samples/README.md

```
# Set CMake Config-package path
$ export OpenEnclave_DIR=/opt/openenclave/lib/openenclave/cmake

# Update PATH.
$ export PATH=${PATH}:/opt/openenclave/bin
```

The file *openenclaverc* is located in the folder *share/openenclave* of the Open Enclave SDK installation directory. Initialize the Open Enclave build environment.

```
$ . /opt/openenclave/share/openenclave/openenclaverc
```

Note You can use `.` in Bash to `source`.

5. Go to the sample applications' directory:

```
$ cd ~/mysamples
```

6. To build the sample applications for example using GNU Make, go into each subfolder and build and execute only an individual project, for example:

```
$ cd helloworld
$ make build
$ make run
```

Verify that the sample application runs successfully. You should see the following messages in the console: `Hello world from the enclave` and `Enclave called into host to print: Hello World!`

azureadmin@acc-vm1: ~/mysamples/helloworld

```
openssl genrsa -out private.pem -3 3072
Generating RSA private key, 3072 bit long modulus (2 primes)
.....................++++
...................................++++
e is 3 (0x03)
openssl rsa -in private.pem -pubout -out public.pem
writing RSA key
make[2]: Leaving directory '/home/azureadmin/mysamples/helloworld/enclave'
make sign
make[2]: Entering directory '/home/azureadmin/mysamples/helloworld/enclave'
oesign sign -e helloworldenc -c helloworld.conf -k private.pem
Created helloworldenc.signed
make[2]: Leaving directory '/home/azureadmin/mysamples/helloworld/enclave'
make[1]: Leaving directory '/home/azureadmin/mysamples/helloworld/enclave'
make -C host
make[1]: Entering directory '/home/azureadmin/mysamples/helloworld/host'
Compilers used: clang-7, g++
oeedger8r ../helloworld.edl --untrusted
Generating edge routines for the Open Enclave SDK.
Success.
clang-7 -g -c -fstack-protector-strong -mllvm -x86-speculative-load-hardening -I/opt/openenclave/share/pkgconfig/../../i
nclude host.c
clang-7 -g -c -fstack-protector-strong -mllvm -x86-speculative-load-hardening -I/opt/openenclave/share/pkgconfig/../../i
nclude helloworld_u.c
clang-7 -o helloworldhost helloworld_u.o host.o -L/opt/openenclave/share/pkgconfig/../../lib/openenclave/host -rdynamic
-Wl,-z,noexecstack -loehost -ldl -lpthread -lsgx_enclave_common -lsgx_dcap_ql -lsgx_urts -lssl -lcrypto
make[1]: Leaving directory '/home/azureadmin/mysamples/helloworld/host'
azureadmin@acc-vm1:~/mysamples/helloworld$ make run
host/helloworldhost ./enclave/helloworldenc.signed
Hello world from the enclave
Enclave called into host to print: Hello World!
azureadmin@acc-vm1:~/mysamples/helloworld$ _
```

Enjoy your first exploration of the Azure Confidential Computing (ACC) and the Open Enclave SDK (OESDK).

Studying and the sample applications' code will help you understand how to develop enclaves using the OESDK.

This is also the purpose of the next module.

Module 2: Developing TEE-based application in Azure

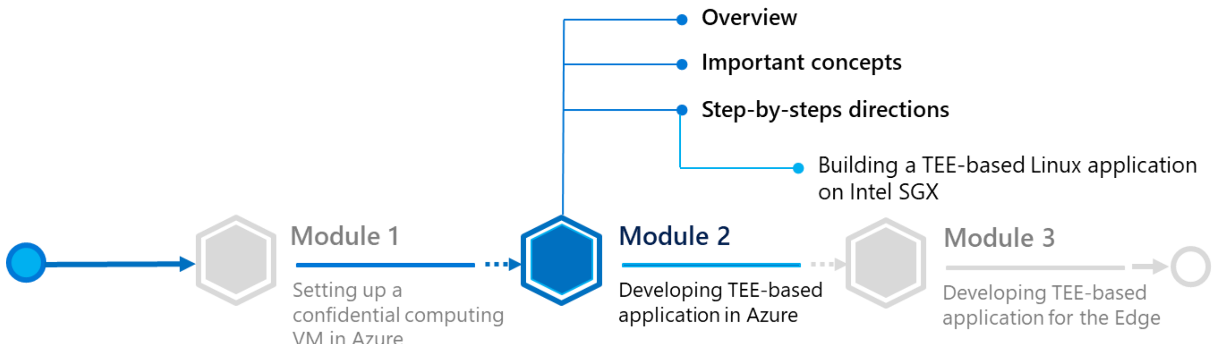
Overview

In the direct line of the previous module, this second module of this guide will illustrate the basics on how to develop Trusted Execution Environment (TEE) based application for Linux with the Open Enclave SDK (OESDK) in C and C++ on top of a DC-series VM in Azure (see sections § *Deploying a v1 DC-series VM on Azure* or § *Deploying a v2 DCsv2-series VM on* Azure above).

It's intended to help you understand the key characteristics of such an application and to illustrate the available tooling as a developer.

For these purposes, this module will more specifically cover a Linux host app and an enclave on Intel SGX.

As stated in the **Guide prerequisites**, you will use a local Windows 10 machine to develop and cross-build such applications for Linux.



You will unsurprisingly use the DC-series VM running Ubuntu 18.04 you have setup in the previous module as a build machine: this machine that is SGX-capable will thus be used as your remote compiler and linker for your application.

Important note A non-SGX machine can still be used in simulation mode.

Note For a system to be considered to be SGX enabled, it must meet all the following three conditions: i) the CPU in the system must support the Intel SGX extension, ii) the system BIOS must support Intel SGX control, and iii) Intel SGX must be enabled in the BIOS. For more information, see article DETERMINE THE SGX SUPPORT LEVEL[47].

Visual Studio will be used on the local Windows 10 machine for the integrated development environment (IDE) and you will need to configure it with the address (or name) of your Linux machine for cross-building the TEE-based application.

[47] DETERMINE THE SGX SUPPORT LEVEL: https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/SGXSupportLevel.md

This machine can also be any of the followings running Ubuntu 18.04 or Ubuntu 16.04 (64-bit) and the Open Enclave SDK:

- A remote (SGX-capable) Linux machine,
- A Linux VM running on your local Windows 10 machine.

Note To install the Open Enclave SDK, see articles INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04)[48] or INSTALL THE OPEN ENCLAVE SDK (UBUNTU 16.04)[49].

Before doing that, let's take the time to consider some important concepts first, and in particular, some of the data marshalling and unmarshalling principles and how to transfer control between the host application and the secure enclave.

Important concepts

Terminology

Let's clarify some of the commonly used terminology in the rest of this module:

- **Untrusted**. refers to code or construct that runs in the host application environment outside the enclave.
- **Trusted**. refers to code or construct that runs in the Trusted Execution Environment (TEE) inside the enclave.
- **ECALL**. A call from the host application into an interface function within the enclave.
- **OCALL**. A call made from within the enclave to the host application.
- **Generated code**. refers to code automatically generated by the Open Enclave edger8r tool through the definition/use of enclave interface definition files; i.e. EDL files (see below):
    - Boilerplate code in the normal execution environment that executes outside the enclave environment and performs functions such as loading and manipulating an enclave (e.g. destroying an enclave), and making calls (ECALLs) to an enclave and receiving calls (OCALLs) from an enclave.
    - Boilerplate code in the trusted execution environment that executes within the enclave environment and performs functions such as receiving calls (ECALLs) from the host application and making calls outside (OCALLs) the enclave, and managing the enclave itself.

Important note For information about the Open Enclave edger8r tool, see article GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R[50].

[48] INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04): https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md

[49] INSTALL THE OPEN ENCLAVE SDK (UBUNTU 16.04): https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_16.04.md

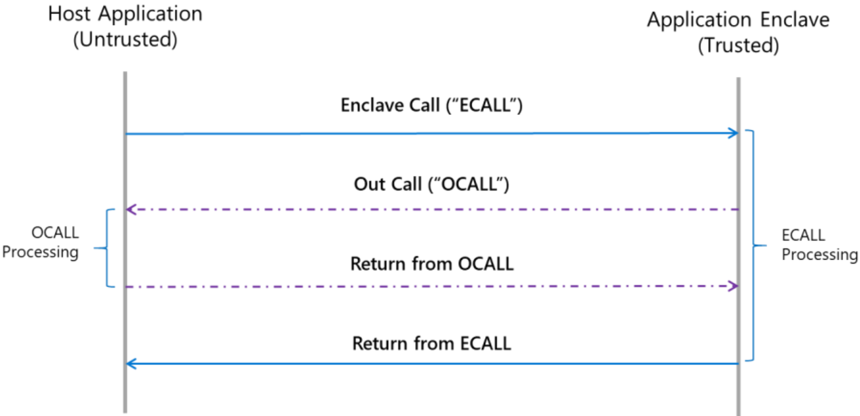
[50] GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R: https://github.com/openenclave/openenclave/blob/feature.new_platforms/docs/GettingStartedDocs/Edger8rGettingStarted.md

Enclave interface definition

As introduced above, the interface between the host application (untrusted) and the application enclave (trusted) is defined using the Enclave Definition Language or EDL. The EDL file defines the interfaces and data types the enclave will support.

The EDL file is used to define:

1. How a host application calls in to an enclave to request a secure service, i.e. an Enclave CALL or ECALL
2. And how an enclave calls into its host application to request an unsecured service, i.e. an Out CALL or OCALL.



There are thus two parts to an EDL file: the trusted section that defines the ECALLS whereas the untrusted section defines the OCALLS. While an ECALL defines entry point into the enclave, the OCALL defines the transfer of control from inside the enclave to the host application to perform system calls and other I/O operations. OCALLS could also be used in cases where the enclave needs to transfer data back to the host application.

> **Important note** An SGX enabled application should always have at least one public ECALL to enter the enclave. OCALLs are optional.

Definitions must be described using the EDL file syntax[51].

Furthermore, the same EDL file is used to define the interface between the host application and the enclave, and regardless of whether the enclave is:

1. An Intel SGX enclave,

-or-

2. An Open Portable Trusted Execution Environment (OP-TEE) Trusted Application (TA) based on ARM TrustZone to provide isolation of the TEE from the rich OS in hardware. See section § *Module 3: Developing TEE-based application for the Edge*.

An EDL file may include other EDL files and is processed using the Open Enclave edger8r tool, i.e. ooedger8r, which generates boilerplate code for you.

---

[51] ENCLAVE DEFINITION LANGUAGE FILE SYNTAX - INTEL® DEVELOPER ZONE: https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-definition-language-file-syntax

Data marshalling

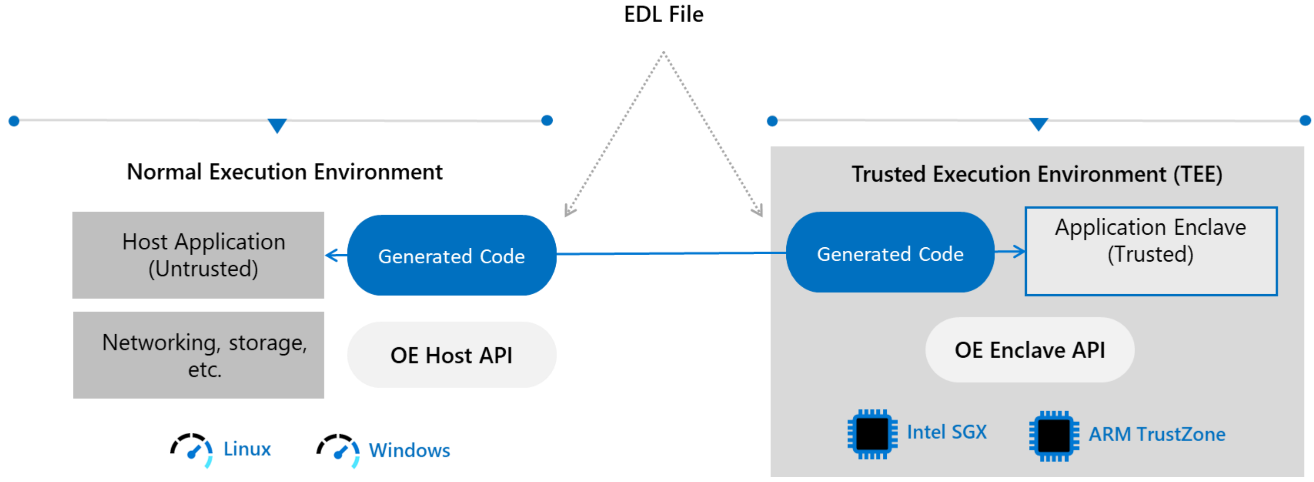
Calling into and out of enclaves is done through special methods that switch into and out of the enclave, along with the marshaling of parameters that are passed into these functions. A lot of the code necessary to handle these calls and parameter marshaling are common to all function calls. Marshaling parameters from the host application to the enclave for security purposes, and in doing so, also helps to mitigate certain processor vulnerabilities such as Meltdown and Spectre[52].

The aforementioned Open Enclave edger8r helps to define these special functions through the use of EDL file(s) and then generates boilerplate code for you. It more specifically generates five files as follows:

- A source (<*host*>*_u.c*) and a header file (<*host*>*_u.h*) to be included by the host application (when ooedger8r is executed with the `--untrusted` flag),
- Conversely, a source (<*enclave*>*_t.c*) and a header file (<*enclave*>*_t.c*) to be included by the enclave (when ooedger8r is executed with the `--trusted` flag),
- And a header file (<*host|enclave*>*_args.h*) that defines the parameters that are passed to all functions defined in the EDL file.

Note For more information on using the oeedger8r tool, see article GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R[53].

The above generated files contain code to aid in the marshalling of function calls and data across the host application/enclave boundary such that the ECALLs and OCALLs appears as normal function calls to you as a developer. The underlying platform and TEE specifics behaviors are abstracted away.



[52] Meltdown and Spectre: https://meltdownattack.com/

[53] GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R: https://github.com/openenclave/openenclave/tree/master/docs/GettingStartedDocs/Edger8rGettingStarted.md

Step-by-step directions

This module covers the following three activities:

1. Building a TEE-based Linux application on Intel SGX.
2. Building a TEE-based Linux application on a simulated ARM TrustZone environment.
3. Building a TEE-based Linux module on an Edge ARM TrustZone device.

Each activity is described in order in the next sections.

Building a TEE-based Linux application on Intel SGX

This section covers the following activities:

1. Installing and configuring Visual Studio on your Windows 10 development machine.
2. Creating a C/C++ TEE-based Linux application.
3. Modifying the TEE-based Linux application.

Each activity is described in order in the next sections.

> **Note** For more information, see articles AN INTRODUCTION TO CREATING A SAMPLE ENCLAVE USING INTEL SOFTWARE GUARD EXTENSIONS[54] and USING VISUAL STUDIO TO DEVELOP ENCLAVE APPLICATIONS FOR LINUX[55].

You will first need to setup the development environment on your local machine. Let's see how to proceed.

Installing and configuring Visual Studio on your Windows 10 development machine

This section describes how to configure on your local Windows 10 development machine a Visual Studio IDE, as it supports an Open Enclave extension as well as a remote compiler: the Open Enclave Wizard – Preview extension[56].

This extension includes preview support for TEE platforms, including Intel SGX and ARM TrustZone with a Windows or Linux host application. In addition, this preview includes support for testing your enclave under simulation when developing for Intel SGX or ARM TrustZone.

As the title of this activity indicates, you will walk through the development for Intel SGX.

Perform the following steps:

1. Install Visual Studio 2017 or Visual Studio 2019 (Community Edition[57], or any other edition). (Visual Studio 2017 is featured in the steps below. Any difference with Visual Studio 2019 if any will be highlighted.)
2. Launch Visual Studio.

---

[54] AN INTRODUCTION TO CREATING A SAMPLE ENCLAVE USING INTEL SOFTWARE GUARD EXTENSIONS: https://software.intel.com/en-us/articles/intel-software-guard-extensions-developing-a-sample-enclave-application

[55] USING VISUAL STUDIO TO DEVELOP ENCLAVE APPLICATIONS FOR LINUX: https://github.com/openenclave/openenclave/blob/feature.new_platforms/docs/GettingStartedDocs/VisualStudioLinux.md

[56] Open Enclave Wizard – Preview extension: https://marketplace.visualstudio.com/items?itemName=MS-TCPS.OpenEnclaveSDK-VSIX

[57] Visual Studio Community: https://visualstudio.microsoft.com/vs/community

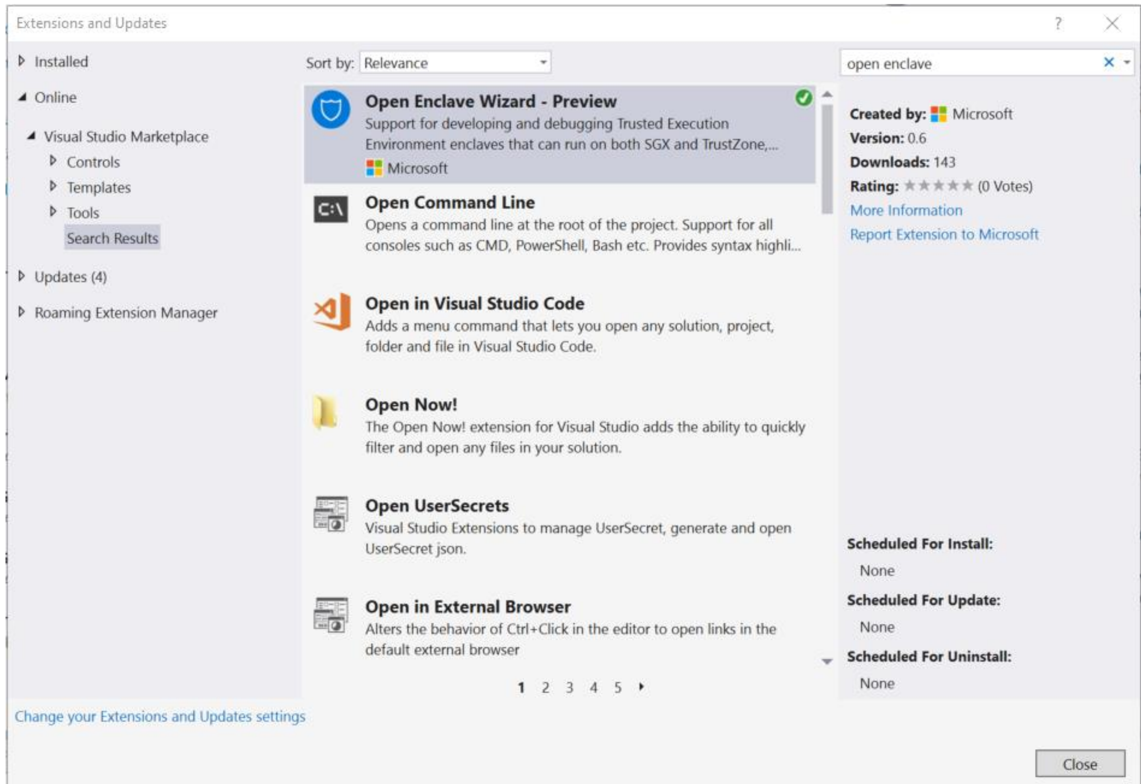
3. On the menu bar of Visual Studio, select **Tools** and then **Get Tools and Features**. This will launch **Visual Studio Installer**. (If a user control dialog pops up, click on **Yes**.)

Note In Visual Studio 2019, simply select **Tools**.

4. In **Visual Studio Installer**, select **Workloads > Other Toolsets > Linux Development with C++**. Click on **Modify**.
5. Back in Visual Studio, on the menu bar, select **Tools > Extensions and Updates > Online**.

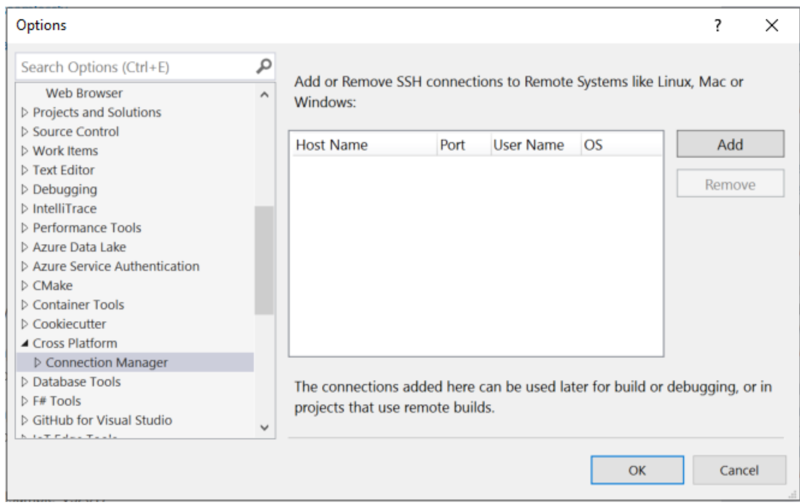
Note In Visual Studio 2019, select **Extensions -> Manage Extensions -> Online**.

6. Search for "*Open Enclave Wizard – Preview*", install the extension.

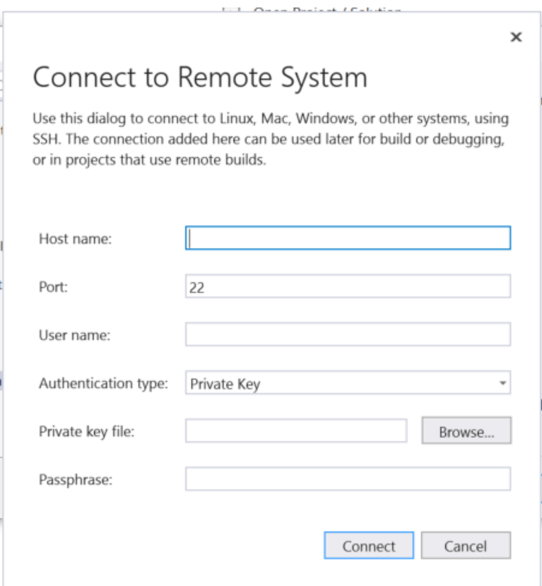


You MUST restart Visual Studio after installing the extension to complete the installation.

7. Finally, configure Visual Studio with the address (or name) of your DC-series VM (or any other Linux build machine), via **Tools > Options > Cross Platform > Connection Manager**.



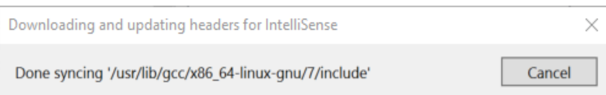
Click on **Add**. A Connect to Remote System dialog opens up.



8. Fill in the fields required to connect to your DC-series VM and click on **Connect**.

Important note Your PuTTY private key .ppk file that you generated for your key pair must be first converted to the OpenSSH format. You can use the **Conversions** > **Export OpenSSH key** from the application menu.

This step may take a minute or two, as Visual Studio will copy some files locally for use by IntelliSense.



Note For more information, see article [Connect to your target Linux system in Visual Studio][58].

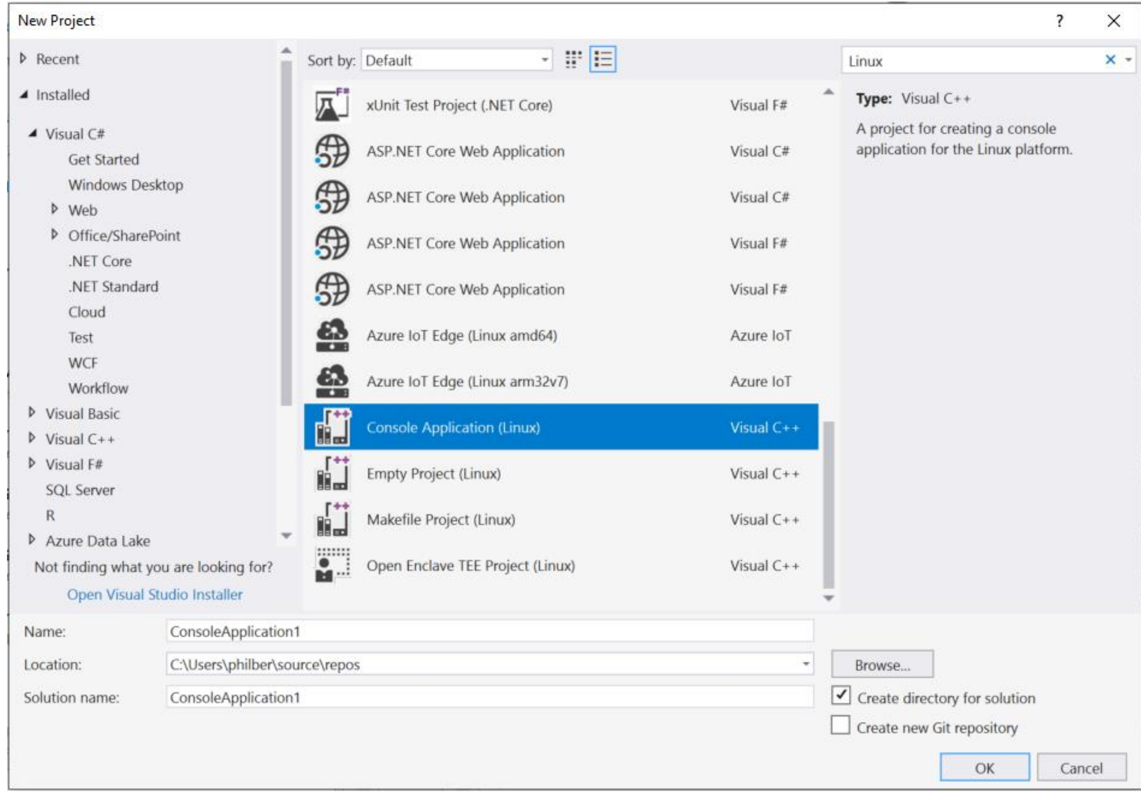
Congrats! Your Windows 10 development machine is now fully configured.

You will now walk through the process of creating a C/C++ TEE-based application that uses an enclave.

Creating a C/C++ TEE-based Linux application

Perform the following steps:

1. Create a new Linux application by choosing **File > New > Project** on the menu bar of Visual Studio. The **New Project** dialog box opens up.
2. Search the Linux console app template called "Console Application (Linux)" by typing "*Linux*".



Important note This is NOT the "Console App (.NET Core)".

Note In Visual Studio 2019, this template is called "Console App".

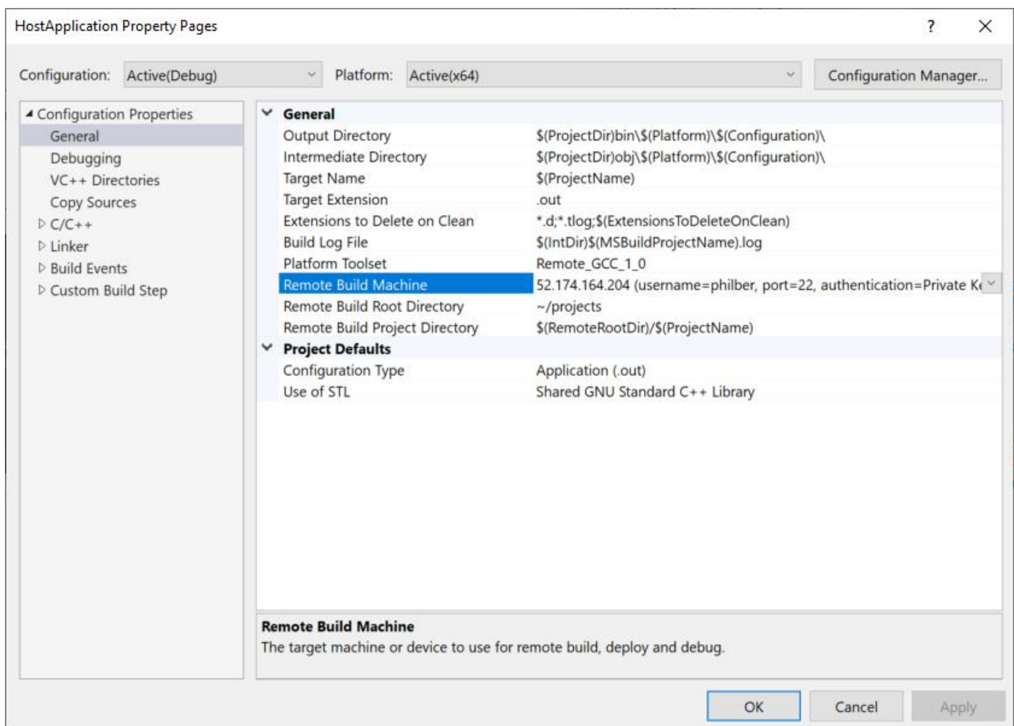
(If it is not immediately visible, the template can be found under **Installed > Visual C++ > Cross Platform > Linux**.)

a. Enter a name, for example "HostApp" in our illustration, a location, and solution name in the appropriate fields like any other Visual Studio project.

[58] Connect to your target Linux system in Visual Studio: https://docs.microsoft.com/en-us/cpp/linux/connect-to-your-remote-linux-computer?view=vs-2017

b. Click on **OK**. This will create a "Hello World" console application.

3. Configure the application project to use your Linux build environment, by right clicking on the project in the **Solution Explorer** and selecting **Properties**. The application property pages opens up.



4. Under **Configuration Properties > General -> Remote Build Machine**, explicitly set the build machine to the build machine you configured in the **Connection Manager**.

Note Due to a current Visual Studio bug, this step is required even if the correct value is shown by default. In other words, make sure the connection is shown in **bold**.

Note If you're using Visual Studio 2019 instead, you also need to update **Configuration Properties > Debugging > Remote Debug Machine** to your build machine, again due to a current Visual Studio 2019 bug.

At this point, you should be able to build and debug your newly created Hello World application.

Note For further discussion, see articles DEPLOY, RUN, AND DEBUG YOUR LINUX PROJECT[59] and LINUX DEBUGGING WALKTHROUGH[60].

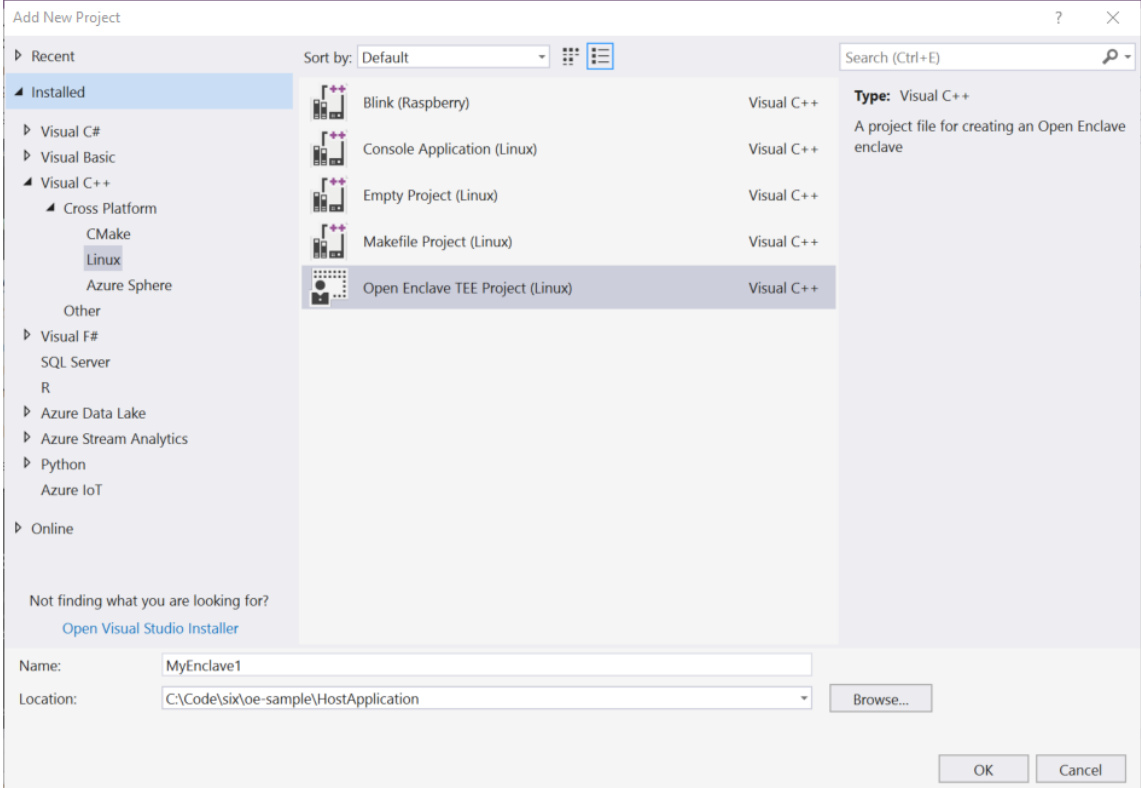
You now have your first application project, but this is only the host application. You now need to create the enclave component.

---

[59] DEPLOY, RUN, AND DEBUG YOUR LINUX PROJECT: https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017

[60] LINUX DEBUGGING WALKTHROUGH: https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017

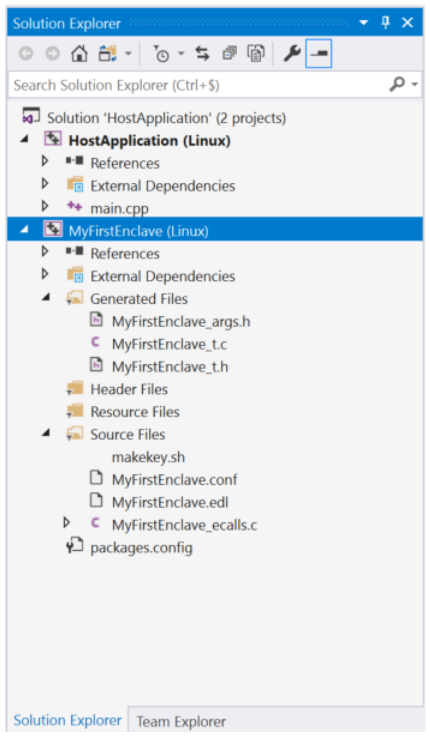
5. Right-click on your newly solution in the **Solution Explorer**, click on **Add > New project > Open Enclave TEE Project (Linux)** to add an enclave library project.



(If it is not immediately visible, look under **Installed > Visual C++ > Cross Platform > Linux**.)

6. Give it a name, for example "*MyFirstEnclave*" in our illustration and click on **OK**.

   The Wizard creates an enclave project with several files.



The sample enclave has an `ecall_DoWorkInEnclave()` method exposed to host applications, that will simply call an `ocall_DoWorkInHost()` method that will be implemented in the host application, as reflected in the EDL file (*<YourEnclaveProjectName>.edl*).

```
enclave {
    trusted {
        /* define ECALLs here. */
        public int ecall_DoWorkInEnclave();
    };

    untrusted {
        /* define any OCALLs here. */
        void ocall_DoWorkInHost();
    };
};
```

In accordance, the wizard will create a sample enclave with an `ecall_DoWorkInEnclave()` method exposed to host applications, that will simply call an `ocall_DoWorkInHost()` method that will be implemented in the host application. In this walkthrough, you'll leave this project as is for now, but afterwards you can modify it as you like.

7. Configure the enclave project to use your Linux build environment, as you did in the above step 3. At this point, the enclave would build, but cannot be run as the host application doesn't invoke it yet. At this stage, you have indeed two projects in place in solution, but they aren't linked together for now…

9. You now need to import the enclave into your host application project. To do so, right-click on your host application project in the S**olution Explorer** and select **Open Enclave Configuration > Import enclave**, then navigate to and select the EDL file (*<YourEnclaveProjectName>.edl*) in your enclave project. This file is containing all the enclave details.

Visual Studio will then link your projects by adding references between them and importing relevant libraries. As such, this step will modify your host application project settings and add some additional files to it, including a C file named *<YourEnclaveProjectName>_host.c*. This C file contains a `sample_enclave_call()` method that will load and call `ecall_DoWorkInEnclave()`, and also contains a sample implementation of a `ocall_DoWorkInHost()` method that just prints a message when called.

Although the app could be compiled and run at this point, `sample_enclave_call()` is still not called from anywhere.

8. Open the host application's file *main.cpp* and add a call to `sample_enclave_call()`. For example, update the file *main.cpp* to look like this, where the extern C declaration is needed because *main.cpp* is a C++ file whereas the *<YourEnclaveProjectName>_host.c* file is a C file:

```
include <cstdio>

extern "C" {
    void sample_enclave_call(void);
};

int main()
{
    printf("hello from LinuxApp!\n");
    sample_enclave_call();
    return 0;
}
```

8. You can now set breakpoints in Visual Studio, e.g., inside `ecall_DoWorkInEnclave()` and inside `ocall_DoWorkInHost()` and run and debug the enclave application just like any other application.

The resulting Visual Studio solution will have three configurations: **Debug**, **SGX-Simulation-Debug**, and **Release**.

The **SGX-Simulation-Debug** will work the same as **Debug**, except that SGX support will be emulated rather than using hardware support. This allows debugging on hardware that does not support SGX. The **Debug** and **Release** configurations can only be run (whether natively or in a VM) successfully on SGX-capable hardware like your DC-series VM.

For the platform, use **x64** since the Open Enclave SDK currently only supports 64-bit enclaves.

Modifying the TEE-based Linux application

Once you have the basic application working, you can modify it as desired.

For example, to define new APIs between the enclave and the host application, perform the following steps:

1. Edit the file *<YourProjectName>.edl*.
2. Define any trusted APIs (called "ECALLs" as covered in section § *Before doing* that, let’s take the time to consider some important concepts first, and in particular, some of the data marshalling and unmarshalling principles and how to transfer control between the host application and the secure enclave.
3. Important concepts) you want to call from your application in the `trusted{}` section, and in the `untrusted{}` section.
4. Likewise, define any application APIs (called "OCALLs" as covered in section § *Before doing* that, let’s take the time to consider some important concepts first, and in particular, some of the data marshalling and unmarshalling principles and how to transfer control between the host application and the secure enclave.

5. Important concepts) that you want to call from your enclave.
6. Edit the *<YourProjectName>_ecalls.c* file and fill in implementations of the ECALL(s) you added.
7. Edit your application sources and fill in implementations of the OCALL(s) you added.
8. Run and debug the enclave application just like any other application.

Let's now consider how to develop TEE-based application for the so-called "Intelligent Edge".

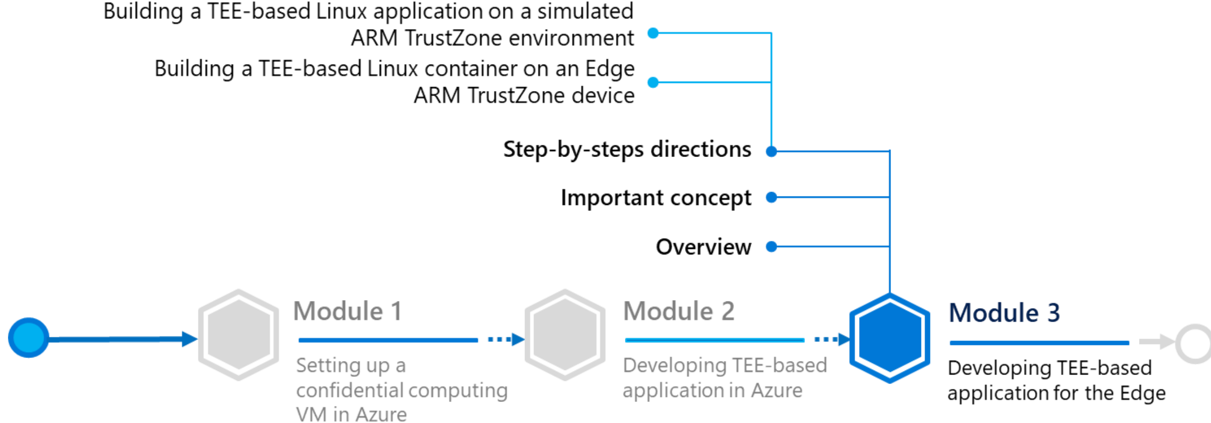
This is the purpose of the next module.

Module 3: Developing TEE-based application for the Edge

Overview

This third and last module of this guide will illustrate the basics on how to develop Trusted Execution Environment (TEE) based application (or containers) for Linux on ARM TrustZone with the Open Enclave SDK (OESDK) in C and C++.

In addition, and as such, it will help you understand the key characteristics of such applications (or containers) in the context of the so-called "Intelligent Edge".



For that purpose, this module will more specifically cover the following two types of TEE-based applications (or containers):

1. A Linux host app and an enclave on a (simulated) ARM TrustZone environment.
2. An enclave-based Linux container on an Edge ARM TrustZone device.

> **Note** This first module is partially inspired by the webcast Deep Dive: Confidential Computing in IoT using Open Enclave SDK[61] available on the IoT Show[62] on Microsoft Channel 9.

As stated in the **Guide prerequisites**, you will use still a local Windows 10 machine to develop and cross-build such applications (or containers) for Linux.

In the former case, for the sake of simplicity, you will also leverage your DC-series VM running Ubuntu 18.04 as a Linux machine, see section § *Module 1: Setting up a Confidential Computing VM in Azure*.

---

[61] Deep Dive: Confidential Computing in IoT using Open Enclave SDK: https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Confidential-Computing-in-IoT-using-Open-Enclave-SDK

[62] IoT Show: https://aka.ms/IoTShow

Similarly, this machine can also be any of the followings running Ubuntu 18.04 and the Open Enclave SDK:

- A remote Linux machine,
- A Linux VM running on your local Windows 10 machine,
- Or the Windows Subsystem for Linux (WSL) environment on your local Windows 10 machine.

Note For information about Windows Subsystem for Linux (WSL), see the WSL documentation[63], or the WSL learning resources page[64].

Moreover, Visual Studio Code will be used instead for the IDE. In addition, the Visual Studio Code Remote Development Extension Pack[65] will be used to connect in SSH to your Linux machine, or to integrate with WSL 2, for cross-building the TEE-based application.

Eventually, in the above-mentioned second and last case, a real edge device would typically be one like the Scalys TrustBox Edge[66] device, which is an industrial grade, tamper-resistant secured Azure IoT Edge[67] device optimized for confidential computing using TEEs. Azure IoT Edge is an implementation of a secure Intelligent Edge platform that is operating system, processor architecture, and hardware agnostic (see section § *Azure IoT Platform for the "Intelligent Cloud, Intelligent Cloud"* below).

This device comes pre-loaded with Open Enclave SDK and is Azure IoT Edge ready so you can immediately focus on your TEEs-based workloads. It is powered by a NXP QorIQ LS1012A[68] single core 64-bit ARM, and as such, supports the aforementioned ARM TrustZone v8 architecture.

However, to lessens the barrier for the walkthrough, and not requiring you to buy some hardware, you will use instead an Edge Linux VM[69] running Ubuntu 16.04 as a (virtual) Edge device.

---

[63] WINDOWS SUBSYSTEM FOR LINUX DOCUMENTATION: https://aka.ms/wsldocs

[64] LEARN ABOUT WINDOWS CONSOLE & WINDOWS SUBSYSTEM FOR LINUX (WSL): https://aka.ms/learnwsl

[65] Visual Studio Code Remote Development Extension Pack: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack

[66] Scalys TrustBox Edge: https://scalys.com/trustbox-industrial/

[67] Azure IoT Edge: https://azure.microsoft.com/en-us/services/iot-edge/

[68] QorIQ® Layerscape 1012A Low Power Communication Processor: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-communication-process/qoriq-layerscape-1012a-low-power-communication-processor:LS1012A

[69] Azure IoT Edge on Ubuntu: https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft_iot_edge.iot_edge_vm_ubuntu

As far as the development machine is concerned, you can use:

- Your local Windows 10 machine,
- A remote Linux machine,
- A Linux VM running on your local Windows 10 machine.

As long as your development machine can run Linux containers: a container engine on the development machine is indeed required for Linux devices.

To limit the required resources and for simplicity, you will also use the same above Edge Linux VM but one must points out both the difference and the distinction between the edge device and the development machine that are normally speaking separate ones.

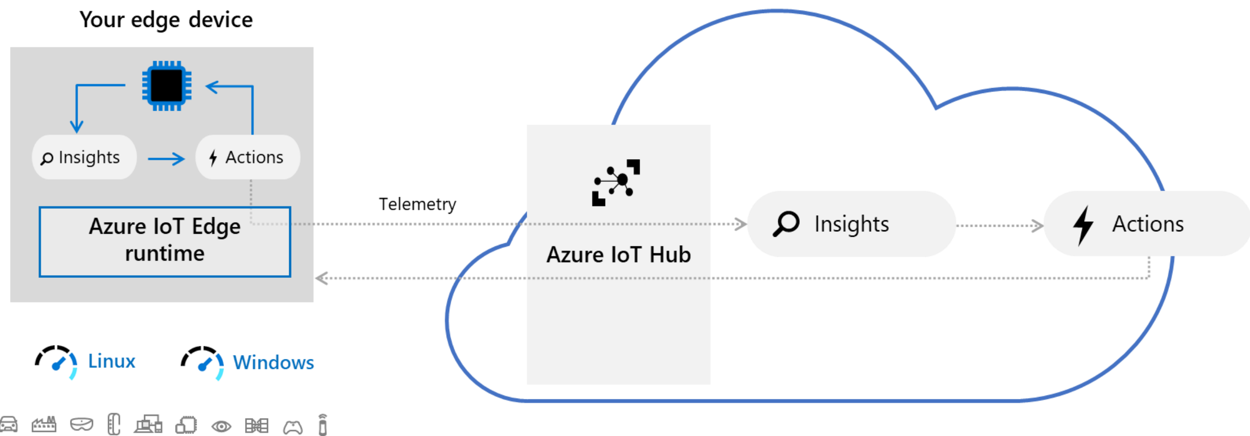
Likewise, Visual Studio Code will also be used for the IDE as well as the Visual Studio Code Remote Development Extension Pack to connect in SSH to your edge Linux VM for cross-building the TEE-based container.

Before dive in, let's consider what the Azure IoT[70] platform is for the so-called "Intelligent Cloud, Intelligent Edge".

Important concept

Azure IoT Platform for the "Intelligent Cloud, Intelligent Cloud"

Generally speaking, a cloud gateway represents a key component for the so-called "Intelligent Cloud, Intelligent Edge". It indeed provides a cloud hub for (edge) devices to connect securely to the cloud and send data. It also provides device management, capabilities, including command and control of devices.



In the Azure IoT platform, Azure IoT Hub[71] is a hosted cloud service that ingests events from (edge) devices, acting as a message broker between these devices and backend services. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management.

[70] Azure IoT : https://azure.microsoft.com/en-us/overview/iot/

[71] Azure IoT Hub: https://azure.microsoft.com/en-us/services/iot-hub/

Mobile and (Industrial) Internet of Things ((I)IoT) devices can thus securely register with the cloud, here the Azure cloud, and can connect to the cloud to send and receive data. Some devices may be edge devices that perform some data processing on the device itself or in a field gateway.

The Intelligent Edge indeed brings the power of the cloud to edge devices and demands security for trust. "Cloud-enabled computing at the edge means concentrating data, and therefore inherent value even if only momentarily. It also means moving tremendous value from the cloud to the edge in the form of intellectual property, algorithms, curated parameters, and value operations like policy enforcements, metering, and monetization. The Intelligent Edge is without a doubt a high-value bullseye to nefarious hacking and demands a high bar for security."[72]

For edge processing, we then recommend Azure IoT Edge. Azure IoT Edge is an implementation of a secure Intelligent Edge platform that is operating system, processor architecture, and hardware agnostic.

As such, Azure IoT Edge is a fully managed service built on Azure IoT Hub that allows you to remotely manage code on your devices so that you can send more of your (cloud) workloads to the edge devices - Artificial Intelligence (AI), Azure and third-party services, or your own business logic - to run on these devices via standard containers. By moving certain workloads to the edge of the network, your devices spend less time communicating with the cloud, react more quickly to local changes, and operate reliably even in extended offline periods.

> **Note** For more information, see the Microsoft Azure IoT Architecture Reference[73] guide. This guide aims to accelerate customers building IoT solutions on the Azure IoT platform, and more generally speaking on Azure, by providing a proven production ready architecture, with proven technology implementation choices, and with links to Solution Accelerator reference architecture implementations such as Remote Monitoring[74] and Connected Factory[75] on GitHub.
>
> This document offers an overview of the IoT space, recommended subsystem factoring for scalable IoT solutions, prescriptive technology recommendations per subsystems, and detailed sections per subsystem that explore use cases and technology alternatives.

Step-by-step directions

This module covers the following two activities:

1. Building a TEE-based Linux application on a simulated ARM TrustZone environment.
2. Building a TEE-based Linux module on an Edge ARM TrustZone device.

Each activity is described in order in the next sections.

---

[72] Securing the Intelligent Edge: https://azure.microsoft.com/en-us/blog/securing-the-intelligent-edge/

[73] Microsoft Azure IoT Architecture Reference guide: http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf

[74] Remote Monitoring Solution with Azure IoT: https://github.com/Azure/azure-iot-pcs-remote-monitoring-dotnet/

[75] Azure IoT connected factory preconfigured solution: https://github.com/Azure/azure-iot-connected-factory

Building a TEE-based Linux application on a simulated ARM TrustZone environment

This first section covers the following four activities:

1. Installing and configuring Visual Studio Code on your Windows 10 development machine.
2. Creating a standalone C/C++ application.
3. Building the standalone C/C++ application.
4. Debugging the standalone C/C++ application.

Each activity is described in order in the next sections.

Note For more information, see article OPEN ENCLAVE EXTENSION FOR VISUAL STUDIO CODE[76].

You will first need to setup the development environment on your local machine. Let's see how to proceed.

Installing and configuring Visual Studio Code on your Windows 10 development machine

This section describes how to configure on your local Windows 10 machine a Visual Studio Code IDE.

In addition, you will need to install some Visual Studio Code extensions:

1. Locally, the Visual Studio Code Remote Development extension pack[77] that allows you to open any folder in a container on a remote machine like your DC-series VM in Azure, A Linux machine elsewhere, or in the local Windows Subsystem for Linux (WSL) of your Windows 10 local machine. This pack includes three extensions:
    a. Visual Studio Code Remote - SSH[78]. It allows you to work with source code in any location by opening folders on a remote machine/VM using SSH and supports connecting to x86_64 Linux SSH servers like your above DC-series VM.
    b. Visual Studio Code Remote - WSL[79]. It allows you to get a Linux-powered development experience from the comfort of Windows 10 by opening any folder in WSL.
    c. Visual Studio Code Remote - Containers[80]. It allows you to work with a sandboxed toolchain or container-based application by opening any folder inside (or mounted into) a container.
2. Remotely, the Open Enclave extension for Visual Studio Code[81], supporting the Open Enclave SDK, including development, debugging, emulators, and deployment.

---

[76] OPEN ENCLAVE EXTENSION FOR VISUAL STUDIO CODE: https://github.com/openenclave/openenclave/blob/feature.new_platforms/new_platforms/vscode-extension/README.md

[77] Remote Development Extension Pack: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack

[78] Visual Studio Code Remote – SSH: https://aka.ms/vscode-remote/download/ssh

[79] Visual Studio Code Remote – WSL: https://aka.ms/vscode-remote/download/wsl

[80] Visual Studio Code Remote – Containers: https://aka.ms/vscode-remote/download/containers

[81] Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-Open Enclave

You will create a Standalone project on your Linux machine, once connected to it in SSH. For that purpose, in the suggested configuration, you will need to install on your DC-series VM all the requirements[82] listed.

Perform the following steps:

1. Install Visual Studio Code or Visual Studio Code – Insiders[83].
2. Install Git for Windows[84] (2.10 or later) and make sure that long paths are enabled:

```
$ git config --global core.longpaths true
```

3. Install an OpenSSH compatible SSH client[85] if one is not already present on your local Windows 10 machine. See section § *Installing OpenSSH on Windows 10*.
4. Install the Remote Development extension pack.

Note For details on setting up and working with each specific extension of the Remote Development extension pack, see articles REMOTE DEVELOPMENT USING SSH[86], DEVELOPING IN WSL[87], and DEVELOPING INSIDE A CONTAINER[88]. For troubleshooting tips and tricks for each of these extensions, see article REMOTE DEVELOPMENT TIPS AND TRICKS[89].

5. **Visual Studio Code uses SSH configuration files[90] and requires SSH key based authentication to connect to your host.**
6. Back in Visual Studio Code, press F1 or CTRL-Shift-P to open the Command Palette, and run `Remote-SSH: Open Configuration File...`

[82] Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements

[83] Visual Studio Code – Insiders: https://code.visualstudio.com/insiders

[84] Git for Windows: https://git-for-windows.github.io/

[85] INSTALLATION OF OPENSSH FOR WINDOWS SERVER 2019 AND WINDOWS 10: https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse

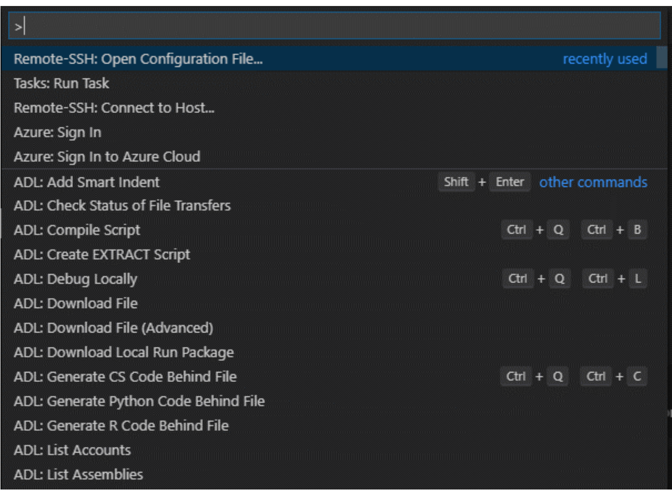
[86] REMOTE DEVELOPMENT USING SSH: https://code.visualstudio.com/docs/remote/ssh

[87] DEVELOPING IN WSL: https://code.visualstudio.com/docs/remote/wsl

[88] DEVELOPING INSIDE A CONTAINER: https://code.visualstudio.com/docs/remote/containers

[89] REMOTE DEVELOPMENT TIPS AND TRICKS: https://code.visualstudio.com/docs/remote/troubleshooting#_installing-a-supported-ssh-client

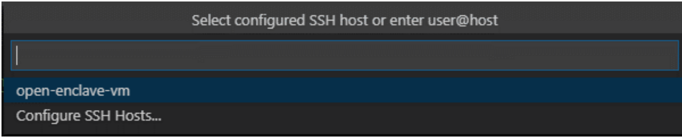
[90] ssh_config: https://linux.die.net/man/5/ssh_config



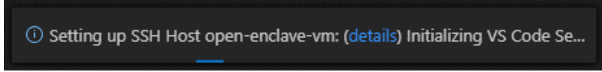
7. Select the SSH config file you wish to change, for example *%USERPROFILE%\.ssh\config*, and add (or modify) a host entry in the config file as follows:

```
# Read more about SSH config files: https://linux.die.net/man/5/ssh_config
Host <name-of-your-dc-series-vm-here>
    User <your-user-name-on-your-dcseries-vm>
    HostName <dc-series-vm-ip-goes-here>
    IdentityFile c:\users\<your-user-name-on-Windows10>\.ssh\id_rsa
```

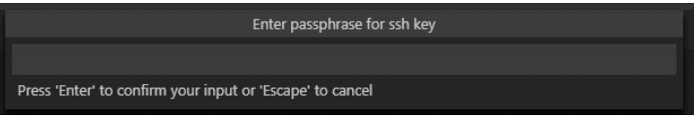
8. Press CTRL+S to save the config file.
9. Open a remote SSH session to your DC-series VM:
   a. Press F1 or CTRL-Shift-P to specify a command, run `Remote-SSH: Connect to SSH Host...`, and select your DC-series VM name in the list.

   A SSH session to your DC-series VM is opening. Visual Studio Code will now continue to configure it.

   b. When prompted, type your passphrase for your SSH key and press ENTER.

   Once finished, you now see a SSH indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!

   SSH: open-enclave-vm

ET voila! Any Visual Studio Code operations you perform in this window will be executed in the SSH environment, everything from editing and file operations, to debugging, using terminals, and more.

10. Ensure that the requirements[91] are met for the Open Enclave extension for Visual Studio Code:

    a. In Visual Studio Code, make sure that the Visual Studio Code Native Debug extension[92] is installed. Press CRTL-p and run `ext install webfreak.debug` in Visual Studio Code and install GDB/LLDB.

    b. Install CMake 3.12 or above[93], currently 3.15:

        i. From Visual Studio Code, click on **Terminal** > **New Terminal**.

        ii. From the terminal console, download the archive file *cmake-3.15.2.tar.gz* and compile it:

```
$ wget http://www.cmake.org/files/v3.15/cmake-3.15.2.tar.gz
$ tar -xvzf cmake-3.15.2.tar.gz
$ cd cmake-3.15.2/
$ ./configure
$ make
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL                                1: bash

philber@open-enclave-vm:~$ wget http://www.cmake.org/files/v3.15/cmake-3.15.2.tar.gz
--2019-08-12 13:03:51--  http://www.cmake.org/files/v3.15/cmake-3.15.2.tar.gz
Resolving www.cmake.org (www.cmake.org)... 66.194.253.19
Connecting to www.cmake.org (www.cmake.org)|66.194.253.19|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://cmake.org/files/v3.15/cmake-3.15.2.tar.gz [following]
--2019-08-12 13:03:51--  http://cmake.org/files/v3.15/cmake-3.15.2.tar.gz
Resolving cmake.org (cmake.org)... 66.194.253.19
Reusing existing connection to www.cmake.org:80.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://cmake.org/files/v3.15/cmake-3.15.2.tar.gz [following]
--2019-08-12 13:03:51--  https://cmake.org/files/v3.15/cmake-3.15.2.tar.gz
Connecting to cmake.org (cmake.org)|66.194.253.19|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9269449 (8.8M) [application/x-gzip]
Saving to: 'cmake-3.15.2.tar.gz.2'

cmake-3.15.2.tar.gz.2              100%[==========================================================================>]  8.84M  9.68MB/s    in 0.9s

2019-08-12 13:03:53 (9.68 MB/s) - 'cmake-3.15.2.tar.gz.2' saved [9269449/9269449]

philber@open-enclave-vm:~$
```

        iii. Make's install command installs `cmake` by default in the folder */usr/local/bin/cmake*. Run the following command to install (copy) the binary and libraries to the new destination:

```
$ sudo make install
```

        iv. If you haven't already installed a newer `cmake` installation, run the following command to tell your distro, e.g. Ubuntu in our illustration, that the `cmake` command is now being replaced by an alternative installation:

---

[91] Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements

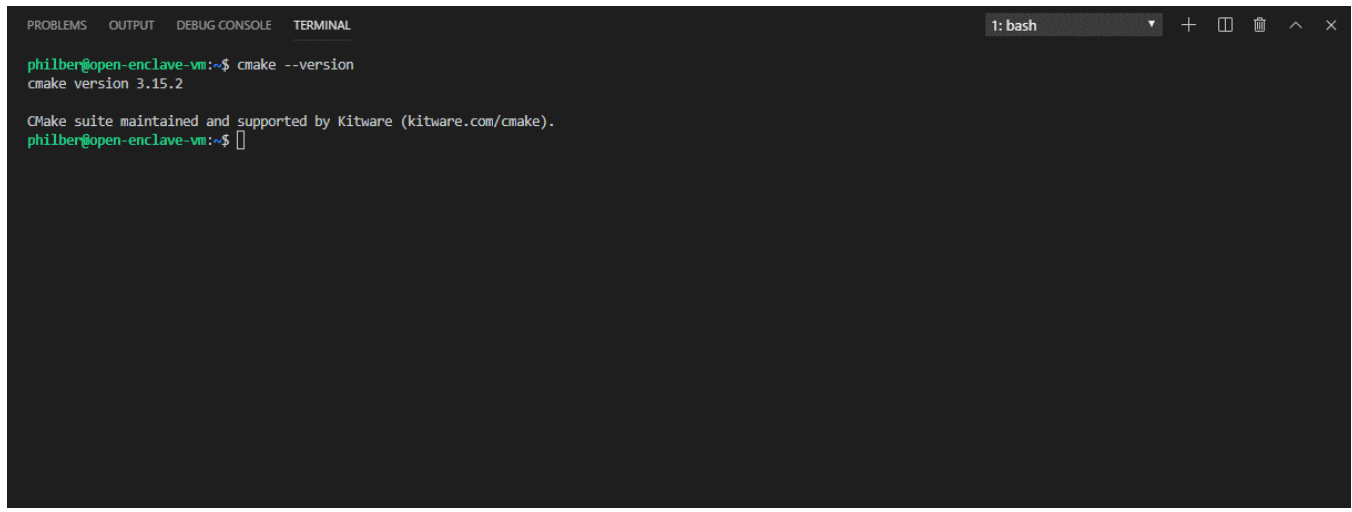
[92] Native Debug extension: https://marketplace.visualstudio.com/items?itemName=webfreak.debug

[93] CMake download: https://cmake.org/download/

```
$ sudo update-alternatives --install /usr/bin/cmake cmake /usr/local/bin/cmake 1 --force
```

Verify the cmake version using the following command:

```
$ cmake --version
```



c. Install the required build components from the bash shell:

```
$ sudo apt update && sudo apt install -y build-essential cmake gcc-arm-linux-gnueabihf gcc-aarch64-
linux-gnu g++-arm-linux-gnueabihf g++-aarch64-linux-gnu gdb-multiarch python
```

When prompted, type your password.

Furthermore, you may get the following warning and error:

```
W: GPG error: https://packages.microsoft.com/repos/azure-cli bionic InRelease: The following
signatures couldn't be verified because the public ckey is not available: NO_PUBKEY EB3E94ADBE1229CF
E: the repository 'https://packages.microsoft.com/repos/azure-cli bionic InRelease' is not signed.
N: Updating from such a repository can't be done securely, and its therefore disabled by default.
N: See apt-secure(8) manpage for repository creation and user configuration details.
```

It happens when you don't have a suitable public key for this repository.

To solve this problem, perform the following steps:

i. Use this command, which retrieves the key from ubuntu key server:

```
$ gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv EB3E94ADBE1229CF
```

ii. And then this one, which adds the key to apt trusted keys:

```
$ gpg --export --armor EB3E94ADBE1229CF | sudo apt-key add –
```

iii. Rerun the above command to install the required build components.

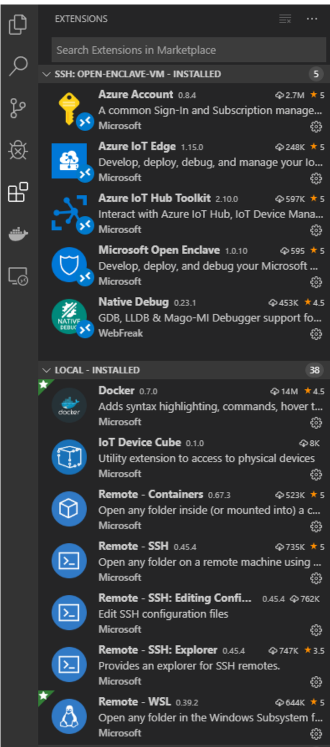
d. Ensure that all QEMU dependencies are installed. QUEMU[94] is a system emulator that can run ARM TrustZone Trusted Applications (TAs) on a x64 machine as though they were running on a TrustZone-capable hardware.

In the terminal console, run:

```
$ sudo apt update && sudo apt install -y libpixman-1-0 zlib1g libc6 libfdt1 libglib2.0-0 libpcre3
libstdc++6
```

11. Back in Visual Studio code, on the menu bar, install the Open Enclave extension for Visual Studio Code. When prompted, click on **Install**.

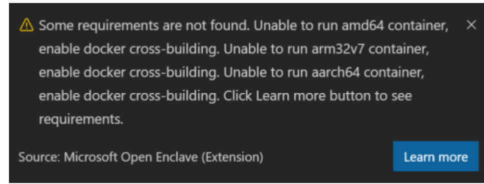
    Visual Studio Code runs extensions in one of two places: i) locally on the UI / client side, or ii) remotely in the SSH session. While extensions that affect the Visual Studio Code UI, like themes and snippets, are installed locally, most extensions needed here will reside inside your DC-series VM. This will be the case for the Open Enclave extension for Visual Studio Code.



12. Use the `Microsoft Open Enclave: Check System Requirements` command - commands can be found using F1 or CTRL-Shift-P to validate your system.

    The command will query whether the required tools and the required versions are present on your system. Any unmet requirements will be presented in a Visual Studio Code warning window.

[94] QEMU: https://www.qemu.org/



Otherwise, you should see instead the message "Your system meets the requirements".

Creating a standalone C/C++ application

Perform the following steps:

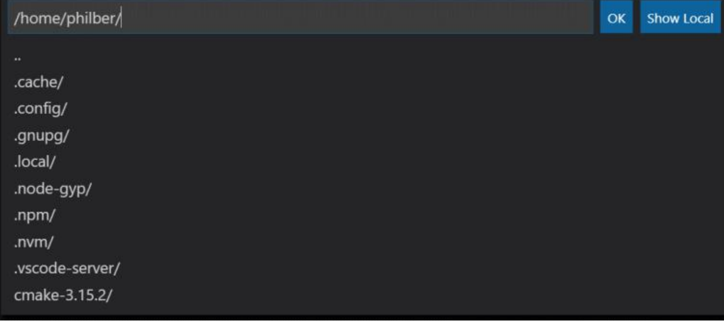
1. Launch Visual Studio Code.
2. As already stated, and illustrated above, the Visual Studio Code Remote - SSH extension lets you connect to your DC-series VM as your full-time development environment right from Visual Studio Code. You can develop in a Linux-based environment, use Linux specific toolchains and utilities, and run and debug your Linux-based applications all from the comfort of Windows.

   Open a remote SSH session to your DC-series VM:

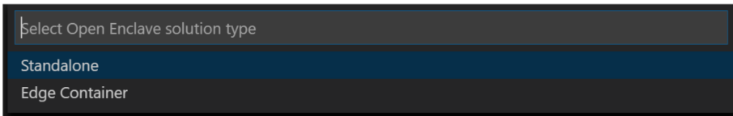
   a. Press F1 or CTRL-Shift-P to specify a command, run `Remote-SSH: Connect to SSH Host...`, and select your DC-series VM name in the list.
   b. When prompted, type your passphrase for your SSH key and press ENTER.

Note For more information, see article Remote Development using SSH[95].

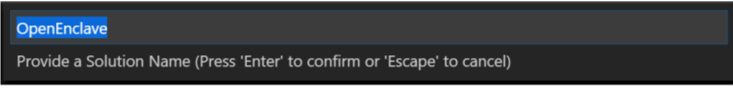
3. Use the `Microsoft Open Enclave: New Open Enclave Solution` command - commands can be found using F1 or CTRL-Shift-P - to create your first new standalone application.



4. Specify the folder in which you want to create the application.



5. When invited to choose the option, create a **Standalone** project.



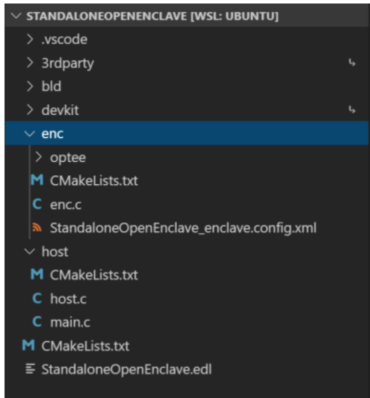
[95] REMOTE DEVELOPMENT USING SSH: https://code.visualstudio.com/docs/remote/ssh

6. Provide a name for the host application/enclave, for example "*Standalone*", and press ENTER to confirm.

A new solution will be created in the folder you've selected.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                   Microsoft Open Enclave
msiot-vscode-openenclave.checkSystemRequirements:
msiot-vscode-openenclave.checkSystemRequirements:
msiot-vscode-openenclave.checkSystemRequirements:
msiot-vscode-openenclave.newSolution:
msiot-vscode-openenclave.newSolution:
Creating base files
Base files created
Creating standalone files
Creating build folders
Downloading SDK (this is infrequent)
Downloading SDK (this is infrequent). Cleaning up SDK folder if needed
Downloading SDK (this is infrequent). Cloning SDK from git
Executing: git clone --recursive --branch feature.new_platforms https://github.com/microsoft/openenclave.git "/home/philber/.vscode-server/data/User/
globalStorage/ms-iot.msiot-vscode-openenclave/1.0.10/3rdparty/openenclave"
Cloning into '/home/philber/.vscode-server/data/User/globalStorage/ms-iot.msiot-vscode-openenclave/1.0
                                   Microsoft Open Enclave: Downloading SDK (this is infrequent). Cloning...
```

That solution will contain both the host and enclave as well as the required EDL file. The solution appears to somehow like the one in the previous activity.



The EDL file is as follows:

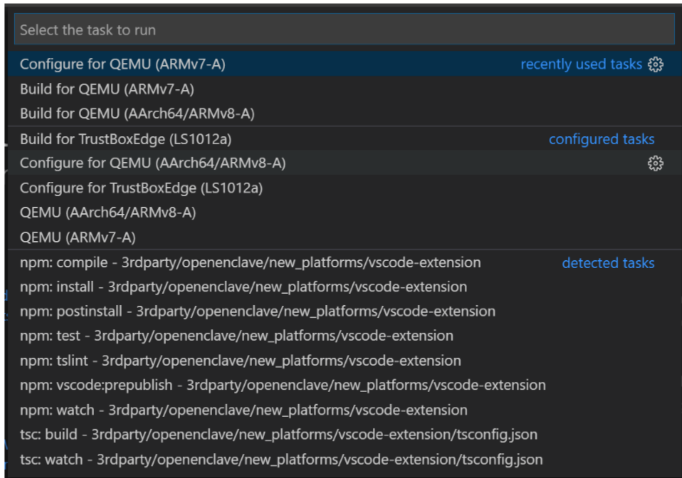
```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License.

enclave {
    from "openenclave/stdio.edl" import *;

    trusted {
        /* define ECALLs here. */
        public int ecall_handle_message([in, string] char *input_msg, [out, count=enclave_msg_size]
char *enclave_ms, unsigned int enclave_msg_size);
    };

    untrusted {
        /* define OCALLs here. */
        int ocall_log([in, string] char *msg);
    };
};
```

Building the standalone C/C++ application

As already noticed, and considering previous requirements, the underlying system used in Visual Studio Code to build is CMake.

Perform the following steps

1. Once still connect to your DC-series VM via a SSH session, press F1 or CTRL-Shift-P to specify a command.
2. Select `Tasks: Run Task`. You should see tasks configured and build for each target: **ARMv7-A** (32-bit), **AArch64/ARMv8-A** (64-bit), and **TrustBoxEdge (LS1012a)**.



3. Select `Configure for AArch64/ARMv8-A`.

Note For more information, see document TRUSTZONE TECHNOLOGY FOR ARM V8-M ARCHITECTURE[96].

The configure task will invoke `cmake` to create the required build files. This is only required to be run once.

4. Select `Build for AArch64/ARMv8-A`. The build task will do the actual compiling and linking.

---

[96] TRUSTZONE TECHNOLOGY FOR ARM V8-M ARCHITECTURE:
https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                              2: Task - Build for QEMU

                  from /home/philber/samples/Standalone/3rdparty/openenclave/3rdparty/googletest/googletest/src/gtest-all.cc:38:
/usr/arm-linux-gnueabihf/include/c++/7/bits/stl_iterator.h: In function '_Iterator std::__niter_base(__gnu_cxx::__normal_iterator<_Iterator, _Container>) [with _Iterator = const doubl
e*; _Container = std::vector<double>]':
/usr/arm-linux-gnueabihf/include/c++/7/bits/stl_iterator.h:986:5: note: parameter passing for argument of type '__gnu_cxx::__normal_iterator<const double*, std::vector<double> >' chan
ged in GCC 7.1
     __niter_base(__gnu_cxx::__normal_iterator<_Iterator, _Container> __it)
     ^~~~~~~~~~~~
[ 93%] Linking CXX static library ../../../../../out/lib/libgtest.a
[ 93%] Built target gtest
[ 93%] Generating oetests_u.h, oetests_u.c
Generating edge routines for the Open Enclave SDK.
Success.
Scanning dependencies of target oetests_host
[ 93%] Building CXX object 3rdparty/openenclave/new_platforms/tests/oetests/host/CMakeFiles/oetests_host.dir/main.cpp.o
[ 93%] Building CXX object 3rdparty/openenclave/new_platforms/tests/oetests/host/CMakeFiles/oetests_host.dir/OEEnclaveTest.cpp.o
[ 96%] Building CXX object 3rdparty/openenclave/new_platforms/tests/oetests/host/CMakeFiles/oetests_host.dir/TrustedAppTest.cpp.o
[100%] Building C object 3rdparty/openenclave/new_platforms/tests/oetests/host/CMakeFiles/oetests_host.dir/oetests_u.c.o
[100%] Building CXX object 3rdparty/openenclave/new_platforms/tests/oetests/host/CMakeFiles/oetests_host.dir/OEHostTest.cpp.o
[100%] Building CXX object 3rdparty/openenclave/new_platforms/tests/oetests/host/CMakeFiles/oetests_host.dir/sockets_test.cpp.o
[100%] Linking CXX executable ../../../../../../out/bin/oetests_host
[100%] Built target oetests_host

Terminal will be reused by tasks, press any key to close it.
```

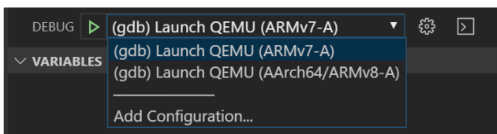
Debugging the standalone C/C++ application

Debugging your standalone project's enclave is easy as you will see now.

> **Important note** Ensure that all of the QEMU dependencies are installed in your development environment. See step 4.d in above section § *Installing and configuring Visual Studio Code on your Windows 10 development machine*.

Perform the following steps:

1. Set breakpoints in the files you wish to debug. Breakpoints in the enclave may only be added before the emulator (QEMU) starts or when the debugger is already broken inside the enclave.
2. Choose the architecture you are interested in debugging:
   a. Navigate to the Visual Studio `Debug` view – Use CTRL-Shift-D -.



   b. Select `(gdb) Launch QEMU (AArch64/ARMv8-A)` from the debug configuration dropdown.
3. You can simply hit `F5`. This will run cmake configuration, run the build, start QEMU, and load the host and enclave symbols into an instance of the debugger.
4. Open the terminal view.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                        3: ARMv8-A)
[    1.141958] Freescale USDPAA process IRQ driver
[    1.148187] optee: probing for conduit method from DT.
[    1.148858] optee: revision 3.3 (2c8b3b02)
[    1.153461] optee: initialized driver
[    1.160823] NET: Registered protocol family 17
[    1.161364] 8021q: 802.1Q VLAN Support v1.8
[    1.162080] 9pnet: Installing 9P2000 support
[    1.164052] Key type dns_resolver registered
[    1.166587] registered taskstats version 1
[    1.173522] input: gpio-keys as /devices/platform/gpio-keys/input/input0
[    1.176345] rtc-efi rtc-efi: setting system clock to 2019-08-12 09:45:06 UTC (1565603106)
[    1.181996] ALSA device list:
[    1.182095]   No soundcards found.
[    1.184040] uart-pl011 9000000.pl011: no DMA platform data
[    1.238765] Freeing unused kernel memory: 1280K
Starting logging: OK
Initializing random number generator... [    1.753660] random: dd: uninitialized urandom read (512 bytes read)
done.
Starting tee-supplicant...
Starting network: OK
Trying to copy host TAs...

Welcome to Buildroot, type root to login
buildroot login:
```

5. Log into QEMU using root (no password is required).
6. Start the host process by entering "`/mnt/host/bin/<YourOpenEnclaveSolutionName>`", for example in our illustration:

```
/mnt/host/bin/Standalone
```

```
enc > C enc.c
1   // Copyright (c) Microsoft Corporation. All rights reserved.
2   // Licensed under the MIT License.
3
4   #include <openenclave/enclave.h>
5   #include <openenclave/bits/stdio.h>
6
7   // Include the trusted Standalone header that is generated
8   // during the build. This file is generated by calling the
9   // sdk tool oe_edger8r against the Standalone.edl file.
10  #include "Standalone_t.h"
11
12  // This is the function that the host calls. It wraps
13  // a message from the host before calling back out to
14  // the host to print a message from there.
15  int ecall_handle_message(char *input_msg, char *enclave_msg, unsigned int enclave_msg_size)
16  {
17      if (snprintf(enclave_msg, enclave_msg_size, "{ \"enclave\": %s, \"signature\": \"<TODO>\" }", input_msg) < 0)
18      {
19          fprintf(stderr, "message handling failed\n");
20          return 1;
21      }
22
23      // Call back into the host
24      int retval = 0;
25      oe_result_t result = ocall_log(&retval, enclave_msg);
26      if (result != OE_OK)
27      {
28          fprintf(
29              stderr,
30              "Call to ecall_handle_message failed: result=%u (%s)\n",
31              result,
32              oe_result_str(result));
33          return 1;
34      }
35
```

Once you have the standalone application working, you can modify it as desired.

Building a TEE-based Linux module on an Edge ARM TrustZone device

As already introduced, the Intelligent Edge brings the power of the cloud to mobile and (Industrial) Internet of Things ((I)IoT) devices and demands security for trust.

So, let's now considered more specifically TEE-based application. As already stressed, we recommend Azure IoT Edge for edge processing (see section § *Azure IoT Platform for the "Intelligent Cloud, Intelligent Cloud"* earlier in this document), and consequently the Azure IoT Edge will be featured in this section.

With an IoT Edge device as a target, the TEE-based code will take the form of an Azure IoT Edge module. An Azure IoT Edge module, or sometimes just module for short, is a container that contains executable code. You can deploy one or more modules to an Azure IoT Edge device. Modules perform specific tasks like ingesting data from sensors, performing data analytics or data cleaning operations, or sending messages to an IoT hub.

> **Note** For more information, see article UNDERSTAND AZURE IOT EDGE MODULES[97].

As such, this section and illustration walks through what it takes to develop and deploy your own TEE-based module to an IoT Edge device.

More specifically, this section covers the following ten activities:

1. Setting up an Azure IoT environment with an edge device.
2. Setting up your development machine.
3. Configuring your development environment for the Open Enclave SDK.
4. Creating an Edge container C/C++ project.
5. Providing your registry credentials to the IoT Edge agent.
6. Reviewing the generated host program code for the Azure IoT Edge module.
7. Selecting your target architecture for the Azure IoT Edge module.
8. Building the Azure IoT Edge module.
9. Pushing the Azure IoT Edge module to your container registry.
10. Deploying the Azure IoT Edge module to your actual Azure IoT Edge device.

Each activity is described in order in the next sections.

> **Note** For more information, see articles TUTORIAL: DEVELOP IOT EDGE MODULES FOR LINUX DEVICES[98] and TUTORIAL: DEVELOP A C IOT EDGE MODULE FOR LINUX DEVICES[99].

Setting up an Azure IoT environment with an edge device

As already introduced, you will create an Azure IoT Edge device using a Linux VM.

For that purpose, perform all the instructions outlined in section § *Setting up a core Azure IoT environment* in Appendix. Prerequisites and additional configuration.

---

[97] UNDERSTAND AZURE IOT EDGE MODULES: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-modules

[98] TUTORIAL: DEVELOP IOT EDGE MODULES FOR LINUX DEVICES: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-develop-for-linux

[99] TUTORIAL: DEVELOP A C IOT EDGE MODULE FOR LINUX DEVICES: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-c-module

Once completed, you will now need to setup your development machine to connect to your Azure IoT environment and your newly created Azure IoT Edge device. This is the purpose of the next section.

Setting up your development machine

This section describes how to configure your development environment. As such, and as already outlined, when developing IoT Edge modules, it's important to understand the different concepts between the development machine and the target IoT Edge device where the module will eventually be deployed.

Microsoft recommend that you don't run IoT Edge on your development machine, but instead use a separate device. This distinction between development machine and IoT Edge device more accurately mirrors a true deployment scenario and helps to keep the different concepts straight.

However, for the sake of simplicity, we will here use the same Ubuntu VM in Azure: IoT Edge modules are packaged as containers, so you need a container engine on your development machine to build and manage the Linux containers of the edge device. A container engine, i.e. Docker CE, is already up and running on this Ubuntu VM.

You will use on your local Windows 10 machine your Visual Studio Code IDE to connect to this Ubuntu VM and do TEE-based applications cross-development with it.

In the previous illustration of this module (see section § *Building a TEE-based Linux application on a simulated ARM TrustZone environment*), you already installed Visual Studio Code along with the Visual Studio Code Remote Development Extension Pack that allows you to open any folder in a container on a remote machine like your Ubuntu VM.

Installing the Azure IoT Tools extension pack

In addition, you will now need to install the Azure IoT Tools extension pack[100] to help developing IoT Edge modules.

These extensions make it easy to discover and interact with Azure IoT Hub that power your Azure IoT Edge device(s), provide project templates, automate the creation of the deployment manifest, and allow you to monitor and manage your Azure IoT Edge device(s):

- The Azure IoT Hub Toolkit[101] extension allows you to interact with an Azure IoT Hub, manage connected Azure IoT Edge devices, and enable distributed tracing for your Azure IoT applications.
- The Azure IoT Edge for Visual Studio Code[102] allows you to easily code, build, and debug your custom logic and deploy it to your Azure IoT Edge devices.

You will install The Azure IoT Tools extension pack, then set up your Azure account to manage IoT Hub resources from within Visual Studio Code.

Note For more information, see article USE VISUAL STUDIO CODE TO DEVELOP AND DEBUG MODULES FOR AZURE IOT EDGE[103].

[100] Azure IoT Tools extension pack: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools

[101] Azure IoT Hub Toolkit: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-toolkit

[102] Azure IoT Edge for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-edge

[103] USE VISUAL STUDIO CODE TO DEVELOP AND DEBUG MODULES FOR AZURE IOT EDGE: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module

Perform the following steps:

1. Launch Visual Studio Code.
2. select **View** > **Extensions**.
3. Search for **Azure IoT Tools**, which is actually a collection of extensions that help you interact with your IoT Hub and your devices, as well as developing IoT Edge modules.
4. Select **Install**. Each included extension installs individually.

As previously stated, you will use the Ubuntu VM also as your Linux development machine. Since it's should be already up and running if you followed the steps in order, Let's now connect it to your Visual Studio Code IDE.

Connecting to your remote Linux development machine

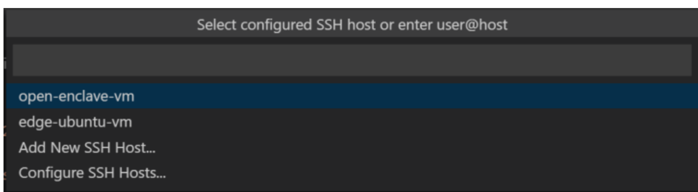
As you already did, you will use the Visual Studio Code Remote Extension Pack to connect your Visual Studio Code IDE to your development machine, i.e. to your Ubuntu VM.

To do so, perform the following steps:

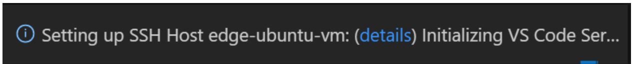
1. From Visual Studio Code, press F1 or CTRL-Shift-P to open the Command Palette, and run `Remote-SSH: Open Configuration File...`
2. Select the SSH config file you wish to update with the Ubuntu VM information, for example *%USERPROFILE%\.ssh\config* used before and add a host entry in the config file for this machine as follows:

```
# Read more about SSH config files: https://linux.die.net/man/5/ssh_config
Host <name-of-your-edge-ubuntu-vm-here>
    User <your-user-name-on-your-edge-ubuntu-vm>
    HostName <edge-ubuntu-vm-ip-goes-here>
    IdentityFile c:\users\<your-user-name-on-Windows10>\.ssh\id_rsa
```

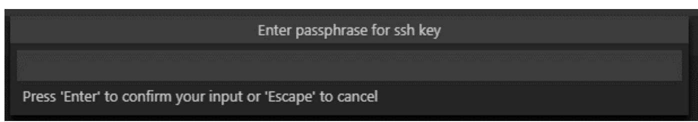
3. Press CTRL+S to save the config file.
4. Open a remote SSH session to your Ubuntu VM as previously done:
    a. Press F1 or CTRL-Shift-P to specify a command, run `Remote-SSH: Connect to SSH Host...`, and select your Ubuntu VM name in the list.



    b. A SSH session to your Ubuntu VM is opening. Visual Studio Code will now continue to configure it.



    c. When prompted, type your passphrase for your SSH key and press ENTER.



Once finished, and like before, you now see a SSH indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!

SSH: edge-ubuntu-vm

Configuring the Azure IoT Hub extension

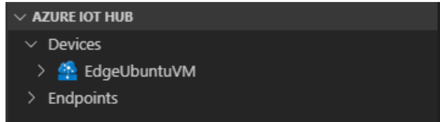
You will install now set up your Azure account to manage IoT Hub resources from within Visual Studio Code.

Note For more information, see article USE VISUAL STUDIO CODE TO DEVELOP AND DEBUG MODULES FOR AZURE IOT EDGE[104].

Perform the following steps:

1. Open the command palette by selecting **View** > **Command Palette**.
2. In the command palette, search for and select **Azure: Sign in**. A browsing session opens. Follow the prompts to sign in to your Azure account.
3. In the command palette again, search for and select **Azure IoT Hub: Select IoT Hub**. Follow the prompts to select your Azure subscription, and then your IoT hub.

4. Open the explorer section of Visual Studio Code by either selecting the icon in the activity bar on the left, or by selecting **View** > **Explorer**.
5. At the bottom of the explorer section, expand the collapsed **Azure IoT Hub Devices** menu. You should see your IoT Edge device associated with the IoT hub that you selected through the command palette.



Let's now configure your (remote) Linux development machine for the Open Enclave SDK.

Configuring your development environment for the Open Enclave SDK

The Open Enclave SDK requires a series of prerequisites to be fulfilled on your remote Linux development machine before creating an Edge container project from Visual Studio Code.

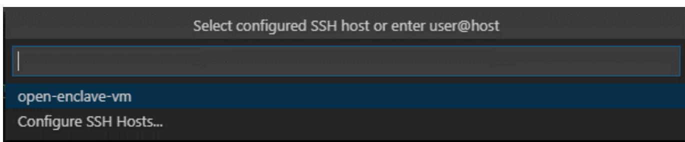
Let's see how to complete the configuration of your remote Linux development machine.

[104] USE VISUAL STUDIO CODE TO DEVELOP AND DEBUG MODULES FOR AZURE IOT EDGE: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module

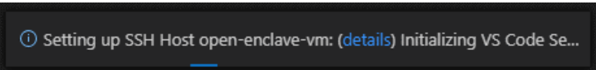
Perform the following steps:

1. From Visual Studio, open a remote SSH session to your Ubuntu VM:

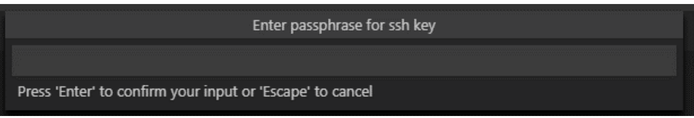
   a. Press F1 or CTRL-Shift-P to specify a command, run `Remote-SSH: Connect to SSH Host...`, and select your DC-series VM name in the list.



   A SSH session to your Ubuntu VM is opening. Visual Studio Code will now continue to configure it.



   b. When prompted, type your passphrase for your SSH key and press ENTER.



   Once finished, you now see a SSH indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!

   SSH: open-enclave-vm

   ET voila! Any Visual Studio Code operations you perform in this window will be executed in the SSH environment, everything from editing and file operations, to debugging, using terminals, and more.

2. Ensure that the requirements[105] are met for the Open Enclave extension for Visual Studio Code:

   a. From Visual Studio Code, click on **Terminal** > **New Terminal** to open a Bash prompt on the Ubuntu VM.

   b. Make sure that the Visual Studio Code Native Debug extension[106] is installed. This should be the case.

   c. Install the required build components:

```
$ sudo apt update && sudo apt install -y build-essential cmake gcc-arm-linux-gnueabihf gcc-aarch64-linux-gnu g++-arm-linux-gnueabihf g++-aarch64-linux-gnu gdb-multiarch python
```

---

[105] Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements

[106] Native Debug extension: https://marketplace.visualstudio.com/items?itemName=webfreak.debug

d. Install CMake 3.12 or above[107], currently 3.15:

i. From the terminal console, remove the installed version if any:

```
$ sudo apt purge cmake
```

ii. Download the binary distribution archive file *cmake-3.15.3.tar.gz*, unpack it, and go to the folder of cmake:

```
$ wget http://www.cmake.org/files/v3.15/cmake-3.15.3-Linux-x86_64.tar.gz
$ tar -xvzf cmake-3.15.3-Linux-x86_64.tar.gz
$ cd cmake-3.15.3-Linux-x86_64/
```

iii. Go to the folder of cmake and from there run the following commands:

```
$ sudo cp -r bin /usr/
$ sudo cp -r share /usr/
$ sudo cp -r doc /usr/share/
$ sudo cp -r man /usr/share/
```

iv. Go outside the folder of cmake and run the following commands:

```
$ cd ..
$ sudo rm -r cmake-3.15.3-Linux-x86_64
$ sudo rm cmake-3.15.3-Linux-x86_64.tar.gz
```

Verify the `cmake` version using the following command:

```
$ cmake --version
```

```
cmake-3.15.3-Linux-x86_64/share/icons/hicolor/32x32/
cmake-3.15.3-Linux-x86_64/share/icons/hicolor/32x32/apps/
cmake-3.15.3-Linux-x86_64/share/icons/hicolor/32x32/apps/CMakeSetup.png
cmake-3.15.3-Linux-x86_64/share/aclocal/
cmake-3.15.3-Linux-x86_64/share/aclocal/cmake.m4
cmake-3.15.3-Linux-x86_64/bin/
cmake-3.15.3-Linux-x86_64/bin/ctest
cmake-3.15.3-Linux-x86_64/bin/cmake
cmake-3.15.3-Linux-x86_64/bin/cmake-gui
cmake-3.15.3-Linux-x86_64/bin/cpack
cmake-3.15.3-Linux-x86_64/bin/ccmake
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ cd cmake-3.15.3-Linux-x86_64/
philber@edge-ubuntu-vm:~/EdgeOpenEnclave/cmake-3.15.3-Linux-x86_64$ sudo cp -r bin /usr/
philber@edge-ubuntu-vm:~/EdgeOpenEnclave/cmake-3.15.3-Linux-x86_64$ sudo cp -r share /usr/
philber@edge-ubuntu-vm:~/EdgeOpenEnclave/cmake-3.15.3-Linux-x86_64$ sudo cp -r doc /usr/share/
philber@edge-ubuntu-vm:~/EdgeOpenEnclave/cmake-3.15.3-Linux-x86_64$ sudo cp -r man /usr/share/
philber@edge-ubuntu-vm:~/EdgeOpenEnclave/cmake-3.15.3-Linux-x86_64$ cd ..
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ sudo rm -r cmake-3.15.3-Linux-x86_64
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ sudo rm cmake-3.15.3-Linux-x86_64.tar.gz
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ cmake --version
cmake version 3.15.3

CMake suite maintained and supported by Kitware (kitware.com/cmake).
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$
```

[107] CMake download: https://cmake.org/download/

3. Ensure that the requirements[108] are met for the Azure IoT Edge extension for Visual Studio Code:

    a. On your Linux development machine, for Edge Container projects, you will need to enable cross-building. The container that you will build to hold your TEE-based module code must match the operating system (OS) of the edge device (and target the underlying platform). (You should keep in mind the difference between the development machine OS and the container OS.)

    To do so, on Ubuntu 16.04, run the followings:

```
$ sudo apt-get install -y qemu qemu qemu-system-misc qemu-user-static qemu-user binfmt-support
$ sudo mkdir -p /lib/binfmt.d
$ sudo sh -c 'echo :qemu-
arm:M::\\x7fELF\\x01\\x01\\x01\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x02\\x00\\x28\\x00:\\xff
\\xff\\xff\\xff\\xff\\xff\\xff\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xfe\\xff\\xff\\xff:/usr/
bin/qemu-arm-static:F > /lib/binfmt.d/qemu-arm-static.conf'
$ sudo sh -c 'echo :qemu-
aarch64:M::\\x7fELF\\x02\\x01\\x01\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x02\\x00\\xb7\\x00:\
\xff\\xff\\xff\\xff\\xff\\xff\\xff\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xfe\\xff\\xff\\xff:/
usr/bin/qemu-aarch64-static:F > /lib/binfmt.d/qemu-aarch64-static.conf'
sudo systemctl restart systemd-binfmt.service
```

    b. Add your Linux user to the docker group. This will allow your user to connect and issue commands to the Docker daemon:

```
$ sudo usermod -aG docker $USER
```

    This command requires to log out and log back in so that your group membership is re-evaluated. With your Ubuntu VM, you will need to restart the VM for changes to take effect. Restart the Ubuntu VM and reopen a Bash terminal from Visual Studio Code.

    c. To validate that your system is configured for cross-building, try testing the docker containers that will be later needed in the build:

```
$ docker run arm32v7/ubuntu:xenial
$ docker run aarch64/ubuntu:xenial
$ docker run amd64/ubuntu:xenial
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                  1: bash
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ docker run arm32v7/ubuntu:xenial
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ docker run aarch64/ubuntu:xenial
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ docker run amd64/ubuntu:xenial
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$
```

    d. Install the Azure IoT EdgeHub Dev Tool[109] (iotedgehubdev).

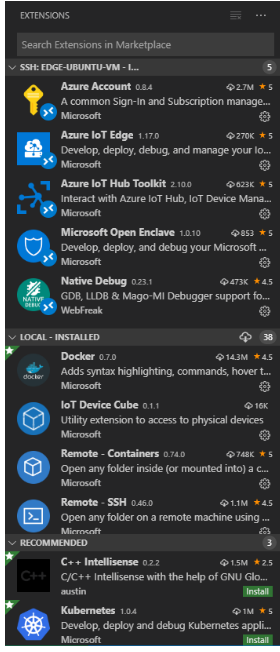
[108] Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements

[109] Iotedgehubdev: https://pypi.org/project/iotedgehubdev/

This tool provides a local development experience with a simulator for creating, developing, testing, running, and debugging Azure IoT Edge modules and solutions.

```
$ sudo apt install pip-python
$ pip install --upgrade iotedgehubdev
```

2. Back in Visual Studio code, on the menu bar, install the Open Enclave extension for Visual Studio Code. When prompted, click on **Install**.



3. Use the `Microsoft Open Enclave: Check System Requirements` command - commands can be found using F1 or CTRL-Shift-P - to validate your system.

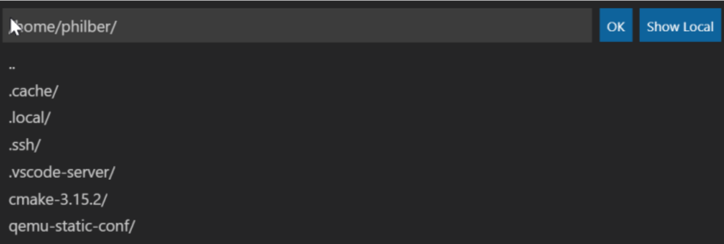
   The command will query whether the required tools and the required versions are present on your system. Any unmet requirements will be presented in a Visual Studio Code warning window.

   Otherwise, you should see instead the message “System meets the requirements”.

Creating an Edge container C/C++ project

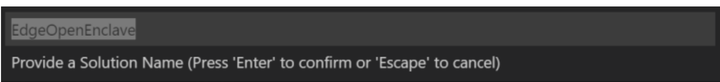
Perform the following steps:

1. Launch Visual Studio Code.
2. As already stated, the Visual Studio Code Remote - SSH extension let you use a remote Linux-based Azure IoT Edge environment as your full-time development environment right from Visual Studio Code.
3. Use the `Microsoft Open Enclave: New Open Enclave Solution` command - commands can be found using F1 or CTRL-Shift-P - to create your first new Azure IoT Edge Module project.

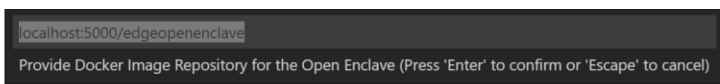


4. Specify the folder in which you want to create the application.

5. When invited to choose the option, create a **Edge container** project.



6. Provide a name for the host application/enclave, for example the default "EdgeOpenEnclave" name in our illustration, and press ENTER to confirm.



7. Provide your image repository. An image repository includes the name of your container registry and the name of your container image.

   Your container image is prepopulated from the name you provided in the last step. Replace **localhost:5000** with the login server value from your Azure container registry, for example, iotedge369.azure.io in our illustration, see section § *Creating an Azure Container Registry* in Appendix. Prerequisites and additional configuration. You can retrieve the login server from the **Overview** page of your container registry in the Azure portal.

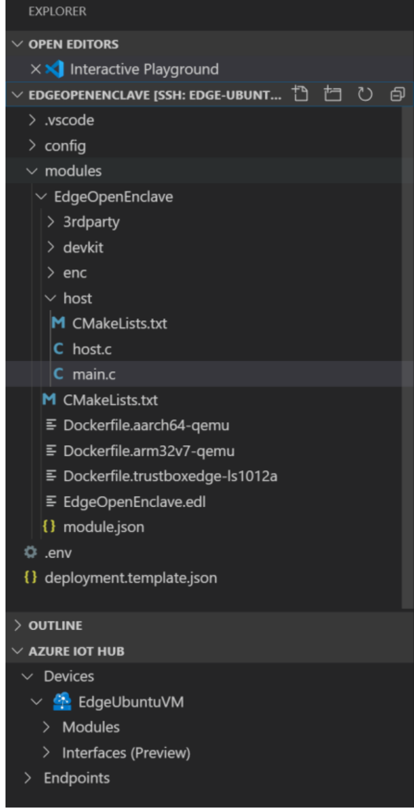
   The final image repository looks like <*registry name*>.azurecr.io/edgeopenenclave. For example, in our illustration: iotedge369.azurecr.io/edgeopenenclave.

Microsoft Open Enclave: Downloading SDK (this is infrequent). Cloning...

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                Microsoft Open Enclave
msiot-vscode-openenclave.checkSystemRequirements:
msiot-vscode-openenclave.checkSystemRequirements:
msiot-vscode-openenclave.checkSystemRequirements:
msiot-vscode-openenclave.newSolution:
Failed to create new Open Enclave solution
Cancelled by user
msiot-vscode-openenclave.newSolution:
Creating base files
Base files created
Creating edge files
Updating deployment template
Downloading SDK (this is infrequent)
Downloading SDK (this is infrequent). Cleaning up SDK folder if needed
Downloading SDK (this is infrequent). Cloning SDK from git
Executing: git clone --recursive --branch feature.new_platforms https://github.com/microsoft/openenclave.git "/home/philber/.vscode-server/data/User/globalStorage/
ms-iot.msiot-vscode-openenclave/1.0.10/3rdparty/openenclave"
Cloning into '/home/philber/.vscode-server/data/User/globalStorage/ms-iot.msiot-vscode-openenclave/1.0.10/3rdparty/openenclave'...
Submodule '3rdparty/OpteeCalls' (https://github.com/ms-iot/OpteeCalls.git) registered for path '3rdparty/OpteeCalls'
Submodule '3rdparty/RIoT' (https://github.com/Microsoft/RIoT.git) registered for path '3rdparty/RIoT'
Submodule '3rdparty/googletest' (https://github.com/google/googletest.git) registered for path '3rdparty/googletest'
Submodule '3rdparty/optee_client' (https://github.com/ms-iot/optee_client.git) registered for path '3rdparty/optee_client'
Submodule '3rdparty/optee_os' (https://github.com/ms-iot/optee_os.git) registered for path '3rdparty/optee_os'
```

Microsoft Open Enclave: Expanding shared devkit

A new solution will be created in the folder you've selected.



The solution will contain all the files and code that you need to deploy a working module to test on your Azure IoT Edge device, or give you a starting point to customize it with your own business logic.

As such, that solution will contain both the host and enclave as well as the required EDL file. The solution appears to somehow like the one in the previous activity, where the code that pertains to TEEs is mostly the same. However, the host will include some code that implements the required Azure IoT Hub communication.

Once your new solution loads in the Visual Studio Code window, take a moment to familiarize yourself with the files that it created:

- The *.vscode* folder contains a file called *launch.json*, which is used for debugging modules.
- The *modules* folder contains a folder for each module in your solution. Right now, that should only be *EdgeOpenEnclave*, or whatever name you gave to the module in the above step #6. The *EdgeOpenEnclave* folder contains both the host and enclave program code as well as the required EDL file.

  The EDL file is as follows:

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License.

enclave {
    from "openenclave/stdio.edl" import *;

    trusted {
        /* define ECALLs here. */
```

```
        public int ecall_handle_message([in, string] char *input_msg, [out, count=enclave_msg_size]
char *enclave_ms, unsigned int enclave_msg_size);
    };

    untrusted {
        /* define OCALLs here. */
        int ocall_log([in, string] char *msg);
    };
};
```

The program code appears to somehow like the one in the previous activity. However, the host will include some code that implements the required Azure IoT Hub communication. As notice above, it also contains the module metadata, and several Docker files.

- The *.env* file holds the credentials to your container registry. These credentials are shared with your Azure IoT Edge device, i.e. your Ubuntu VM, so that it has access to pull the container images.
- The *deployment.debug.template.json* file and *deployment.template.json* file are templates that help you create a deployment manifest (see section § *Pushing the Azure IoT Edge module* below).

  A deployment manifest is a file that defines exactly which modules you want deployed on an edge device, how they should be configured, and how they can communicate with each other and the cloud. The template files use pointers for some values. When you transform the template into a true deployment manifest, the pointers are replaced with values taken from other solution files. Locate the two common placeholders in your deployment template:

  - In the `registryCredentials` section, the address is auto filled from the information you provided when you created the solution. However, the username and password reference the variables stored in the *.env* file. This is for security, as the *.env* file is git ignored, but the deployment template is not.
  - In the `EdgeOpenEnclave` section, the container image isn't filled in even though you provided the image repository when you created the solution. This placeholder points to the *module.json* file inside the *EdgeOpenEnclave* folder.

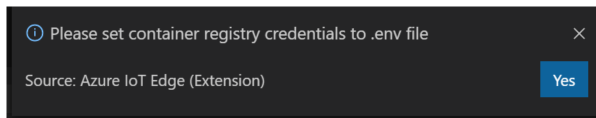
    If you go to that file, you'll see that the image field does contain the repository, but also a tag value that is made up of the version and the platform(s) of the container.

    You can iterate the version manually as part of your development cycle, and you select the container platform using a switcher that we introduce later in this section.

Providing your registry credentials to the IoT Edge agent

The environment file stores the credentials for your container registry and shares them with the IoT Edge runtime. The runtime needs these credentials to pull your container images onto the IoT Edge device.

On Visual Studio Code, the IoT Edge extension tries to pull your container registry credentials from Azure and populate them in the environment file. You may see the following popup during the creation of the above solution.



Check to see if your credentials are already included. If not, add them now.

Perform the following steps:

- Open the *.env* file in your module solution.

```
iotedge369_USERNAME=
iotedge369_PASSWORD=
```

- Add the `username` and `password` values that you copied from your Azure container registry, i.e. the values `registry-username` and `registry-password`, see section § *Creating an Azure Container Registry* in Appendix. Prerequisites and additional configuration.
- Save your changes to the *.env* file.

Reviewing the generated host program code for the Azure IoT Edge module

As aforementioned, the host include some code that implements the required Azure IoT Hub communication.

This code simply receives messages and then passes them on. The pipeline functionality demonstrates here an important concept in IoT Edge, which is how modules communicate with each other.

Each module can have multiple *input* and *output* queues declared in their code. The IoT Edge hub running on the device routes messages from the output of one module into the input of one or more modules. The specific language for declaring inputs and outputs varies between languages, but the concept is the same across all modules.

Note For more information about routing between modules, see article DECLARE ROUTES[110].

The C code that comes with the project template uses the IoTHubModuleClient_LL module[111] from the Azure IoT C SDK:

- Open the *main.c* file, which is inside the folder *modules/EdgeOpenEnclave/host*.
- In the *main.c* file, find the `IoTHubModuleClient_LL_SetInputMessageCallback` function. The `IoTHubModuleClient_LL_SetInputMessageCallback`[112] function sets up an input queue callback to receive incoming messages. Review this function call and see how it initializes an input queue callback called `InputQueue1Callback`.

```
static int SetupCallbacksForModule(IOTHUB_MODULE_CLIENT_LL_HANDLE iotHubModuleClientHandle)
{
    int ret;


    if (IoTHubModuleClient_LL_SetInputMessageCallback(iotHubModuleClientHandle, "input1",
           InputQueue1Callback, (void*)iotHubModuleClientHandle) != IOTHUB_CLIENT_OK)


{
        printf("ERROR:
```

[110] DECLARE ROUTES: https://docs.microsoft.com/en-us/azure/iot-edge/module-composition#declare-routes

[111] IOTHUB_MODULE_CLIENT_LL.H: https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h

[112] IOTHUBMODULECLIENT_LL_SETINPUTMESSAGECALLBACK(): https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-setinputmessagecallback

```
            IoTHubModuleClient_LL_SetInputMessageCallback(\"input1\")..........FAILED!\r\n");
        ret = -1;
    }
    else
    {
        ret = 0;
    }

    return ret;
}
```

- Next, find the IoTHubModuleClient_LL_SendEventToOutputAsync function. The [IoTHubModuleClient_LL_SendEventToOutputAsync][113] function processes received messages and sets up an output queue to pass them along. Review this call and see that it initializes an output queue called output1.

```
static IOTHUBMESSAGE_DISPOSITION_RESULT SendEnclaveResponse(IOTHUB_MODULE_CLIENT_LL_HANDLE
iotHubModuleClientHandle, char* messageBodyStr)
{
    IOTHUBMESSAGE_DISPOSITION_RESULT result;
    IOTHUB_CLIENT_RESULT clientResult;

    char* enclaveMessage = (char*)malloc(512 * sizeof(char));
    int enclaveResult = call_enclave(messageBodyStr, enclaveMessage, 512);
    if (enclaveResult != 0)
    {
        result = IOTHUBMESSAGE_ABANDONED;
    }
    else
    {
        // This message should be sent to next stop in the pipeline, namely "output1".  What happens
        // at "outpu1" is determined by the configuration of the Edge routing table setup.
        MESSAGE_INSTANCE *messageInstance = CreateMessageInstance(enclaveMessage);
        if (NULL == messageInstance)
        {
            result = IOTHUBMESSAGE_ABANDONED;
        }
        else
        {
            printf("Sending message (%zu) to the next stage in pipeline\n",
                messagesReceivedByInput1Queue);


         clientResult = IoTHubModuleClient_LL_SendEventToOutputAsync(iotHubModuleClientHandle,
                messageInstance->messageHandle, "output1", SendConfirmationCallback,
                (void *)messageInstance);


            if (clientResult != IOTHUB_CLIENT_OK)
            {
                IoTHubMessage_Destroy(messageInstance->messageHandle);
                free(messageInstance);
                printf("IoTHubModuleClient_LL_SendEventToOutputAsync failed on sending msg#=%zu,
                    err=%d\n", messagesReceivedByInput1Queue, clientResult);
                result = IOTHUBMESSAGE_ABANDONED;
```

[113] IoTHubModuleClient_LL_SendEventToOutputAsync(): https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-sendeventtooutputasync

```
            }
            else
            {
                result = IOTHUBMESSAGE_ACCEPTED;
            }
        }
    }

    free(enclaveMessage);
    return result;
}
```

- Now, open the *deployment.template.json* file.
- Find the `modules` property of the `$edgeAgent` desired properties. There should be two modules listed here:
    1. The first is `SimulatedTemperatureSensor`, which is included in all the templates by default to provide simulated temperature data that you can use to test your modules.
    2. The second is the `EdgeOpenEnclave` module that you created as part of this solution.
- At the bottom of the file, find the desired properties for the `$edgeHub` module.

  One of the functions of the IoT Edge hub module is to route messages between all the modules in a deployment. Review the values in the `routes` property:

    1. The first route, i.e. `EdgeOpenEnclaveToIoTHub`, uses a wildcard character (*) to indicate any messages coming from any output queues in the `EdgeOpenEnclave` module. These messages go into `$upstream`, which is a reserved name that indicates IoT Hub.
    2. The second route, i.e. `sensorToEdgeOpenEnclave`, takes messages coming from the `SimulatedTemperatureSensor` module and routes them to the `input1` input queue that you saw initialized in the `EdgeOpenEnclave` code.

```
"$edgeHub": {
   "properties.desired": {
   "schemaVersion": "1.0",
   "routes": {
      "EdgeOpenEnclaveToIoTHub": "FROM /messages/modules/EdgeOpenEnclave/outputs/* INTO $upstream",
      "sensorToEdgeOpenEnclave": "FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO
                                  BrokeredEndpoint(\"/modules/EdgeOpenEnclave/inputs/input1\")"
   },
   "storeAndForwardConfiguration": {
      "timeToLiveSecs": 7200
   }
}
```

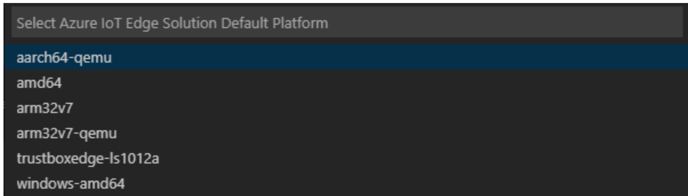
Selecting your target architecture for the Azure IoT Edge module

Currently, Visual Studio Code can develop C# modules for Linux ARMv7-A, AArch64/ARMv8-A, and TrustBoxEdge (LS1012a) devices.

You need to select which architecture you're targeting with each solution, because that affects how the Linux container is built and runs.

Proceed with the following steps:

1. Use `Azure IoT Edge: Set Default Target Platform for Edge Solution` - commands can be found using F1 or CTRL-Shift-P -.



2. In the command palette, select the target architecture from the list of options. Select `aarch64-qemu` for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here `trustboxedge-ls1012a` instead.)

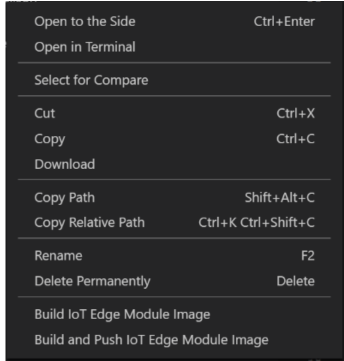
Building the Azure IoT Edge module

You've reviewed the module code for the host, and the deployment template to understand some key module and deployment concepts. Now, you're ready to build the `EdgeOpenEnclave` container image.

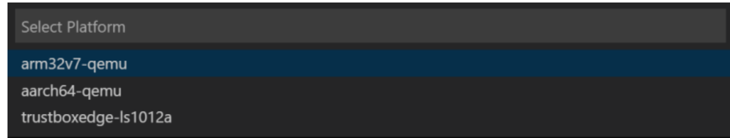
As already noticed, and considering previous requirements, the underlying system used in Visual Studio Code to build is CMake. For Azure IoT Edge projects, Ubuntu containers are used to configure and build. The build task will invoke docker and leverage project dockerfiles.

To build the Azure IoT Edge Module, perform the following steps:

1. Right-click on *modules/EdgeOpenEnclave/module.json*.



2. Select **Build IoT Edge Module Image**.



3. You should see tasks configured to build for each target: **ARMv7-A**, **AArch64/ARMv8-A**, and **TrustBoxEdge(LS1012a)**.

4. Select `aarch64-quemu` for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here `trustboxedge-ls1012a` instead.)

   This "build" command starts two operations:

   a. First, it creates a new folder in the solution called *config* that holds the full deployment manifest, built out of information in the deployment template and other solution files.

   b. Second, it runs `docker build` to build the container image based on the appropriate dockerfile for your target architecture: i.e. *Dockerfile.aarch64-qemu* here.

   As far as the latter are concerned, this will result in executing the following command:

```
$ docker build  --rm -f "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave/Dockerfile.aarch64-qemu" -t iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave"
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                    2: Azure IoT Edge

philber@edge-ubuntu-vm:~/EdgeOpenEnclave$ docker build  --rm -f "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnc
lave/Dockerfile.aarch64-qemu" -t iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu "/home/philber/EdgeOpenE
nclave/modules/EdgeOpenEnclave"
Sending build context to Docker daemon  797.6MB
Step 1/20 : FROM amd64/ubuntu:xenial AS amd64-build-base
 ---> 5e13f8dd4c1a
Step 2/20 : RUN apt-get update &&     apt-get install -y --no-install-recommends software-properties-common gcc gc
c-aarch64-linux-gnu g++ g++-aarch64-linux-gnu make wget python-dev python-pip libcurl3 libssl1.0.0 &&     rm -rf /
var/lib/apt/lists/*
```

This process may take several minutes the first time but is faster the next time that you run the commands.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                    2: Azure IoT Edge

Step 16/20 : WORKDIR /app
 ---> Using cache
 ---> 1dfe10d1ffd2
Step 17/20 : COPY --from=amd64-module /app/host/EdgeOpenEnclave ./host/EdgeOpenEnclave
 ---> Using cache
 ---> 8d8e6eca4fd5
Step 18/20 : COPY --from=amd64-module /app/out/bin ./out/bin
 ---> Using cache
 ---> 4daf52b446aa
Step 19/20 : COPY --from=amd64-module /app/out/lib ./out/lib
 ---> a0d527dec286
Step 20/20 : CMD echo "Starting EdgeOpenEnclave" &&     cp -f ./out/bin/3567a53b-d159-42a8-af7e-3a8a46e4f063.ta /o
ptee_armtz/3567a53b-d159-42a8-af7e-3a8a46e4f063.ta &&     exec ./host/EdgeOpenEnclave
 ---> Running in 759eaeec70f1
Removing intermediate container 759eaeec70f1
 ---> 825ef7d30d2f
Successfully built 825ef7d30d2f
Successfully tagged iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$
```

Pushing the Azure IoT Edge module to your container registry

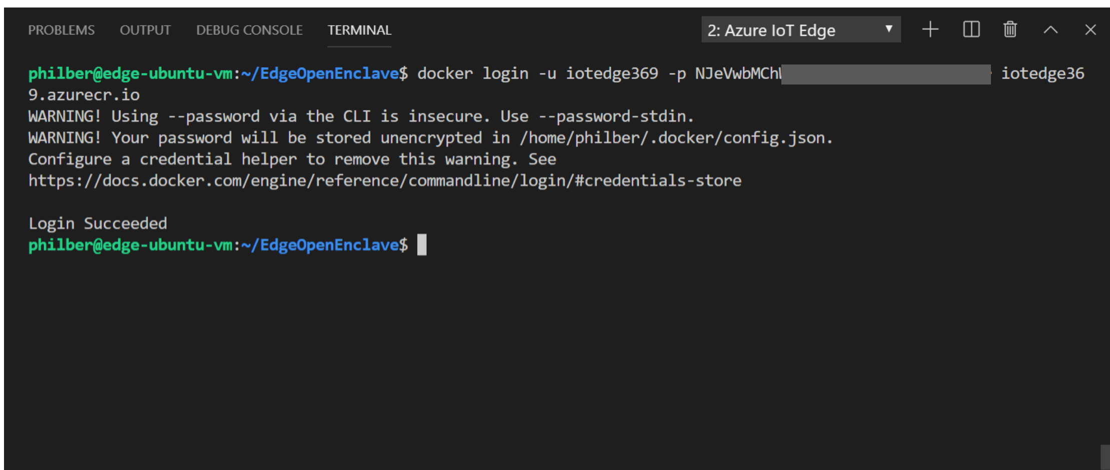
At this stage, you're ready to build the `EdgeOpenEnclave` container image and push it to your container registry. With the IoT tools extension for Visual Studio Code, this step also generates the deployment manifest based on the information in the template file and the module information from the solution files.

Deploying your Edge Container project's enclave is easy as you will see now.

Perform the following steps:

1. Provide your container registry credentials to Docker so that it can push your container image to be stored in your container registry:
    a. Open the Visual Studio Code integrated terminal by selecting **View** > **Terminal**.
    b. Sign into Docker with the Azure container registry credentials that you saved after creating the registry, see section § *Creating an Azure Container Registry* in Appendix. Prerequisites and additional configuration.

```
$ docker login -u <registry-username> -p <registry-password> <registry-login-server>
```



Note You may receive a security warning recommending the use of `--password-stdin` option. The `--password-stdin` option prevents your password from appearing in the command line history. While that best practice is recommended for production scenarios, it's outside the scope of this walkthrough. For more information, see article DOCKER LOGIN[114].

Once you've logged into docker, you can log out (and remove your credentials from the system) by:

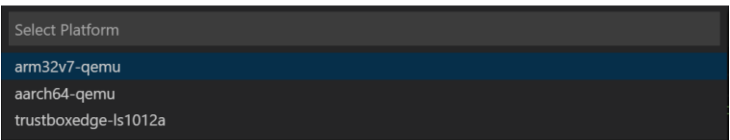
```
$ docker logout <container-url>
```

Visual Studio Code now has access to your container registry, so it's time to turn the solution code into a container image.

[114] DOCKER LOGIN: https://docs.docker.com/engine/reference/commandline/login/#provide-a-password-using-stdin

2. Right-click on *modules/EdgeOpenEnclave/module.json*.

3. Select **Build and Push IoT Edge Module Image**.



4. You should see tasks configured to build for each target: **ARMv7-A**, **AArch64/ARMv8-A**, and **TrustBoxEdge(LS1012a)**.
5. Select `aarch64-quemu` for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here `trustboxedge-ls1012a` instead.)

   This “build and push” command starts three operations:

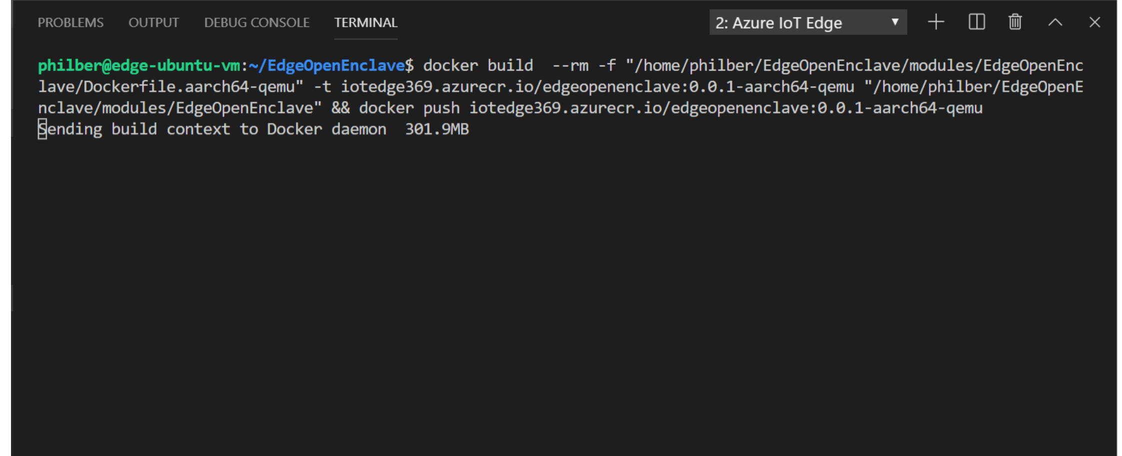
   a. First, it creates a new folder in the solution called *config* that holds the full deployment manifest, built out of information in the deployment template and other solution files.

   b. Second, it runs `docker build` to build the container image based on the appropriate dockerfile for your target architecture: i.e. *Dockerfile.aarch64-qemu* here.

   c. Then, it runs `docker push` to push the image repository to your container registry, i.e. iotedge369.azure.io.

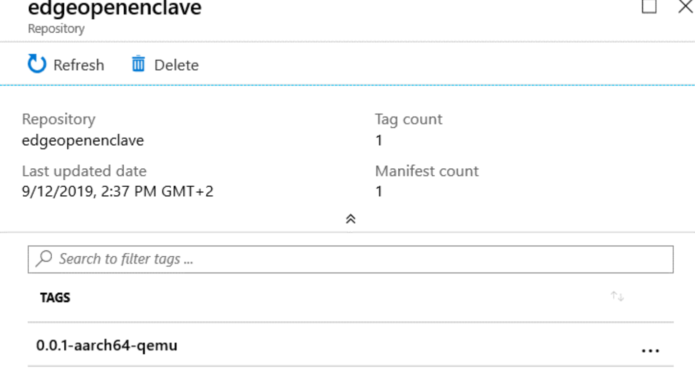
   As far as the latter are concerned, this will result in executing the following command:

```
$ docker build  --rm -f "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave/Dockerfile.aarch64-
qemu" -t iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu
"/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave" && docker push
iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu
```



This process may take several minutes the first time but is faster the next time that you run the commands.

6. At this stage, the container image has been pushed to your container registry in Azure.
   a. To further verify what the above build and push command did, go to the Azure portal at https://portal.azure.com and navigate to your container registry.
   b. In your container registry, select **Repositories** then **edgeopenenclave**.



Likewise, Azure IoT Edge deployment template files have been provided.

Perform the following steps:

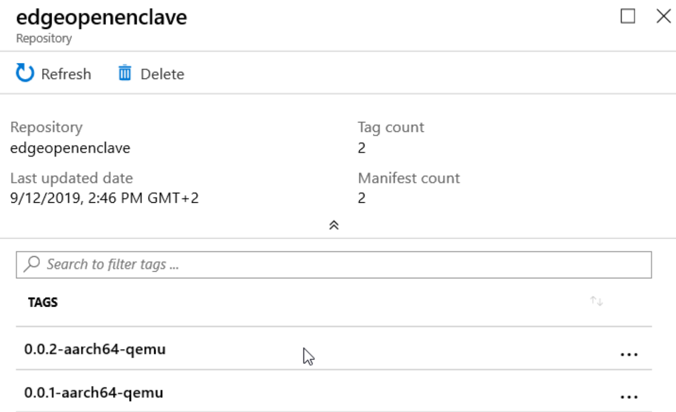
1. Open the *deployment.aarch64-quemu.json* file in newly created *config* folder. The filename reflects the target architecture, so it will be different if you chose a different architecture, for example for a real Scalys TrustBox Edge device.
2. Notice that the two parameters that had placeholders now are filled in with their proper values:
    a. The `registryCredentials` section has your registry username and password pulled from the .env file.
    b. The `EdgeOpenEnclave` has the full image repository with the name, version, and architecture tag from the module.json file.
3. Open the *module.json* file in the *EdgeOpenEnclave* folder.
4. Change the version number for the module image. (The version, not the `$schema-version`.) For example, increment the patch version number to **0.0.2** as though we had made a small fix in the module code.

Note Module versions enable version control and allow you to test changes on a small set of Azure IoT Edge devices before deploying updates to production. If you don't increment the module version before building and pushing, then you overwrite the repository in your container registry.

5. Save your changes to the *module.json* file.
6. Right-click the *deployment.template.json* file again, and select **Build and Push IoT Edge Solution**.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL                    2: Azure IoT Edge
ptee_armtz/3567a53b-d159-42a8-af7e-3a8a46e4f063.ta &&     exec ./host/EdgeOpenEnclave
 ---> Running in 68ba3067875b
Removing intermediate container 68ba3067875b
 ---> 11a3a6359db9
Successfully built 11a3a6359db9
Successfully tagged iotedge369.azurecr.io/edgeopenenclave:0.0.2-aarch64-qemu
The push refers to repository [iotedge369.azurecr.io/edgeopenenclave]
a1f55ec78a6b: Pushed
9e45facd2cd4: Layer already exists
59615a53cf23: Layer already exists
ef163c4beb41: Layer already exists
1a6fbc8db2d6: Layer already exists
2838fcdb5fb3: Layer already exists
2e5ed01d3447: Layer already exists
7078c7b35daf: Layer already exists
9781777d8e79: Layer already exists
777a4eeaaf3d: Layer already exists
0.0.2-aarch64-qemu: digest: sha256:ab316f763672c0225487ee089656279972dd1a9490816141d4d68f477f5aa7bf size: 2406
philber@edge-ubuntu-vm:~/EdgeOpenEnclave$
```

7. Open the *deployment.aarch64-quemu.json* file again. Notice that a new file wasn't created when you ran the build and push command again. Rather, the same file was updated to reflect the changes. The EdgeOpenEnclave image now points to the 0.0.2 version of the container.
8. To further verify what the build and push command did, in your container registry in Azure, select again **Repositories** then **edgeopenenclave**. Verify that both versions of the image were pushed to the registry.



If you encounter errors when building and pushing your module image, it often has to do with Docker configuration on your development machine. Use the following check list to review your configuration:

- Did you run the docker login command using the credentials that you copied from your container registry? These credentials are different than the ones that you use to sign in to Azure.
- Is your container registry correct? Does it have your correct container registry name and your correct module name? Open the *module.json* file in the *modules/EdgeOpenEnclave/*folder to check. The registry value should look like <your registry name>.azurecr.io/edgeopenenclave.
- If you used a different name than default **EdgeOpenEnclave** for your module, is that name consistent throughout the solution?
- You verified that the built container images are stored in your container registry, so it's time to deploy them to a (virtual) Azure IoT Edge device.

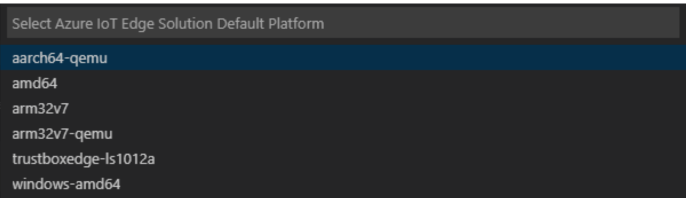
If everything ran well, you verified that the built container images are stored in your container registry, so it's time to deploy them to a (virtual) Azure IoT Edge device.

Deploying the Azure IoT Edge module to your actual Azure IoT Edge device

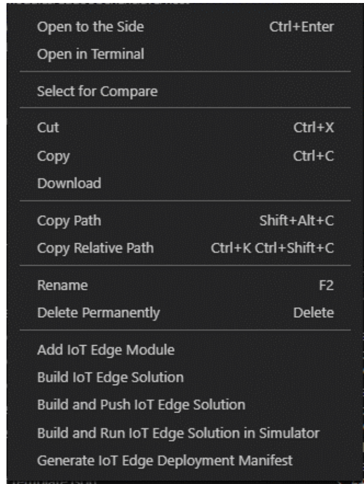
Make sure that your IoT Edge device is up and running.

To create a new deployment configuration based on the current settings in *module.json*, perform the following steps:

1. Use `Azure IoT Edge: Set Default Target Platform for Edge Solution` - commands can be found using F1 or CTRL-Shift-P - to create your first new Azure IoT Edge Module project.



2. Select `aarch64-qemu` for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here `trustboxedge-ls1012a` instead.)
3. Right-click on *deployment.template.json* (or *deployment.debug.template.json*).



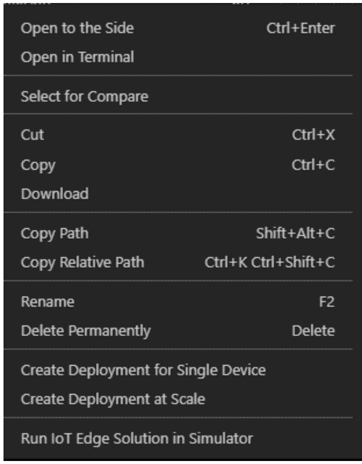
4. Select `Generate IoT Edge Deployment Manifest`. This will generate or replace the appropriate deployment .JSON file in the *config* folder.

Deployment manifest generated at /home/philber/EdgeOpenEnclave/c...

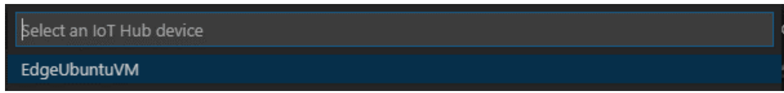
Once your deployment JSON file has been created in the *config* folder, you can deploy to an Azure Edge device.

Perform the following steps:

1. Navigate into to the *config* folder.
2. Right-click on the *deployment.aarch64-quemu.json* file. Do not use the *deployment.template.json* file, which doesn't have the container registry credentials or module image values in it.



3. Select `Create Deployment for Single Device` or `Create Deployment at Scale`.



4. Select your Azure IoT Edge device and press ENTER.

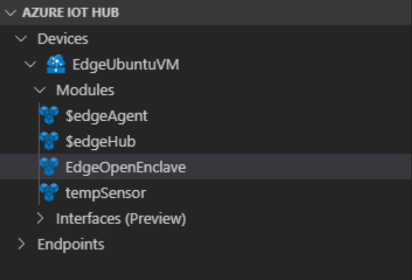
5. Et voila! This is all what it takes to deploy your Azure IoT Edge module onto your actual device.

   It may take a few minutes for both modules to start. The Azure IoT Edge runtime needs to receive its new deployment manifest, pull down the module images from the container runtime, then start each new module.

6. In the Azure IoT Hub extension in Visual Studio Code:

   a. Expand the details for your Azure IoT Edge device, then expand the **Modules** list for your device.

   b. Use the refresh button to update the device view until you see the tempSensor (SimulatedTemperatureSensor) and EdgeOpenEnclave modules running on your device.



7. To further verify what the deployment command did against your Azure IoT Edge device:

   - Go again to the Azure portal at https://portal.azure.com and navigate this time to your Azure IoT Hub.
   - In your Azure IoT Hub, select **IoT Edge**, your device in list, and then **Set Modules**.
   - select again **Repositories** then **edgeopenenclave**. Verify that both versions of the image were pushed to the registry.

Modules	IoT Edge Hub connections	Deployments	
NAME	TYPE	SPECIFIED IN DEPLOYMENT	REPORTED BY DEVICE
$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes
$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes
EdgeOpenEnclave	IoT Edge Custom Module	✓ Yes	✓ Yes
tempSensor	IoT Edge Custom Module	✓ Yes	✓ Yes

This concludes the second illustration of this third module.

This also concludes this starter guide.

Appendix. Prerequisites and additional configuration

Setting up a core Azure IoT environment

To setup a core Azure IoT environment, perform the following four activities:

1. Create an Azure Container Registry.
2. Create an Azure IoT Hub.
3. Register an Azure IoT Edge device to your IoT hub.
4. Install and start the Azure IoT Edge runtime on your edge device.

Each activity is detailed in order in the next sections.

Creating an Azure Container Registry

In section § *Building a TEE-based Linux module on an Edge ARM TrustZone device*, you will build a TEE-based module and create a container image from the related project files. Then you will push this image to a registry that stores and manages your images. Finally, you will deploy your image from your registry to run on your IoT Edge device.

You can use any Docker-compatible registry to hold your container images. Two popular Docker registry services are Azure Container Registry[115] and Docker Hub[116]. You will use here Azure Container Registry (ACR).

Azure Container Registry allows you to manage a Docker private registry in Azure where you can store and manage your private Docker container images.

Perform the following steps:

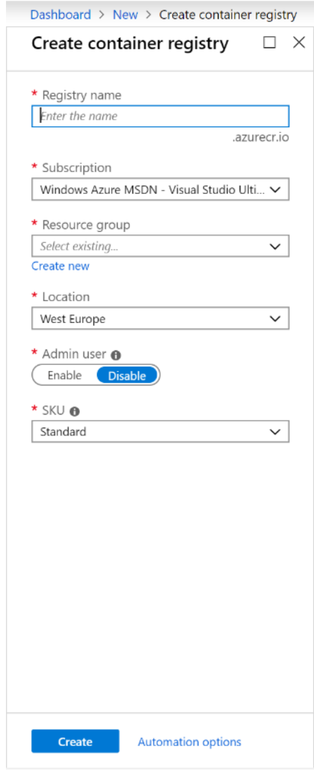
Note	For more information, see article QUICKSTART: CREATE A PRIVATE CONTAINER REGISTRY USING THE AZURE PORTAL[117].

1. Open a browser session and go to the Azure portal at https://portal.azure.com.
2. Sign in with your Azure account.
3. Select **Create a resource** > **Containers** > **Container Registry**.

[115] Azure Container Registry: https://azure.microsoft.com/en-us/services/container-registry/

[116] Docker Hub: https://www.docker.com/products/docker-hub

[117] QUICKSTART: CREATE A PRIVATE CONTAINER REGISTRY USING THE AZURE PORTAL: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal



4. Specify the required settings.

Setting	Description
Registry name	Provide a unique name. The registry name must be unique within Azure and contain 5-50 alphanumeric characters. For example, iotedge369 in our illustration.
Subscription	Select your subscription from the drop-down list if not already selected.
Resource group	You can create a new resource group or use an existing one. To create a new one, click on **Create** new and fill in the name you want to use. For example, in our illustration, RG-IOTEDGE in our illustration. To use instead an existing resource group, click on **Use existing** and select the resource group from the dropdown list.
Location	Choose a location close to you.
Admin user	Set to **Enable**. If enabled, you can use the registry name as username and admin user access key as password to docker login to your container registry, see section § *Providing your registry credentials to the IoT Edge agent*.
SKU	Select **Basic**. The Basic registry, which is a cost-optimized option for developers learning about Azure Container Registry. For details on available service tiers, see article Azure Container Registry SKUs[118].

5. Click on **Create** to deploy the container registry instance.

[118] Azure Container Registry SKUs: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-skus

Resource group (change)
RG-IOTEDGE

Location
North Europe

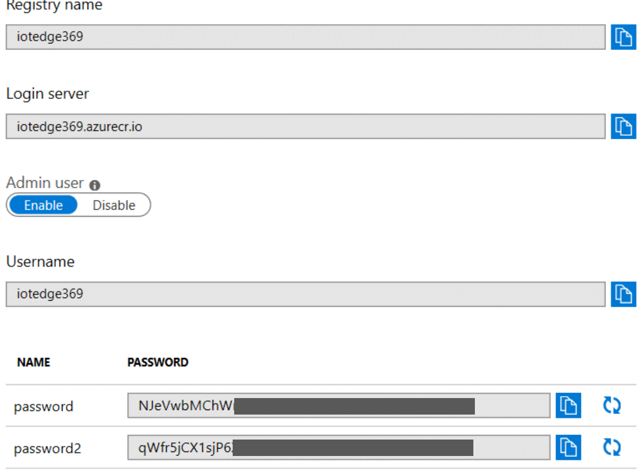
Subscription (change)
Windows Azure MSDN - Visual Studio Ultimate

Login server
iotedge369.azurecr.io

Creation date
9/10/2019, 10:25 AM GMT+2

SKU
Standard

6. After your container registry is created, browse to it, and then select **Access keys** under **Settings**.



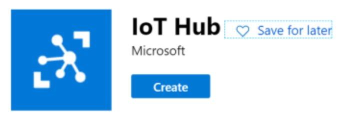
7. Make a note of the values for **Login server** (iotedge369.azurecr.io in our illustration), **Username** (iotedge369 in our illustration) and **Password** and save them somewhere convenient. You use these values throughout this module to provide access to your container registry. These values will be respectively referred as to `registry-login-server`, `registry-username`, and `registry-password`.

Creating an Azure IoT Hub

Perform the following steps:

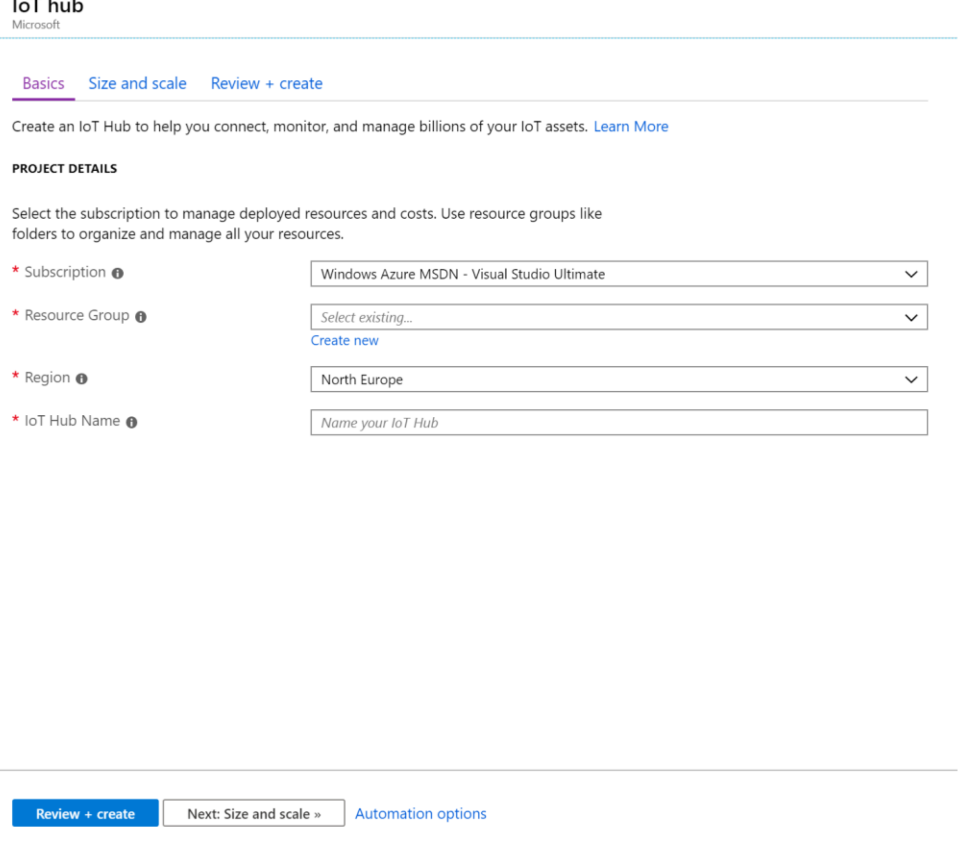
Note For more information, see article CREATE AN IOT HUB USING THE AZURE PORTAL[119].

1. Still from the Azure portal at https://portal.azure.com, in the left pane, select **Create a resource**. Search for "*IoT Hub*" in the **Search the Marketplace** search bar.



2. Select **Create.**

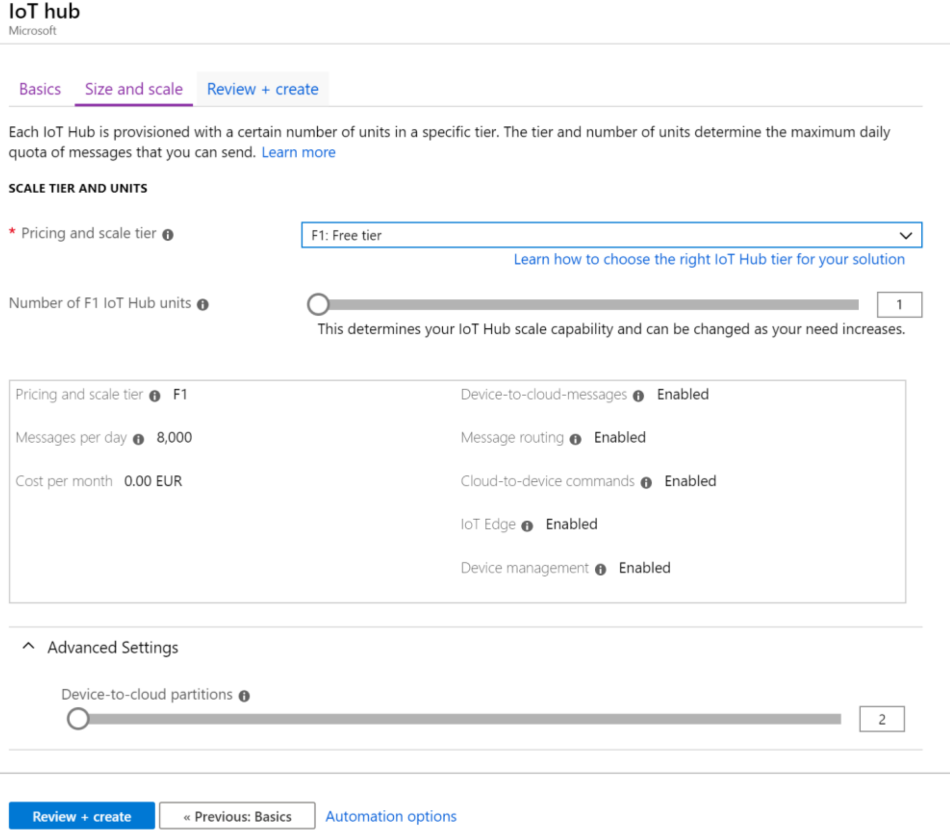
[119] CREATE AN IOT HUB USING THE AZURE PORTAL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-create-through-portal



3. Specify the required settings.

Setting	Description
Subscription	Select your subscription from the drop-down list if not already selected.
Resource Group	You can create a new resource group or use an existing one. To create a new one, click on **Create** new and fill in the name you want to use. To use instead an existing resource group, like the previous resource group, click on **Use existing** and select the resource group from the dropdown list. We recommend that you use the same resource group for all the test resources that you create for your Azure IoT platform as part of this walkthrough. For example, RG-IOTEDGE in our illustration.
Region	This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.
IoT Hub Name	Specify a name for your Azure IoT Hub, for example iothub369 in our illustration. This name must be globally unique. If the name you enter is available, a green check mark appears.

4. Click on **Next: Size and scale** to continue creating your Azure IoT Hub.



5. Specify the required settings.

Setting	Description
Pricing and scale tier	Specify the tier to use. You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free level of Azure IoT Hub works for this guide. If you've used Azure IoT Hub in the past and already have a free hub created, you can use that Azure IoT hub. Keep in mind that each subscription can only have one free IoT hub. Select **F1: Free tier** or **S1: Standard tier** for the pricing tier.
Number of F1|S1 IoT Hub units	Specify the number of units. The number of messages allowed per unit per day depends on your hub's pricing tier. You don't need more than one unit for this guide.
Device-to-cloud partition	Specify the number of partitions. This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most Azure IoT Hubs only need four partitions, you can keep it by default.

6. Select **Review + Create** to review your choices.
7. If validation has passed, click on **Create** to create your new Azure IoT hub. Creating the hub takes a few minutes.

This is it! Your Azure IoT Hub is ready, let's now set up and connect our edge device to our hub.

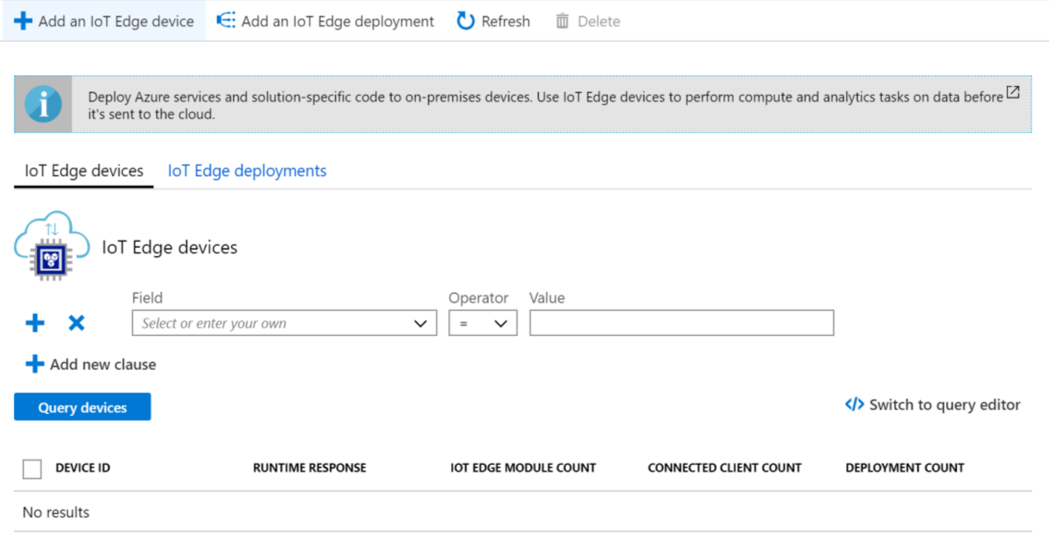
Registering an Azure IoT Edge device to your Azure IoT hub

Let's now create a device identity for your Azure IoT Edge device on your Azure IoT Hub so that it can communicate with your Azure IoT hub. The device identity lives in the cloud, and you use a unique device connection string to associate a physical device to a device identity.

Note For more information, see article REGISTER A NEW AZURE IOT EDGE DEVICE FROM THE AZURE PORTAL[120].

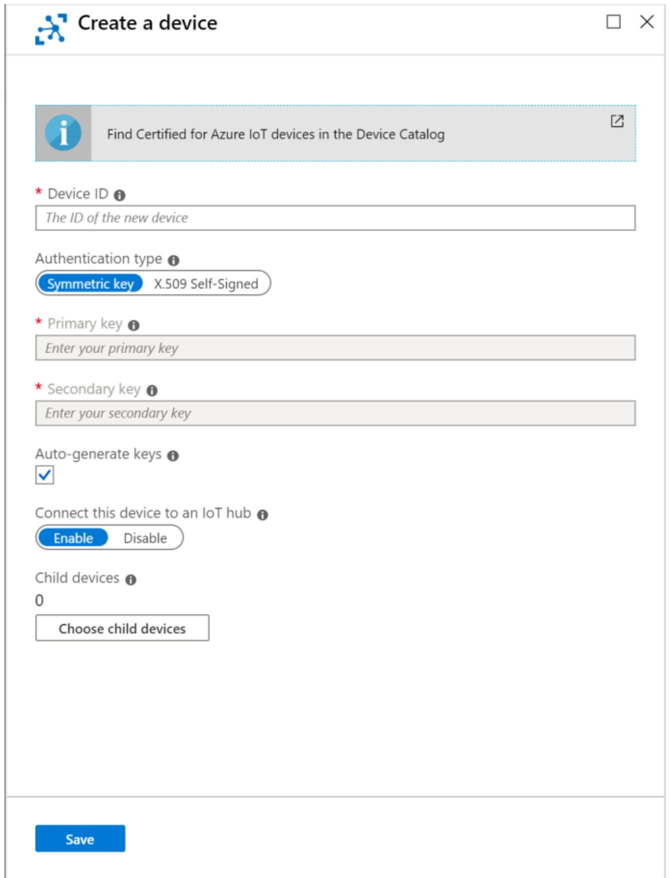
To manually create a device identity, perform the following steps:

1. From Azure portal, search for "*IoT Hub*" in the search bar and select your **IoT Hub**.
2. Click on **IoT Edge** under **Automatic Device Management**.

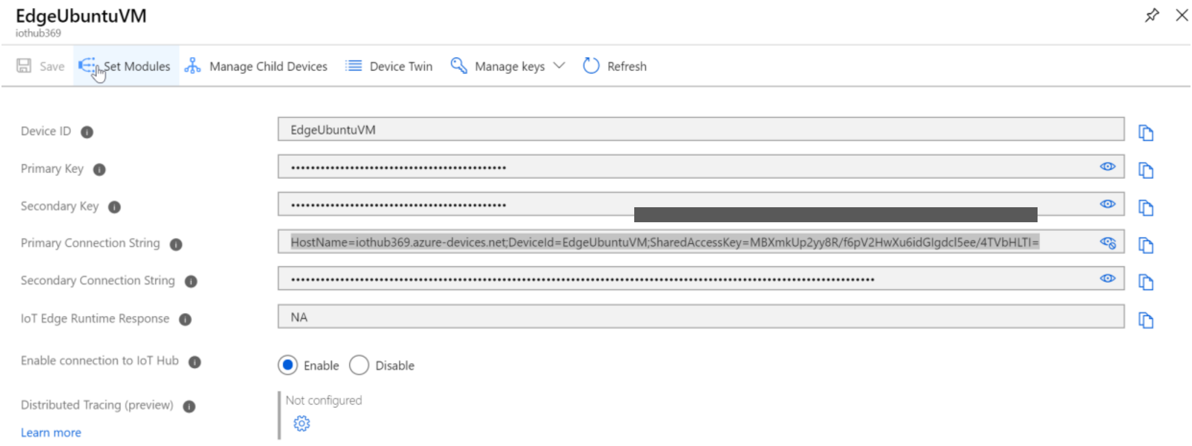


3. Select **Add an IoT Edge Device.**

[120] REGISTER A NEW AZURE IOT EDGE DEVICE FROM THE AZURE PORTAL: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-register-device-portal



4. Specify the required settings. Enter a unique **Device ID**, for example, EdgeUbuntuVM in our illustration, and then click on **Save.**
5. Now click on the newly created device in the list of Azure IoT Edge devices.



You should look out for **Primary Connection String**. This value is the device connection string. You'll use this connection string to connect your actual device, and thus configure for that purpose the Azure IoT Edge runtime in the next section on your Azure IoT Edge device, save it.

It should look like this:

```
HostName=YourIoTHubName.azure-
devices.net;DeviceId=SimulatedDevice;SharedAccessKey={YourSharedAccessKey}
```

Make a note of it. It will be further referred as to the string `your_iothub_edge_connection_string`.

Now we are all set! Let's now configure your actual (virtual) Azure IoT Edge device.

Installing and starting the Azure IoT Edge runtime on your device

The Azure IoT Edge runtime is deployed on all Azure IoT Edge devices. It has three components:

1. The [Azure IoT Edge security daemon][121] starts each time an Azure IoT Edge device boots and bootstraps the device by starting the IoT Edge agent.
2. The Azure IoT Edge agent facilitates deployment and monitoring of modules on the Azure IoT Edge device, including the IoT Edge hub.
3. The Azure IoT Edge hub manages communications between modules on the Azure IoT Edge device, and between the device and your Azure IoT Hub.

During the runtime configuration, you provide a device connection string.

A real edge device can typically be the Scalys TrustBox Edge[122] device, which is an Industrial grade, tamper-resistant secured Azure IoT Edge device optimized for confidential computing using TEEs.

However, as mentioned earlier, for the sake of this guide, and to lessens the barrier for this walkthrough, not requiring some specific hardware, you will use instead an Ubuntu VM in Azure to act as your IoT Edge device, which allows you to quickly create a test machine with all prerequisites installed and then delete it when you're finished with this guide.

For that purpose, you should use the Microsoft-provided Azure IoT Edge on Ubuntu[123] VM, which preinstalls everything you need to run IoT Edge on a device. This virtual machine will install the latest Azure IoT Edge runtime and its dependencies on startup and makes it easy to connect to your IoT Hub.

> **Note** For instructions on how to run the Azure IoT Edge runtime on your own (virtual) device, see article INSTALL THE AZURE IOT EDGE RUNTIME ON DEBIAN-BASED LINUX SYSTEMS[124].

Perform the following steps:

> **Note** For instructions on how to run the Azure IoT Edge runtime on your own (virtual) device, see article INSTALL THE AZURE IOT EDGE RUNTIME ON DEBIAN-BASED LINUX SYSTEMS[125].

[121] Azure IoT Edge security manager: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-security-manager

[122] Scalys TrustBox Edge: https://scalys.com/trustbox-industrial/

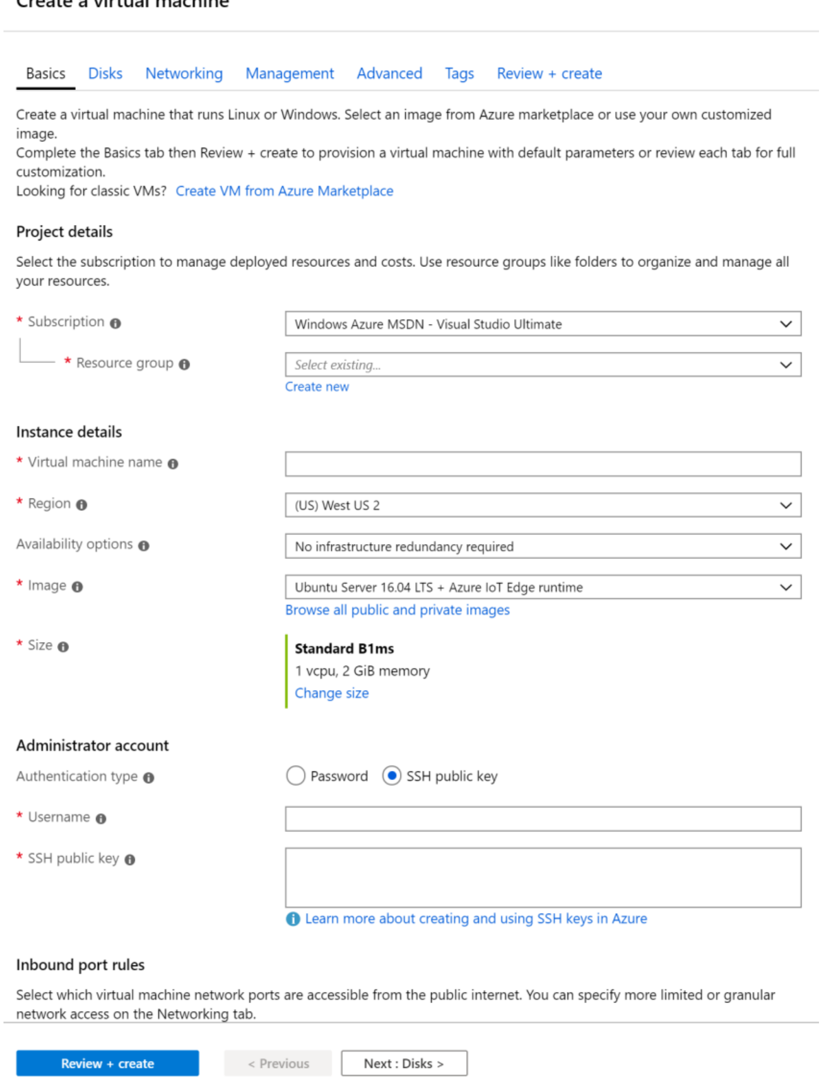
[123] Azure IoT Edge on Ubuntu: https://azuremarketplace.microsoft.com/marketplace/apps/microsoft_iot_edge.iot_edge_vm_ubuntu

[124] INSTALL THE AZURE IOT EDGE RUNTIME ON DEBIAN-BASED LINUX SYSTEMS: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux

[125] INSTALL THE AZURE IOT EDGE RUNTIME ON DEBIAN-BASED LINUX SYSTEMS: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux

1. Still from the Azure portal at https://portal.azure.com, in the left pane, select **Create a resource**. Search for "*Azure IoT Edge on Ubuntu*" in the **Search the Marketplace** search bar.

2. Click on **Create** and follow the wizard to deploy the VM.

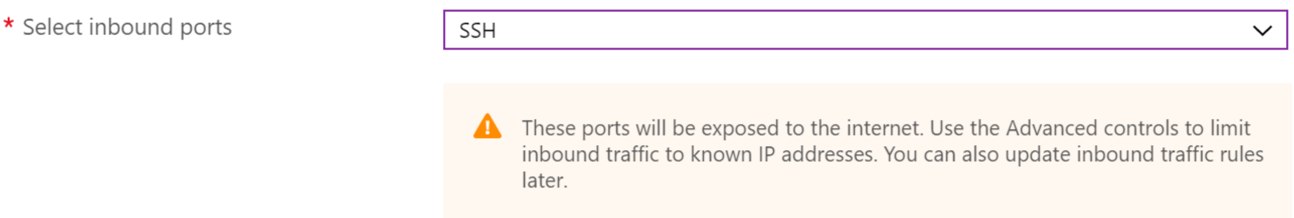


3. Specify the required settings.

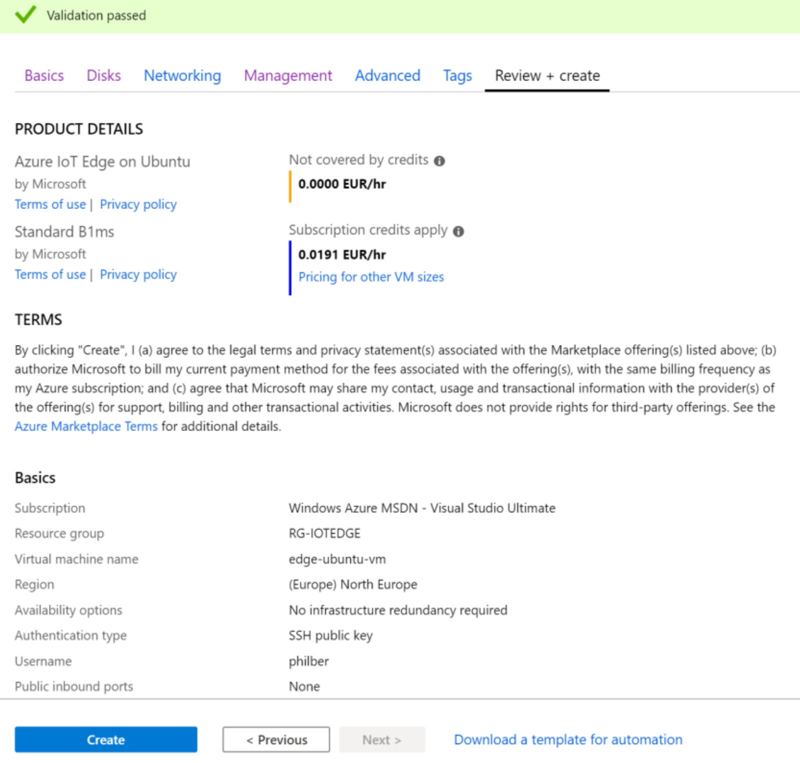
Setting	Description
Subscription	Select your subscription from the drop-down list if not already selected.

Resource group	You can create a new resource group or use an existing one. To create a new one, click on **Create** new and fill in the name you want to use. To use instead an existing resource group, like the previous resource group, click on **Use existing** and select the resource group from the dropdown list. We recommend that you use the same resource group for all the test resources that you create for your Azure IoT platform as part of this walkthrough. For example, RG-IOTEDGE in our illustration.
Virtual machine name	Provide the VM a hostname (as a resource, which will be displayed in Azure). For example, edge-ubuntu-vm in our illustration.
Region	Select the Azure location where you want to deploy the VM.
Availability options	Leave **No infrastructure redundancy required** selected. THIS IS FOR DEMO PURPOSE ONLY
Image	Leave **Ubuntu Server 18.04 LTS + Azure IoT Edge runtime** selected.
Size	Leave **Standard B1ms** selected.
Username	Specify a username for the privileged user account of the VM.
Authentication type	Select **SSH public key** for stronger authentication to later remotely connect to your VM.
SSH public key	Specify a RSA public key in the single-line format beginning with "*ssh-rsa*" - you can use instead the multi-line PEM format -. You can use the same key as the one previously created for your DC-series VM

4. Select **Networking**.
5. In **Select inbound ports**, ensure **SSH** is selected. THIS IS FOR DEMO PURPOSE ONLY.

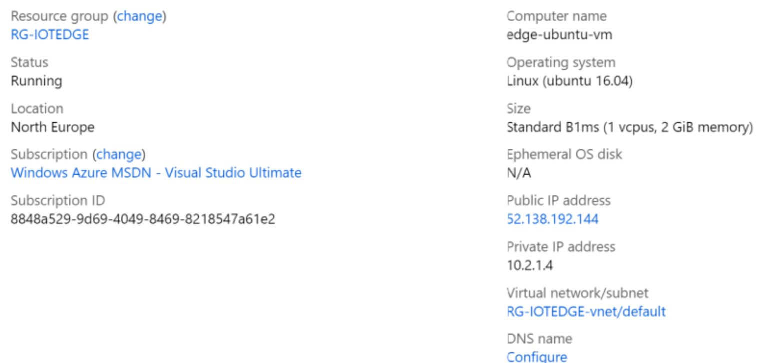


6. Click on **Review + create**.



7. Click on **Create**.

It may take a few minutes to create and start the new VM.



Once your Ubuntu VM is deployed, you need to provision it, i.e. connect it to your Azure IoT Hub. An Azure IoT Edge device can be provisioned manually using a device connection string provided by your Azure IoT Hub or automatically using the Device Provisioning Service (DPS), which is helpful when you have many devices to provision. For the sake of simplicity here, you will provision your device, i.e. VM, manually (since you only deal with a single device here).

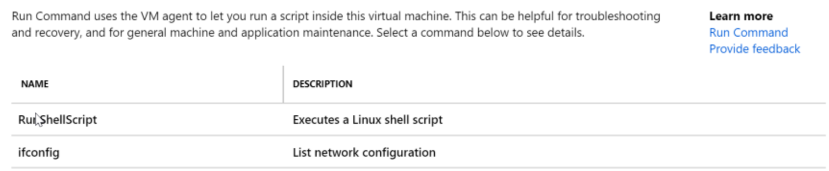
To do so, you now need to configure your Ubuntu VM with the device connection string `your_iothub_edge_connection_string`, you made a copy of in the previous section, which is of the form:

```
HostName=YourIoTHubName.azure-
devices.net;DeviceId=SimulatedDevice;SharedAccessKey={YourSharedAccessKey}
```

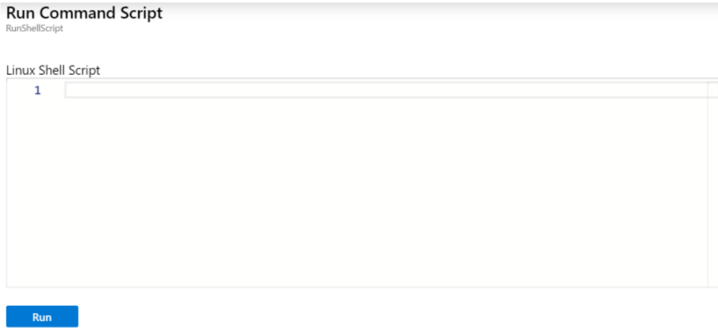
You can do this remotely without having to connect to the Ubuntu VM with the run command **RunShellScript** feature via the Azure portal to execute: `/etc/iotedge/configedge.sh "<your_iothub_edge_connection_string>"`

To do so, perform the following steps:

1. Select your newly created virtual machine resource from the Azure portal and click on **Run command** under **Operations**.



2. Select **RunShellScript**.

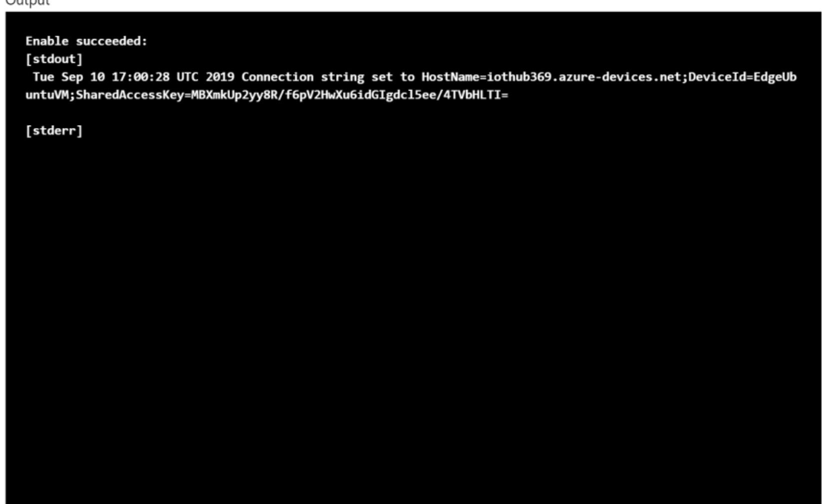


3. Execute the script below via the command window with your device connection string:

```
/etc/iotedge/configedge.sh "<your_iothub_edge_connection_string>"
```

4. Click on **Run**.
5. Wait a few moments, and the screen should then provide a success message indicating the connection string was set successfully.

Output



Copyright © 2019 Microsoft France. All right reserved.

Microsoft France
39 Quai du Président Roosevelt
92130 Issy-Les-Moulineaux

The reproduction in part or in full of this document, and of the associated trademarks and logos, without the written permission of Microsoft France, is forbidden under French and international law applicable to intellectual property.

MICROSOFT EXCLUDES ANY EXPRESS, IMPLICIT OR LEGAL GUARANTEE RELATING TO THE INFORMATION IN THIS DOCUMENT.

Microsoft, Azure, Office 365, Microsoft 365, Dynamics 365 and other names of products and services are, or may be, registered trademarks and/or commercial brands in the United States and/or in other countries.