

Asp.Net Core and Azure with Raspberry Pi 4



.Net Core Applications in
Raspbian OS

—
Sibeesh Venu

Asp.Net Core and Azure with Raspberry Pi 4

**.Net Core Applications
in Raspbian OS**

Sibeesh Venu

Apress®

Asp.Net Core and Azure with Raspberry Pi 4: .Net Core Applications in Raspbian OS

Sibeesh Venu
Birkenfeld, Germany

ISBN-13 (pbk): 978-1-4842-6442-3
<https://doi.org/10.1007/978-1-4842-6443-0>

ISBN-13 (electronic): 978-1-4842-6443-0

Copyright © 2020 by Sibeesh Venu

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 NY Plaza, New York, NY 10014. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6442-3. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To my mother, Thankamani

“When you are looking at your mother, you are looking at the purest love you will ever know.” —Charley Benetto

Thank you for picking up this book and giving me the most important and precious thing you can share: your time.

Table of Contents

- About the Authorix**
- About the Technical Reviewerxi**
- Acknowledgmentsxiii**
- Introductionxv**

- Chapter 1: About Raspberry Pi 1**
 - About Raspberry Pi 1
 - The History of the Raspberry Pi.....2
 - About Raspberry Pi 4.....3
 - Raspberry Pi 4 Accessories5
 - Introduction to the Operating System 10
 - Raspbian..... 10
 - Windows 10 IoT 11
 - About Windows 10 IoT Core..... 12
 - Installing the Raspbian Operating System..... 12
 - Using NOOBS to Install the OS..... 17
 - Summary..... 18

- Chapter 2: Configuring Your Raspberry Pi 19**
 - Enabling SSH 19
 - Enabling Wi-Fi Configuration..... 22
 - Checking Whether the Pi Is Connected to Wi-Fi..... 23

TABLE OF CONTENTS

- Connecting the Raspberry Pi via SSH23
- Summary.....28
- Chapter 3: Setting Up the Prerequisites to Develop the Application29**
 - Developing the Application30
 - Using WSL31
 - WSL vs. WSL2.....31
 - Installing WSL.....32
 - Installing the Linux Distribution34
 - Setting Up the Connection to Raspberry Pi.....36
 - Installing .NET Core on Ubuntu40
 - Summary.....42
- Chapter 4: Creating and Deploying a .NET Core Application to Raspberry Pi.....43**
 - Creating a .NET Core Application43
 - Installing Visual Studio Code Remote WSL Extension48
 - Rewriting the Application.....51
 - Deploying the App to Raspberry Pi52
 - Variables in VSCode56
 - Debugging the App from Raspberry Pi.....58
 - Summary.....60
- Chapter 5: Playing with Azure IoT Hub and Our Application63**
 - Using Azure IoT Hub.....63
 - Creating an Azure IoT Hub64
 - Registering a Device in the IoT Hub74

Connecting Raspberry Pi to Azure IoT Hub	76
Monitoring the Device Data and IoT Hub.....	81
Adding Custom Event Message Properties	85
Summary.....	86
Chapter 6: Finally, A Windows Terminal That You Can Customize	87
Using Windows Terminal	87
Windows Terminal Key Features	88
Configuring Windows Terminal	89
Windows Terminal Preview Version	91
Open Folders in Windows Terminal	91
Font Weight Support.....	91
Support to Open a Profile as a Pane.....	91
Change the Tab Color.....	92
Rename a Tab	93
Summary.....	93
Chapter 7: Cloud to Device Communication	95
Cloud-to-Device Communication Options	96
Direct Methods	96
Twin's Desired Properties	104
Cloud-to-Device Messages.....	120
Demo Application.....	125
Summary.....	127
Chapter 8: IoT Edge	129
IoT Edge	129
IoT Edge Runtime	130
Creating an IoT Edge Device	132
Installing IoT Edge Runtime on Linux Systems	134

TABLE OF CONTENTS

Deploying a Module to IoT Edge Device	142
Viewing Sent Messages	151
Summary.....	152
Chapter 9: Developing IoT Edge Modules	153
Prerequisites	153
Setting Up VSCode	154
Creating an Azure Container Registry	156
Creating a New Project	158
Deploying the Modules to the Device.....	184
Viewing Device Messages	187
Summary.....	190
Chapter 10: Azure IoT Central.....	191
Azure IoT Central.....	191
What Is Azure IoT Central	191
IoT Hub vs. IoT Central.....	192
Creating an IoT Central Application	192
Creating a Device	208
Getting the Device Connection Keys.....	209
Creating a Device Application That Sends Telemetry to IoT Central	211
Test Property and Command	219
Summary.....	228
Summary	229
Index.....	231

About the Author

Sibeesh Venu is a passionate learner who thinks that when you stop learning, you become old, and he doesn't want to feel old. In his career, he has been awarded Most Valuable Professional from Microsoft five times for his technical contributions to the community. His hobbies are reading, writing, listening to music, and creating content on YouTube. He blogs about what he learns at sibeeshpassion.com. He also owns two YouTube channels—youtube.com/sibeeshpassion, where he creates content about technology, and youtube.com/njanorumalayali, where he uploads videos about what he does in his day-to-day life.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, and cloud and IT architecture. His true IT passions are security and Android.

He has been programming and teaching others how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science in computing science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and as a senior lead IT security/cloud/SCADA architect for many years.

Acknowledgments

I want to dedicate this book to two women in my life—my mom and my wife.

To my mom, Thankamani, who sacrificed her entire life for me and struggled to make my life smooth. She worked hard without even noticing that her health was failing, in order to make a good living for us. She showered us with love, and she taught me what life is. She passed her strength on to me so that I could achieve my goals.

And to my wife, Vaidehi, who has always believed in what I do, who supported me in my bad days, and whom I believe as my soul. She taught me how to be calm when I am disturbed. She taught me what unconditional love is. She spent her time transforming me into a better person.

Dev, my son, my joy, my pride, my blessing, my love. He gives me an immense amount of happiness when I look at him.

My father, Venu, who played an amazing role in my life and in making me the person I am today.

My sisters, Sini, and Siji, for having faith in me. A sister is an angel on earth who brings out your best qualities, I am lucky to have you both. My brothers-in-law, Sumesh, Shaiju, and Vaitheesh, for standing with me all the time, no matter what the circumstances were.

My father-in-law, Ramakrishnan, and my mother-in-law, Usha, for giving me continuous support all the time. Thank you for never letting the words “in-law” get in between our relationship. You are the best in-laws in the world.

ACKNOWLEDGMENTS

Ajaybhasy, my friend, without whom I would have not selected this profession. He has been the one true friend with whom I can share anything.

I also want to dedicate this book to all my relatives, friends, and followers. Without them I would not be where I am today.

A debt of gratitude to Jessica Vakilil for her continuous support in editing this book to the state what you have now, to Aaron Black for his initial support and for selecting me to write this book, and to everyone at Apress for their belief in me and in this project.

And last, but certainly not least, you, the reader. Thank you for your support.

Introduction

We live in a world where everything is connected and the future is moving toward IoT (Internet of Things). A Raspberry Pi is a tiny device with which you can do some amazing things. Every device runs on an operating system, and the Raspberry Pi device does too.

Experiencing the Raspberry Pi with the Raspbian Operating System and ASP.NET Core is amazing. What if we added the power of Microsoft Azure to it? That is a tremendous combo.

Here in this book, you will learn:

- What is Raspberry Pi?
- What are the possibilities of Raspberry Pi?
- What operating systems can be used with Raspberry Pi?
- What is the Windows 10 IoT Core?
- How do you set up Raspberry Pi with the Raspbian Operating System?
- How do you run a .Net Core application on Raspberry Pi?
- What is Azure IoT Hub and how do you work with it?
- What is Azure IoT Edge and how do you work with it?
- What is Azure IoT Central and how do you work with it?

Do you find any of these questions interesting? If you do, you are in the right place. Let's learn!

CHAPTER 1

About Raspberry Pi

Welcome to the first chapter; I am happy that you are here. In this chapter, we will discuss the following topics:

- An introduction to Raspberry Pi.
- The history of Raspberry Pi.
- About Raspberry Pi 4.
- Accessories to be used with Raspberry Pi 4.
- The operating systems used with Raspberry Pi 4.
- How to install the operating system?

Let's continue reading.

About Raspberry Pi

According to Wikipedia, a computer is a machine that can be instructed to carry out sequences of arithmetic or logical operations automatically via computer programming. When I say “computer programming,” I mean that we tell the computer what it needs to do via a set of operations called *programs*.

You may be thinking, why I am giving an introduction to computer programming here? The reason is that I will call the Raspberry Pi device a *minicomputer*. The possibilities of this tiny device are endless. It can perform a huge variety of tasks.

Figure 1-1 shows the Raspberry Pi 4, which was released in June 2019 by the Raspberry Pi foundation.

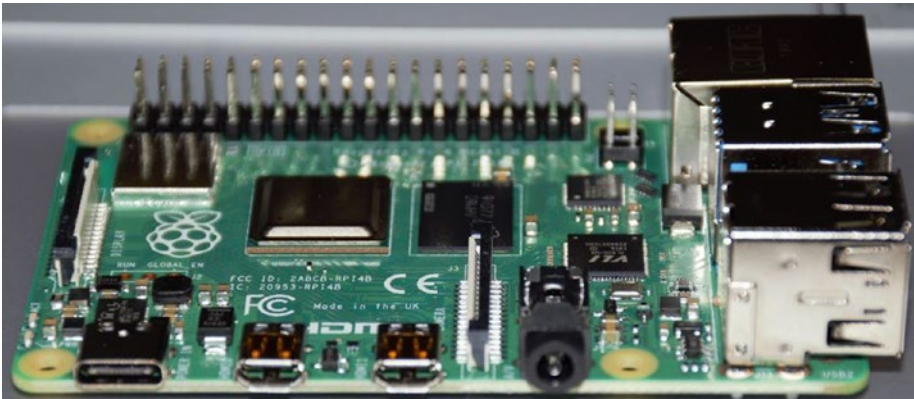


Figure 1-1. *The Raspberry Pi 4*

The History of the Raspberry Pi

The Raspberry Pi is a tiny computer that uses one single board. It was developed by the Raspberry Pi Foundation in the United Kingdom. The initial motive for this project was to promote teaching basic computer science in schools. The first version of the Raspberry Pi was released on February 24, 2012. The latest Raspberry Pi version is RPI 4, as of 2019.

There are many operating systems that we can run on a Raspberry Pi; some of them are listed here:

- Linux
- Windows 10 IoT Core (not supported by Raspberry Pi 3 or 4)

- Windows 10 ARM64
- FreeBSD
- NetBSD

As of now, the maximum memory supported by the Raspberry Pi is 4GB. It has memory variants of 1GB and 2GB as well.

Now you have an idea about this device, you might wonder how popular it is. Table 1-1 shows how many Raspberry Pi devices were sold in the last few years.

***Table 1-1.** Raspberry Pi Sales Over the Years*

Year	Sales
2015	5 million
2016	11 million
2017	15 million
2018	19 million

About Raspberry Pi 4

According to the creators of the Raspberry Pi, Raspberry Pi version 4 is the most advanced and efficient Pi they have ever made. Here are some reasons for such a claim:

- Dual 4K HDMI support.
- Fast data transfer with USB 3.0 and Gigabit Ethernet.
- Silent and energy efficient.
- Many variants, such as 1GB, 2GB, and 4 GB.
- Onboard wireless network connectivity and Bluetooth 5.0.

Now, let's look at the specifications of Raspberry Pi 4.

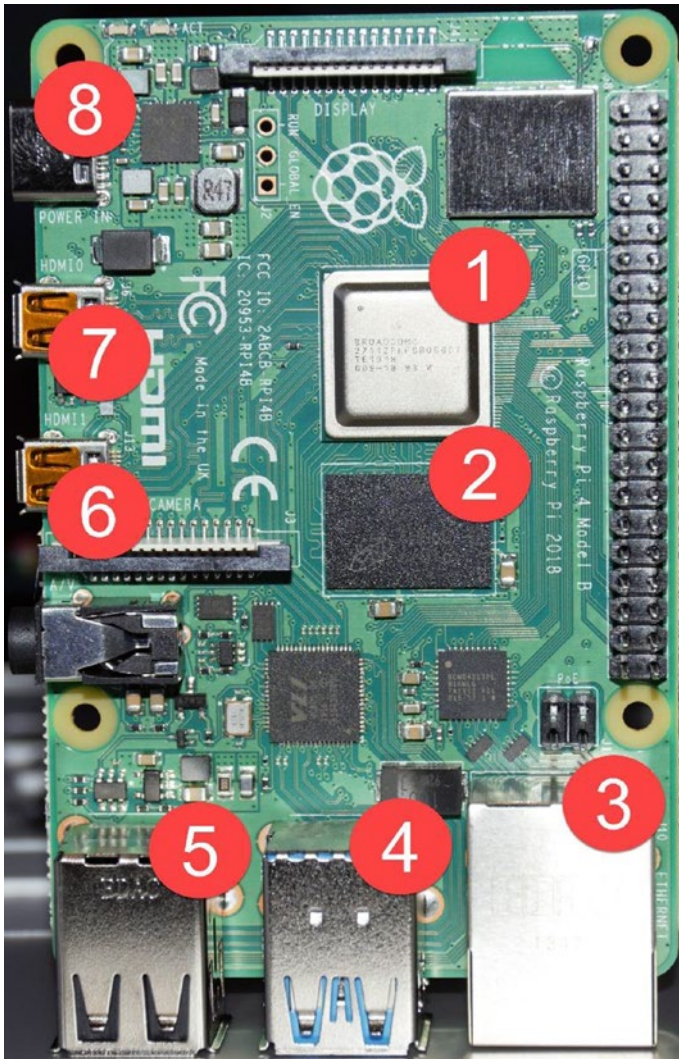


Figure 1-2. Raspberry Pi 4 B model

Here are explanations of the numbers shown in Figure 1-2:

1. *A more powerful processor.* The Raspberry Pi 4 uses Broadcom BCM2711 SoC with a 1.5GHz 64-bit quad-core ARM Cortex-A72 processor.
2. *RAM options.* With the Raspberry Pi 4, you can choose 1GB, 2GB, 4GB or 8GB RAM, depending on the model you select. This was not possible until Raspberry Pi 3, as the maximum RAM provided was 1GB.
3. *Gigabit Ethernet support.* The older version (Raspberry Pi 3) has only 100Mbit capacity.
4. *Two additional USB 3 ports.* Now you should be able to transfer your data 10 times faster. Happy transferring!
5. *Two USB 2 ports.*
6. *Micro HDMI support.* Raspberry Pi 4 provides dual 4K displays. I love this feature, as now I can connect two monitors to my Pi.
7. *Micro HDMI.*
8. *Support for a USB C power supply.* This is the first Pi that supports a USB Type C device.

Raspberry Pi 4 Accessories

If you are buying a Raspberry Pi alone from the store or online, keep in mind that you will not get any accessories with it. You need the following accessories to make it work, though.

- *Memory card.* You must have an SD card. You can select the memory card size. It purely depends on which operating system you are going to install. When you buy an SD card, just make sure that you follow these guidelines.
 - The minimum size requirement to write a Raspbian image is 8GB, and you can install the Lite image of Raspbian in 4GB. I always recommend a memory card of 16GB or 32GB so that you won't have to worry about space (see Figure 1-3).
 - Make sure that you buy a Class 10 memory card. The card class determines the write speed of your card. Class 10 has a write speed of 10MB/s and class 4 has 4MB/s.



Figure 1-3. 32GB memory card

- You should have a 5.1V/3A power adapter to charge your Raspberry Pi (see Figure 1-4).



Figure 1-4. 5.1V/3A power adapter

- You also need a Micro HDM cable to connect your Pi to the monitor (see Figure 1-5).



Figure 1-5. An HDMI cable

- Although you don't specifically need a case for your Pi, I do recommend you get one to keep it safe and clean (see [Figure 1-6](#)).



Figure 1-6. A Raspberry Pi 4 case

Introduction to the Operating System

The Raspberry Pi can be treated as a minicomputer, so we need an operating system to work with it. There are many suitable operating systems on the market now. When you buy a Raspberry Pi device kit, it may already have a default operating system installed on the memory card, which is the Raspbian Operating System. In this chapter, we explain how to install the Raspbian OS onto a memory card.

Raspbian

Raspbian is the one and only official operating system supported by the Raspberry Pi foundation. You can install the Raspbian OS either manually or with the help of NOOBS (New Out Of Box Software). As the name implies, NOOBS is an easy operating system installation manager for the Raspberry Pi.

Windows 10 IoT

As you might have already known, in the world of IoT, we categorize devices as single app devices or multi-app devices.

When you use an app to upload files to the cloud, this is an example of a single app device. Your mobile and smart watch are good examples of multi-app devices. I hope you get the idea.

In the same way, Microsoft introduced two editions of the Windows 10 IoT:

- Windows 10 IoT Core.
- Windows 10 IoT Enterprise.

Table 1-2 lists the differences between them.

Table 1-2. *Windows 10 IoT Core vs. Windows 10 IoT Enterprise*

Windows 10 IoT Core	Windows 10 IoT Enterprise
A version of Windows 10	The full version of Windows 10, it shares all the benefits of the worldwide Windows ecosystem
Optimized for smaller devices	Optimized for complex solutions
Receives fewer updates	Receives more updates than IoT Core
Single-app support, one foreground app at a time with a supporting background application	Multi-app support as traditional Windows
Only UWP UI supported	Full Windows UI supported (UWP, WinForms, etc.)

About Windows 10 IoT Core

Now that you have learned about Raspberry Pi and its relevant operating systems, it is time to start learning about Windows 10 IoT Core. Consider that the Windows 10 IoT Core is a version of Windows 10, but is made for smaller devices, for example, the Raspberry Pi. Although it is a smaller version of Windows 10, there are many differences. Some of these are listed in Table 1-3, which is updated as of September 2019. As the Windows 10 IoT core operating system is being updated, these differences may not be the same in the future releases.

Table 1-3. *Windows 10 vs. Windows 10 IoT Core*

Windows 10	Windows 10 IoT Core
FileOpenPicker API is supported	FileOpenPicker API is not supported
Desktop	Boot to the default app available in the operating system
Inbox Cortana is supported	Inbox Cortana is not supported
More supported drivers	Fewer supported drivers when compared to Windows 10
Support Remove-AppxPackage PowerShell command	Doesn't support Remove-AppxPackage PowerShell command

Installing the Raspbian Operating System

In this section, we discuss the ways that you can install the Raspbian operating system on your Raspberry Pi 4.

Using the Raspberry Pi Imager

The Raspberry Pi foundation created a tool called Raspberry Pi Imager, which allows you to easily write the Raspbian to an SD card. You can download it from the Raspberry Pi official website, at <https://www.raspberrypi.org/downloads/>. Once you are on the site, just select the Imager to be used as per your operating system. See Figure 1-7.

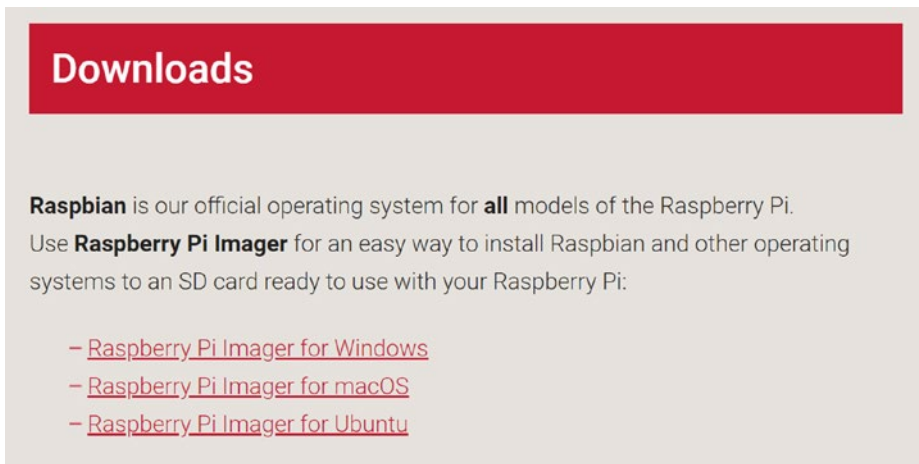


Figure 1-7. Select the Imager to be used

Once you install the application, you will be asked to select the operating system to be installed and to which SD card it must be installed. See Figure 1-8.



Figure 1-8. Choose the appropriate OS and card

Make sure that you have formatted your SD card; otherwise, the card will not be shown after you click on the Choose SD Card button. It is worth a mention that the name of the operating system is now Raspberry Pi OS; it was previously called Raspbian. See Figure 1-9.



Figure 1-9. *After OS and SD card selection*

Now click the Write button. The selected operating system will be added to the SD card. Figure 1-10 shows the pop-up you will see once the process is completed.

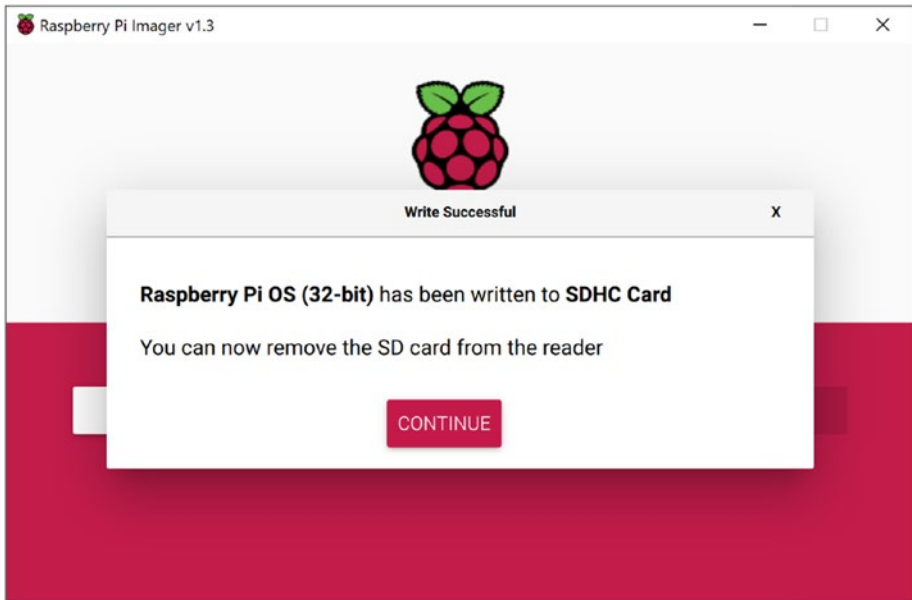


Figure 1-10. Raspbian OS installation complete

Manually Downloading the Image and Writing

The next way to install is to download the OS image from the official website (<https://www.raspberrypi.org/downloads/>). The image is in the downloaded ZIP archive file, and the file is over 4GB. As it uses ZIP64 to compress these files, we must use an unzip tool that supports ZIP64. Here are the supported unzip tools in different operating systems.

- 7 Zip: Windows.
- The Unarchiver: Mac.
- Unzip: Linux.

Note that the File Explorer in Windows XP does not support ZIP64, but Windows Vista and later do.

Once you have unzipped the file, you can write this image to your SD card by using any image-writing tools. My recommendation is to use the

balenaEtcher, which works on Windows, Linux, and macOS. If you are using balenaEtcher, you don't have to unzip the Raspbian image, as the tool supports writing images directly from the ZIP file. You can learn more about this tool at the official website at <https://www.balena.io/etcher/>.

Using NOOBS to Install the OS

Another way to install a Raspbian OS on your SD card is to use NOOBS. You can download NOOBS from <https://www.raspberrypi.org/downloads/>. The good thing about NOOBS is that it generically supports the installation of multiple operating systems, including these:

- Raspbian.
- Windows 10 IoT Core.
- LibreElec.
- Lakka.
- RISC OS.
- TLXOS.
- Screenly OSE.
- REcalbox.
- OSMC.

To set up the SD card, follow these steps:

1. Format the SD card.
2. Copy all the files from the extracted folder of the downloaded NOOBS file. Make sure you copy the contents, not the folder.

That's it. When you boot the device the first time, it will ask you to select the OS you want to install.

Summary

Congratulations, you have finished reading the first chapter! I am sure that you have learned these topics.

- What Raspberry Pi is
- The specifications of the Raspberry Pi 4 device and the accessories that you can use with it
- Operating systems to use with Raspberry Pi 4
- How to install the Raspbian Operating System on Raspberry Pi 4

Now let's move on to the next chapter.

CHAPTER 2

Configuring Your Raspberry Pi

In this chapter, you learn how to set up your Raspberry Pi for development and deployment. You will be performing the following tasks, to be precise:

- Enabling SSH on the Raspberry Pi
- Adding the network information to the Pi so that it can connect to the Wi-Fi
- Connecting the Pi to the Wi-Fi
- Connecting the Pi over SSH

Although these steps might sound a bit difficult, they are easily doable if you follow the instructions in this chapter.

Enabling SSH

Before you jump in and learn how to enable SSH, it is a good idea to understand what SSH is. The key points about SSH are as follows:

- SSH stands for Secure Shell, and it is a cryptographic network protocol.
- SSH uses operating network services securely over an unsecured network.
- All the operations performed—such as authentication, commands, output, file transfer, and so on—are encrypted to protect against network attacks.

Here are the usual steps involved in any SSH connection.

1. The client tries to contact the server and initiate the connection. In our case, we perform this task in a terminal.
2. The server sends the public key.
3. Next is the negotiation process. Once that is done, the secure channel will be opened.
4. Users log in to the server and performs the actions they are intended to perform.

To manually connect and deploy our applications to the Raspberry Pi and allow remote login, we must enable the SSH. If we don't enable the SSH, we will get the error `ssh: connect to host raspberrypi port 22: Connection refused`, as shown in Figure 2-1.

```
C:\Users\sibee>ssh pi@raspberrypi
ssh: connect to host raspberrypi port 22: Connection refused
```

Figure 2-1. Port 22 connection refused

To enable the SSH, follow these steps.

1. Run Notepad.
2. Click File ► Save As.

3. Be sure to set the Save As Type option to All Files to make sure that it is not saved as a text file. By default, Notepad files are saved with a .txt extension. See Figure 2-2.

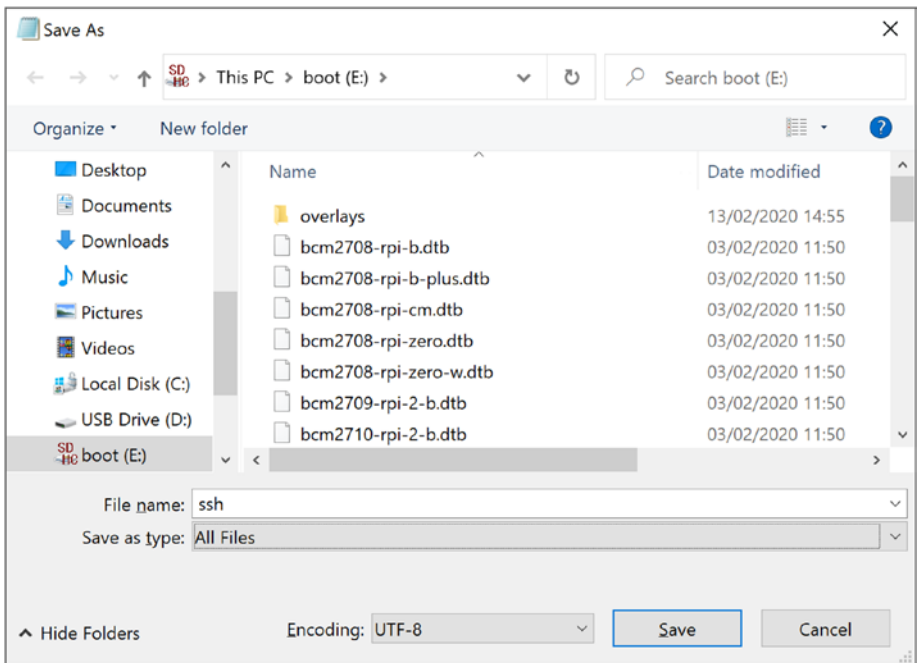


Figure 2-2. Save the SSH File

4. Save the file to the boot drive of your SD card.
5. Close the file.

If you are running on a Mac, you can directly run this command in the Terminal:

```
touch /Volumes/boot/ssh
```

Enabling Wi-Fi Configuration

In this section, you are going to connect the Raspberry Pi to the Wi-Fi. There are many ways you can do this; the easiest way is to follow these steps.

1. Run Notepad.
2. Paste the following code into a Notepad file. Don't forget to change the country code, network name, and network password to yours:

```
country=US
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={
  ssid="NETWORK-NAME"
  psk="NETWORK-PASSWORD"
}
```

3. Click File ► Save As.
4. Be sure to set the Save As Type option to All Files to make sure that the file is saved with the given extension. By default, Notepad files are saved with a .txt extension.
5. Name the file `wpa_supplicant.conf` and save it to the boot drive of your SD card.
6. Close the file.

If you are running on a Mac, you can use this command to generate the `wpa_supplicant.conf` file:

```
touch /Volumes/boot/wpa_supplicant.conf
```

Once the file is generated, open it and add the code mentioned above.

Checking Whether the Pi Is Connected to Wi-Fi

You have done enough configuration for now, so you can eject the SD card and put it back to the Raspberry Pi. Make sure that you connect the power cable to the Pi and wait for a minute or two to ensure it's connected to the given network.

To check the connection, you can go to the IP address of the default gateway of your network. The IP address will typically be 192.168.1.1 or 192.168.0.1. Just typing the IP address in the browser will open the Admin portal, where you can set the LAN, WAN, network management, and other options. Once you are logged in, you can see all the devices connected to the network. Figure 2-3 shows my router page as an example.

All devices connected to your Connect Box are listed below: Refresh

Device name	MAC address	IP address	Speed (Mbps)	Connected to
raspberrypi	DC:A6:32:0D:AD:03	192.168.0.80/24 2a02:8071:4191:aa00:5aa1:8961:c3e2:9398	39	Wi-Fi 5G UPC8305553
Galaxy-S9	16:A4:18:59:7C:43	192.168.0.10/24 2a02:8071:4191:aa00:9979:28c4:c41:3fd3	585	Wi-Fi 5G UPC8305553
DESKTOP-3EF5B65	C4:9D:ED:13:54:BD	192.168.0.206/24 2a02:8071:4191:aa00:a143:1958:eec1:c0c2	702	Wi-Fi 5G UPC8305553

Figure 2-3. All devices connected to the Wi-Fi

Connecting the Raspberry Pi via SSH

Now that the device is configured and connected to the network, you can connect the Pi by using SSH. To do this, you need three things:

- Hostname of the network.
- Username of the device.
- Password of the user.

By default, the username of the Raspberry Pi is Pi, the password is raspberry, and the hostname is raspberrypi.local. Open any command tool and enter the following command:

```
ssh pi@raspberrypi.local
```

You may see this warning:

```
"The authenticity of host 'raspberrypi.local (2a02:8071:4191:aa00:5aa1:8961:c3e2:9398)' can't be established.
ECDSA key fingerprint is SHA256:AkwljiM/KOrojYTMXJDxcP/GPmj4TFY+AkVM/QDtYs8.
Are you sure you want to continue connecting (yes/no)?"
```

You must provide yes here.

In the next step, you will be asked to type the password of the user Pi. If you provide the right username and password and if the connection is successful, you will get the output shown in Figure 2-4.

```
C:\Users\sibee>ssh pi@raspberrypi.local
The authenticity of host 'raspberrypi.local (2a02:8071:4191:aa00:5aa1:8961:c3e2:9398)' can't be established.
ECDSA key fingerprint is SHA256:AkwljiM/KOrojYTMXJDxcP/GPmj4TFY+AkVM/QDtYs8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'raspberrypi.local' (ECDSA) to the list of known hosts.
pi@raspberrypi.local's password:
Linux raspberrypi 4.19.97-v7l+ #1294 SMP Thu Jan 30 13:21:14 GMT 2020 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Apr  6 12:46:08 2020 from 2a02:8071:4191:aa00:a143:1958:eec1:c0c2

SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set a new password.

pi@raspberrypi:~$
```

Figure 2-4. SSH command output

Now that you have connected to the device remotely and it's on the network, it's a good idea to change the user password to something more secure. You can easily do that by running the following command in the SSH session you created:

```
sudo raspi-config
```

You will see the screen in [Figure 2-5](#).

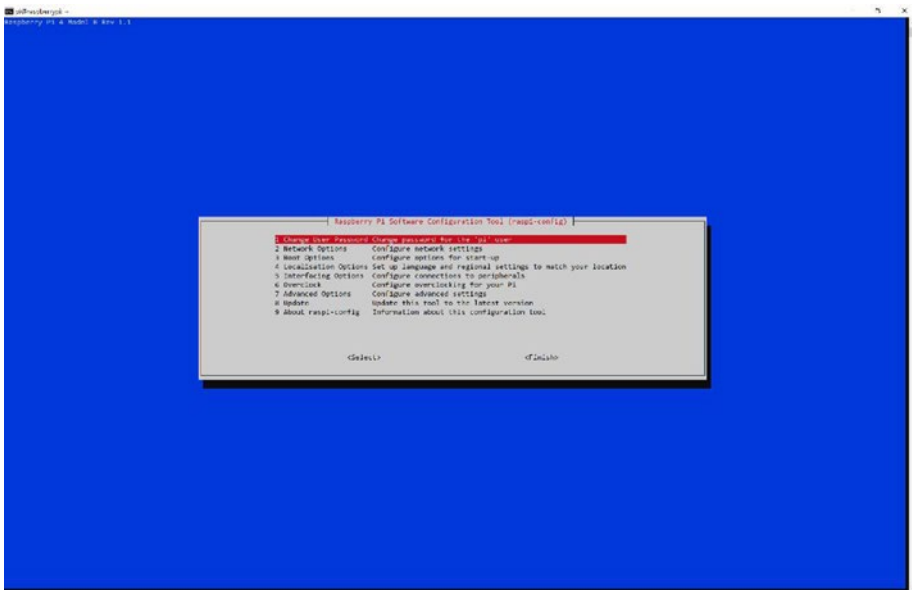


Figure 2-5. *Raspberry Pi configuration window*

Now select the Change Password option. Select OK from the next screen, shown in [Figure 2-6](#).

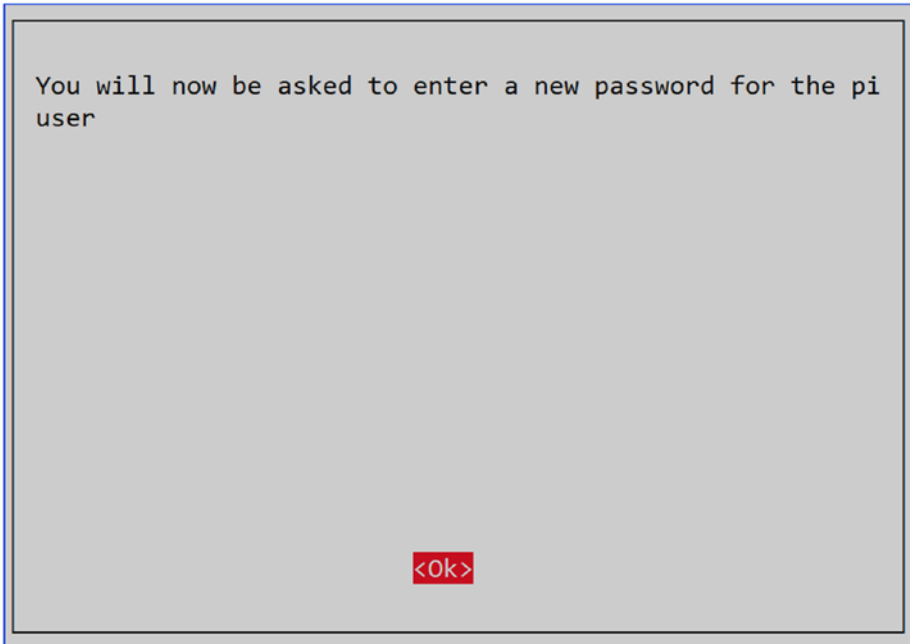


Figure 2-6. *Change Password warning window*

You will be asked to enter the new password for the user. Once you are done typing, press Enter. If everything went well, you should see the window in [Figure 2-7](#).

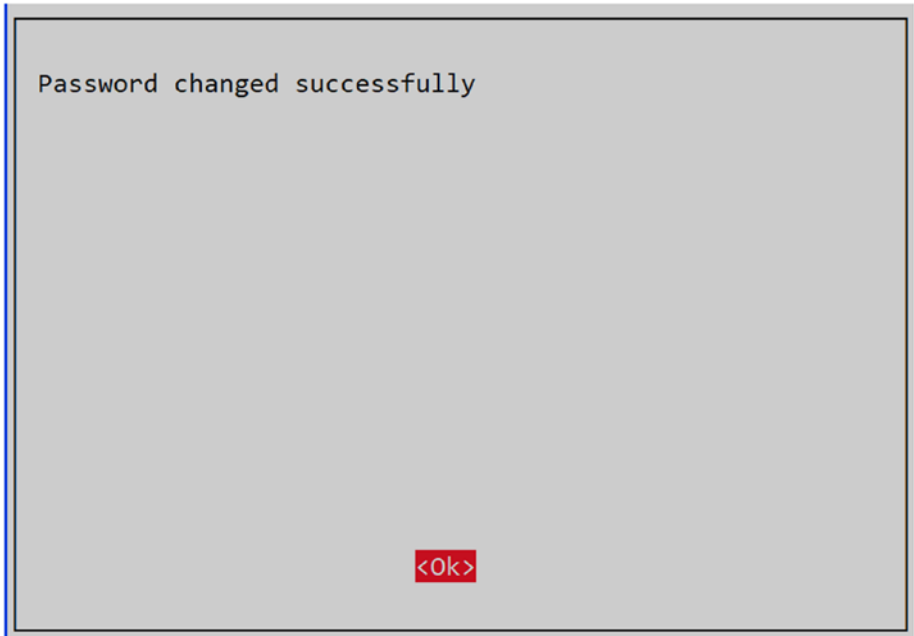


Figure 2-7. *Password Changed Successfully window*

Once you click OK and press Enter, the next screen will load, shown in Figure 2-8. Click the the Finish button and press Enter.

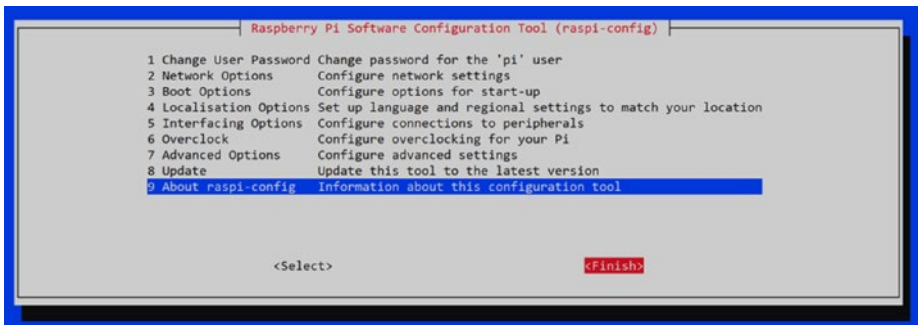


Figure 2-8. *Finish the configuration*

Summary

In this chapter, you learned the following:

- What SSH is and how to enable SSH on a Raspberry Pi device.
- How to set up Wi-Fi on a Raspberry Pi device.
- How to connect a Raspberry Pi device remotely using SSH.
- How to change the default password of the Pi user.

We have many things to cover and an application to develop. Let's jump on to the next chapter.

CHAPTER 3

Setting Up the Prerequisites to Develop the Application

In the last few chapters, we were busy setting up the Raspberry Pi device. In this chapter, we concentrate on setting up our machine to create a custom application so we can deploy and run it in our Raspberry Pi. This chapter assumes that you have already set up your device. If you have not, read the previous two chapters.

We will be developing the application using .NET Core, for several reasons:

- It is an open source, general-purpose development platform.
- It can be used to create applications for Windows, macOS, Linux, and ARM64 processors.

- It is widely supported and maintained by the community, so there are frameworks available for cloud, IoT, machine learning, and so on.
- It supports multiple languages, such as C#, F#, C++, and Visual Basic .NET.
- Good documentation is available to help users learn and use it.

In this chapter, I assume that you have enough knowledge on this topic. If you are not sure, read some of the articles at <https://docs.microsoft.com/en-au/dotnet/core>. You can also try some examples at <https://github.com/dotnet/core>. You can download the .NET Core from <https://dotnet.microsoft.com/download>.

It is worth mentioning that the Pi Zero and Pi 1 models are not supported. This is because the .NET Core JIT depends on armv7 instructions and they are not available in the earlier Pi versions.

Developing the Application

To develop the application, we are going to use Visual Studio code. Here are some reasons why I love Visual Studio Code (VSCoDe):

- It's lightweight and powerful.
- It's available for Windows, macOS, and Linux.
- It has built-in support for JavaScript, TypeScript, and Node.js.
- It has support for other languages, as provided by its wide varieties of extensions.

If you find any of these key points interesting, you can download this tool at <https://code.visualstudio.com>.

Using WSL

First, let's discuss what Windows Subsystem for Linux (WSL) is. The key points of WSL are as follows:

- It allows developers to run a GNU/Linux environment directly on Windows.
- It includes many command-line-tools, utilities, and applications.
- If there is no WSL, you have to create an additional Linux VM and run the programs there, which is a lot of work.

WSL vs. WSL2

The new version of WSL is WSL2. It has many advantages over the WSL version. WSL2 has better file system performance and it uses the latest virtualization technology. Since WSL2 uses a lightweight utility VM, it also uses less memory on startup.

To install WSL2, you should be running Windows 10 with build 19041 or higher (version 2004).

You can easily determine your Windows version by running the `winver` command in the command window (press the Windows key+R, and then type `winver`). You'll see a screen similar to Figure 3-1.



Figure 3-1. Windows 10 version

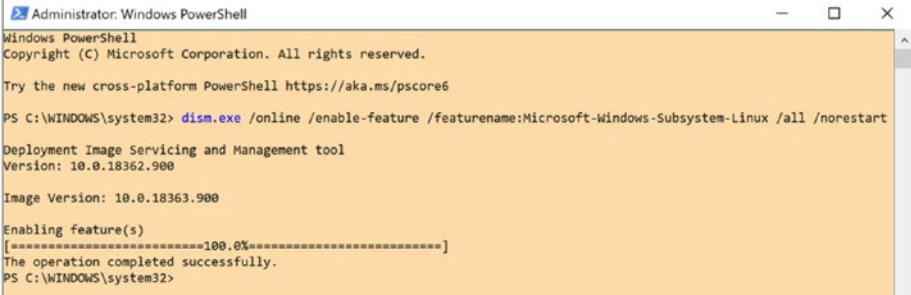
As you can see in Figure 3-1, my Windows version is 1909. I am using Surface Book 2, and the latest update is not yet compatible (as of writing this book).

Installing WSL

The optional feature, called Windows Subsystem for Linux, must be enabled before you can install any Linux distributions. To do that, run the following command in your PowerShell as the administrator:

```
dism.exe /online /enable-feature /featurename:Microsoft-  
Windows-Subsystem-Linux /all /norestart
```

You should see the output shown in Figure 3-2.



```

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

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\WINDOWS\system32> dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

Deployment Image Servicing and Management tool
Version: 10.0.18362.900

Image Version: 10.0.18363.900

Enabling feature(s)
[=====100.0%=====]
The operation completed successfully.
PS C:\WINDOWS\system32>

```

Figure 3-2. *Enabling Windows Subsystem for Linux*

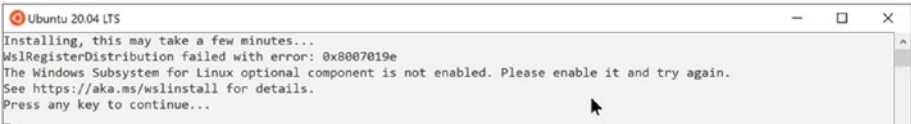
If your system meets the criteria needed to install WSL2, you must enable the Virtual Machine Platform feature. You can do that by running the following PowerShell command:

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

You can also set WSL2 as your default version. This comes in handy when you install a new Linux distribution.

```
wsl --set-default-version 2
```

Don't forget to restart your machine before you go to the next step. If you do not restart it, you will get the error shown in Figure 3-3.



```

Ubuntu 20.04 LTS
Installing, this may take a few minutes...
wslRegisterDistribution failed with error: 0x8007019e
The Windows Subsystem for Linux optional component is not enabled. Please enable it and try again.
See https://aka.ms/wslinstall for details.
Press any key to continue...

```

Figure 3-3. *Optional component is not enabled*

Installing the Linux Distribution

We are going to install the Linux distribution from the Microsoft Store. I recommend Ubuntu 20.04 LTS, as it is the latest version of Ubuntu. You can either open the <https://aka.ms/wslstore> URL in the browser, or search the Microsoft Store app and then search for Ubuntu (see Figure 3-4).

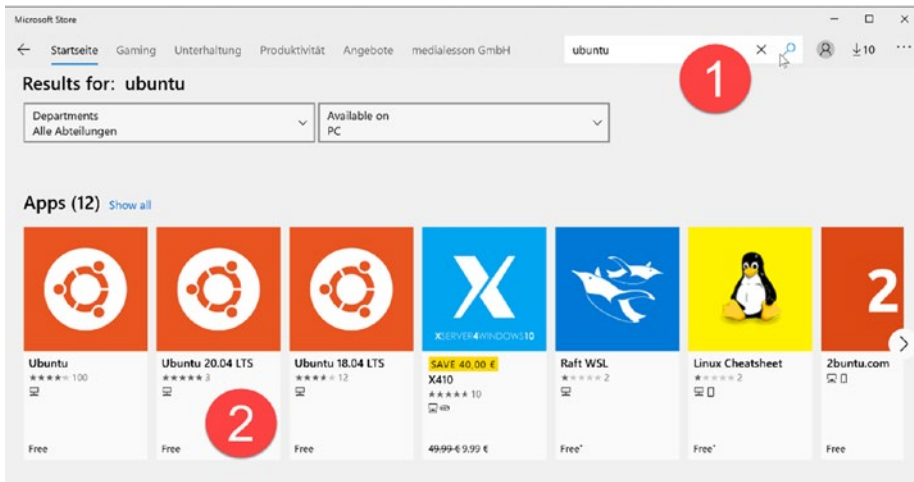


Figure 3-4. Microsoft Store search result

Now click the Get button, as shown in Figure 3-5.

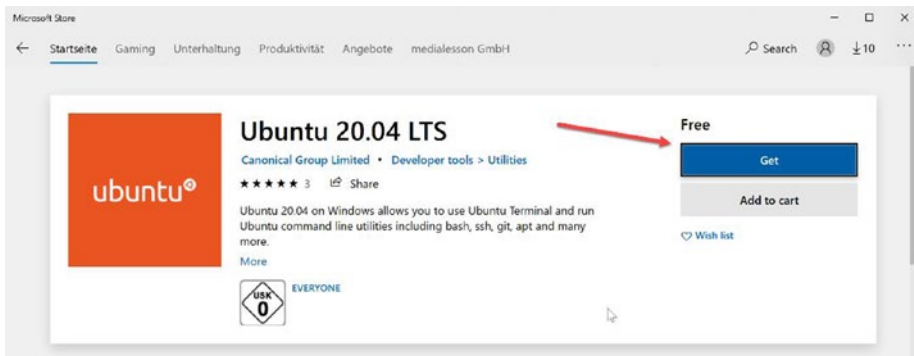


Figure 3-5. Get Ubuntu

The system will start downloading Ubuntu. The file is around 500MB, so the download can take a few minutes, depending on your network speed. Once the installation is complete, you can click the Launch button in the Microsoft Store.

The initial installation may take some time and the command screen will ask you to enter your proposed UNIX username. This username doesn't need to match your Windows username. You can be selective here.

After you set your username, you'll need to enter a password. Make sure you enter the correct password when asked to retype it. Figure 3-6 is for your reference.

```
sibeeshvenu@DESKTOP-SKPI6P9: ~
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: sibeeshvenu
New password:
Retype new password:
Sorry, passwords do not match.
passwd: Authentication token manipulation error
passwd: password unchanged
Try again? [y/N] y
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 20.04 LTS (GNU/Linux 4.4.0-18362-Microsoft x86_64)

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

System information as of Thu Jul  9 15:21:33 CEST 2020

System load:            0.52
Usage of /home:         unknown
Memory usage:          43%
Swap usage:            0%
Processes:             7
Users logged in:       0
IPv4 address for wif10: 192.168.1.105
IPv6 address for wif10: 2001:aa00:90bf:5ac4:fd6e:cc24
IPv6 address for wif10: 2001:aa00:a064:84b9:8f66:d4d0

0 updates can be installed immediately.
0 of these updates are security updates.

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

This message is shown once once a day. To disable it please create the
/home/sibeeshvenu/.hushlogin file.
sibeeshvenu@DESKTOP-SKPI6P9:~$ █
```

Figure 3-6. Ubuntu setup screen

Boom, you have successfully set up Subsystem for Linux in Windows!

Setting Up the Connection to Raspberry Pi

In this section, you will do the following tasks:

- Create a new SSH key.
- Copy the public key to Raspberry Pi. If you don't do this, you will be asked to type the password every time you deploy your app to Raspberry Pi.
- Install the Visual Studio debugger on the Raspberry Pi, which is needed in the next step, as you will be deploying your application.

This section assumes that your Raspberry Pi is known as `raspberrypi.local`. You need to make sure that you can connect to the Raspberry Pi via SSH. If you are on Linux, just open a new Terminal and type the following command.

```
ssh pi@raspberrypi.local
```

If you are on Windows, search for the keyword `wsl` in the Windows search box and then open the WSL Run Command to start a new Linux terminal command prompt. Once the prompt is opened, type the **ssh pi@raspberrypi.local** command to connect to your Raspberry Pi.

If you are using WSL1, you will receive this error:

```
ERROR: ssh: Could not resolve hostname raspberrypi.local: Name or service not known"
```

Note that this issue has been fixed in WSL2. To fix this error in WSL1, you have to change the hostname to the IP address of the Raspberry Pi. To get the IP address, run the `ping` command, as shown in Figure 3-7.

```
C:\Users\SibeeshVenu>ping raspberrypi.local

Pinging raspberrypi.local [2a02:8071:4191:aa00:1cb:35f5:e7e0:745c] with 32 bytes of data:
Reply from 2a02:8071:4191:aa00:1cb:35f5:e7e0:745c: time=2ms
Reply from 2a02:8071:4191:aa00:1cb:35f5:e7e0:745c: time=3ms
Reply from 2a02:8071:4191:aa00:1cb:35f5:e7e0:745c: time=3ms
Reply from 2a02:8071:4191:aa00:1cb:35f5:e7e0:745c: time=2ms

Ping statistics for 2a02:8071:4191:aa00:1cb:35f5:e7e0:745c:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 3ms, Average = 2ms
```

Figure 3-7. Ping raspberrypi.local

As you can see, the ping command shows the IPV6 address, but we need the IPV4 address. To get that, run the same command with -4 on the end. Here is how the command will look:

```
ping raspberrypi.local -4
```

Figure 3-8 shows the results.

```
C:\Users\SibeeshVenu>ping raspberrypi.local -4

Pinging raspberrypi.local [192.168.0.80] with 32 bytes of data:
Reply from 192.168.0.80: bytes=32 time=56ms TTL=64
Reply from 192.168.0.80: bytes=32 time=2ms TTL=64
Reply from 192.168.0.80: bytes=32 time=2ms TTL=64
Reply from 192.168.0.80: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.0.80:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 56ms, Average = 15ms
```

Figure 3-8. Ping raspberrypi.local IPV4

Now change the hostname in the previous command to the IP address. The new command should look like this:

```
ssh pi@192.168.0.80
```

Next, update the Raspberry Pi OS and reboot it.

```
sudo apt update && sudo apt upgrade && sudo reboot
```

You should see the output shown in Figure 3-9.

```
Setting up libavdevice58:armhf (7:4.1.6-1~deb10u1+rpt1) ...
Setting up ffmpeg (7:4.1.6-1~deb10u1+rpt1) ...
Processing triggers for desktop-file-utils (0.23-4) ...
Processing triggers for mime-support (3.62) ...
Processing triggers for hicolor-icon-theme (0.17-2) ...
Processing triggers for gnome-menus (3.31.4-3) ...
Processing triggers for libglib2.0-0:armhf (2.58.3-2+deb10u2) ...
Processing triggers for libc-bin (2.28-10+rpi1) ...
Processing triggers for systemd (241-7~deb10u4+rpi1) ...
Processing triggers for man-db (2.8.5-2) ...
Processing triggers for shared-mime-info (1.10-1) ...
Setting up pi-package (0.7) ...
Processing triggers for initramfs-tools (0.133+deb10u1) ...
Processing triggers for ca-certificates (20200601~deb10u1) ...
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
Processing triggers for libvlc-bin:armhf (3.0.11-0+deb10u1+rpt1) ...
Connection to 192.168.0.80 closed by remote host.
Connection to 192.168.0.80 closed.
sibeeshvenu@DESKTOP-SKPI6P9:/mnt/c/WINDOWS/system32$ █
```

Figure 3-9. Updating the Raspberry Pi OS

We now need to generate a new SSH certificate and copy the public certificate to Raspberry Pi. To do that, we'll run the previous command in the same command prompt. First make sure that you are in the current user's home directory before you run the command. By default, the command prompt will open in the Windows System32 folder. It is not recommended you modify this folder. Figure 3-10 shows the results when we run the `cd` command.

```
sibeeshvenu@DESKTOP-SKPI6P9: ~
sibeeshvenu@DESKTOP-SKPI6P9:/mnt/c/WINDOWS/system32$ cd
sibeeshvenu@DESKTOP-SKPI6P9:~$
```

Figure 3-10. User's home directory

Now you can run the command:

```
ssh-keygen -t rsa && ssh-copy-id pi@192.168.0.80
```

You should see the output in Figure 3-11.

```
sibeeshvenu@DESKTOP-SKPI6P9: ~
sibeeshvenu@DESKTOP-SKPI6P9:/mnt/c/WINDOWS/system32$ cd
sibeeshvenu@DESKTOP-SKPI6P9:~$ ssh-keygen -t rsa && ssh-copy-id pi@192.168.0.80
Generating public/private rsa key pair.
Enter file in which to save the key (/home/sibeeshvenu/.ssh/id_rsa):
/home/sibeeshvenu/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/sibeeshvenu/.ssh/id_rsa
Your public key has been saved in /home/sibeeshvenu/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:2+mnyZsgdc1LmK9UNfmHtWSlXwI4pxwJwcSakv5gGEA sibeeshvenu@DESKTOP-SKPI6P9
The key's randomart image is:
+----[RSA 3072]-----+
|.E   ++o o.  .|
|.   o =... o |
|.  . o .+= *  |
|. o o+ .oo = + |
|+ .o S . o o .|
|. + = * . . . |
|. oo = o      |
|. .o +.o     |
|.  . oB.     |
+----[SHA256]-----+
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/sibeeshvenu/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
pi@192.168.0.80's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'pi@192.168.0.80'"
and check to make sure that only the key(s) you wanted were added.

sibeeshvenu@DESKTOP-SKPI6P9:~$
```

Figure 3-11. Copy SSH to Raspberry Pi

To install the Visual Studio Code .NET debugger in Raspberry Pi, you simply run the following command. Keep in mind that you can use your Raspberry Pi name instead of the IP address if you are running WSL2.

```
ssh pi@192.168.0.80 "curl -sSL https://aka.ms/getvsdbgsh | bash
/dev/stdin -r linux-arm -v latest -l ~/vsdbg"
```

You should see the output in Figure 3-12.

```

Access granted. Press Return to begin session.
Info: Previous installation at '/home/pi/vsdbg' not found
Info: Using vsdbg version '16.6.20415.1'
Info: Creating install directory
Using arguments
  Version           : 'latest'
  Location          : '/home/pi/vsdbg'
  SkipDownloads     : 'false'
  LaunchVsDbgAfter : 'false'
  RemoveExistingOnUpgrade : 'false'
Info: Using Runtime ID 'linux-arm'
Downloading https://vsdebugger.azureedge.net/vsdbg-16-6-20415-1/vsdbg-linux-arm.tar.gz
Info: Successfully installed vsdbg at '/home/pi/vsdbg'

```

Figure 3-12. *Installing the debugger*

Installing .NET Core on Ubuntu

Since we will be creating a .NET Core application, we need to install .NET Core next. Go to the .NET Core download page at <https://dotnet.microsoft.com/download> and install the .NET Core SDK. If you are running WSL in Windows 10 like me, you need to install the SDK on the Windows WSL Linux distribution.

First, add the Microsoft package signing key to your list of trusted keys and add the package repository. To do that, run the following command at the WSL command prompt:

```

wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb

```

This command will ask you to type the user password. If you forgot the password, you can open a new command window and run the following command to open the WSL as a root user:

```
wsl --user root
```

Now type the following command to change the user password. You will be asked to type the new password:

```
passwd sibeeshvenu
```

Figure 3-13 shows this process.

```
root@DESKTOP-SKPI6P9:/mnt/c/Users/SibeeshVenu# passwd sibeeshvenu
New password:
Retype new password:
passwd: password updated successfully
root@DESKTOP-SKPI6P9:/mnt/c/Users/SibeeshVenu#
```

Figure 3-13. Changing the user password in WSL

Figure 3-14 shows the output of adding the Microsoft package signing key.

```
sibeeshvenu@DESKTOP-SKPI6P9:~$ wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
Will not apply HSTS. The HSTS database must be a regular and non-world-writable file.
ERROR: could not open HSTS store at '/home/sibeeshvenu/.wget-hsts'. HSTS will be disabled.
--2020-07-12 09:50:51-- https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb
Resolving packages.microsoft.com (packages.microsoft.com)... 104.214.230.139
Connecting to packages.microsoft.com (packages.microsoft.com)|104.214.230.139|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3124 (3.1K) [application/octet-stream]
Saving to: 'packages-microsoft-prod.deb'

packages-microsoft-prod.deb          100%[=====]
2020-07-12 09:50:51 (90.0 MB/s) - 'packages-microsoft-prod.deb' saved [3124/3124]

sibeeshvenu@DESKTOP-SKPI6P9:~$ sudo dpkg --add-architecture deb --foreign --get http://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb
[sudo] password for sibeeshvenu:
Selecting previously unselected package packages-microsoft-prod.
(Reading database ... 31836 files and directories currently installed.)
Preparing to unpack packages-microsoft-prod.deb ...
Unpacking packages-microsoft-prod (1.0-ubuntu20.04.1) ...
Setting up packages-microsoft-prod (1.0-ubuntu20.04.1) ...
sibeeshvenu@DESKTOP-SKPI6P9:~$
```

Figure 3-14. Adding the Microsoft package signing key

Now you can install the .NET Core SDK by running the following command.

```
sudo apt-get update; \
sudo apt-get install -y apt-transport-https && \
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-3.1
```

If you don't get any errors from the Terminal, you are good to go. Congratulations!

Summary

In this chapter, you learned about the following topics:

- What VSCode is and why it is so popular.
- What WSL is and why is it important.
- How to enable WSL and the differences between WSL1 and WSL2.
- How to install the Linux Distribution (Ubuntu) on Windows 10.
- How to set up Raspberry Pi to connect with Ubuntu.
- How to install .NET Core on Ubuntu.

Now take a deep breath and relax. You did well! It's a good time to take a coffee break.

CHAPTER 4

Creating and Deploying a .NET Core Application to Raspberry Pi

In the last chapter, we set up our machine. In this chapter, we are going to create a .NET Core application and deploy it to Raspberry Pi. This chapter assumes that you followed all the steps mentioned in Chapter 3. If you are not sure, please review that chapter.

Creating a .NET Core Application

By just running two commands, you can easily create a sample dummy application in VSCode. Later, we will be editing that solution. Sound good? If it does, open a new terminal and run the following commands.

```
mkdir raspberrypi.net.core && cd raspberrypi.net.core
```


This command will generate a new folder called `raspberrypi.net.core` and change the current working directory to that new folder. This `cd` command is also known as `chdir` (change directory).

Now run the following command to generate a new solution.

```
dotnet new console --langVersion=latest && dotnet add package  
iot.device.bindings
```

If everything goes well, it will do the following:

- Create an empty console application, since we provided the console template in the command.
- Perform some post-creation actions, such as `dotnet restore`.
- Add the package reference that we provided in the command. In this case, it is `iot.device.bindings`. This `iot.device.bindings` package gives us a set of device bindings that use `system.device.gpio` to communicate with a microcontroller.
- Restore the given package.

Once the restore is finished, all the required files and references will be available in our solution. The sample output is shown in Figure 4-1.

```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode>mkdir raspberrypi.net.core && cd raspberrypi.net.core

C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core>dotnet new console --langVersion=latest && dotnet add package iot.device.bindings
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj...
Restore completed in 149,06 ms for C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj.

Restore succeeded.

Writing C:\Users\sibee\AppData\Local\Temp\tmp1180.tmp
Info : Adding PackageReference for package 'iot.device.bindings' into project 'C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj'.
Info : Restoring packages for C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj...
Info : GET https://api.nuget.org/v3-flatcontainer/iot.device.bindings/index.json
Info : OK https://api.nuget.org/v3-flatcontainer/iot.device.bindings/index.json 495ms
Info : Package 'iot.device.bindings' is compatible with all the specified frameworks in project 'C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj'.
Info : PackageReference for package 'iot.device.bindings' version '1.0.0' added to file 'C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj'.
Info : Committing restore...
Info : Writing assets file to disk. Path: C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\obj\project.assets.json
log : Restore completed in 2,59 sec for C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core\raspberrypi.net.core.csproj.

C:\Users\sibee\Njan Oru Malayali\books - Documents\Apress\Asp.Net Core and Azure with Raspberry Pi 4\SourceCode\raspberrypi.net.core>

```

Figure 4-1. Project creation output

Let's open this folder and see which files were generated; see Figure 4-2.

Name	Status	Date modified	Type	Size
obj	✓	07/04/2020 11:31	File folder	
Program	✓	07/04/2020 11:31	C# Source File	1 KB
raspberrypi.net.core	✓	07/04/2020 11:31	Visual C# Project file	1 KB

Figure 4-2. Created project folder structure

You can see that there is one folder called `obj`. This is where the temporary object files and other files are stored to create the final library. The final library is stored in the folder called `bin`, which we will explain once we compile our application. Now just right-click the root folder and open it in Visual Studio Code. If you don't see the Open with Code option in the right-click menu, the easiest way to fix that issue is to reinstall VSCode and make sure to select the check boxes shown in Figure 4-3.

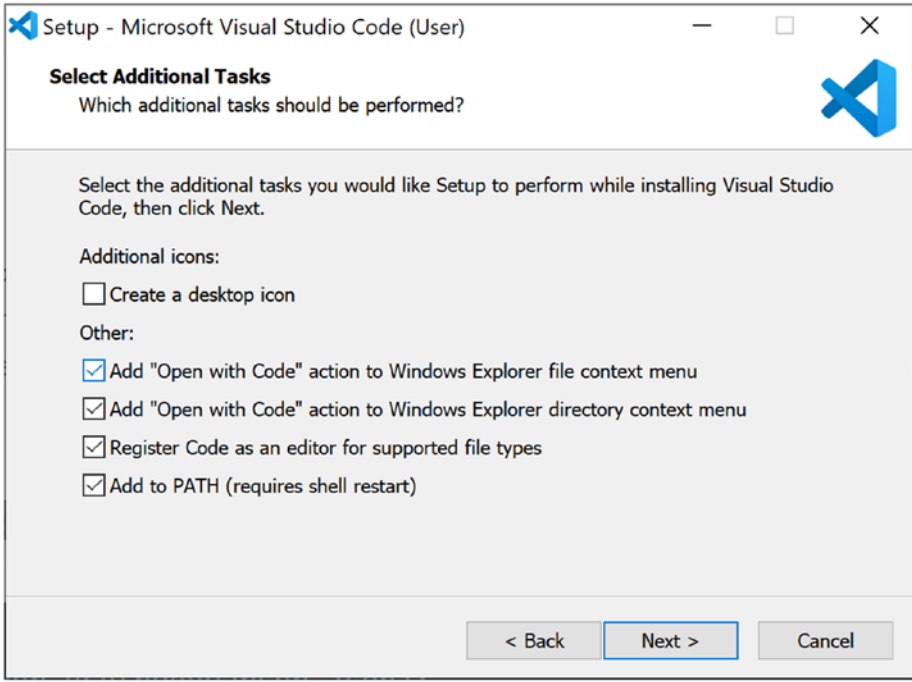


Figure 4-3. *Open with Code options should be selected*

Note that you can also run `code .` from the command-line tool itself. Visual Studio Code will then open the project in the current folder.

If your VSCode displays a popup saying that a C# extension is recommended when you open the `Program.cs` file, go ahead and install it. See Figure 4-4.

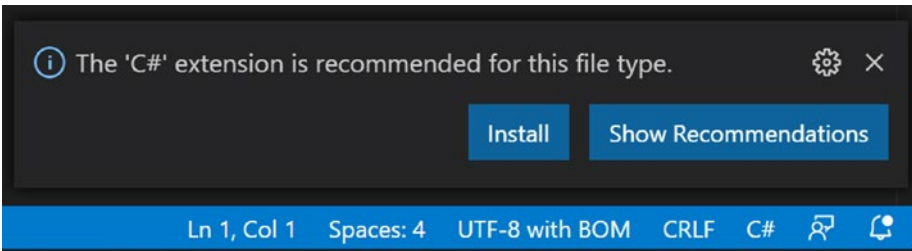


Figure 4-4. *C# extension for VSCode*

If you have ever wondered what is happening when you install an extension, you can see that process in the output window, shown in Figure 4-5.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Installing C# dependencies...
Platform: win32, x86_64

Downloading package 'OmniSharp for Windows (.NET 4.6 / x64)' (33895 KB)..... Done!
Validating download...
Integrity Check succeeded.
Installing package 'OmniSharp for Windows (.NET 4.6 / x64)'

Downloading package '.NET Core Debugger (Windows / x64)' (42010 KB)..... Done!
Validating download...
Integrity Check succeeded.
Installing package '.NET Core Debugger (Windows / x64)'

Downloading package 'Razor Language Server (Windows / x64)' (50668 KB)..... Done!
Installing package 'Razor Language Server (Windows / x64)'

Finished

```

Figure 4-5. *C# extension installation output*

Now it's time to run the application the first time. The easiest way is to press the F5 button; however, you can always go to the Run menu and click the Start Debugging submenu. If it asks you to select the environment, choose .NET Core.

A file called `launch.json` will be generated in the `.vscode` folder. This is where we set the debugging options for our application. You can also see that there is another file generated in the same folder, called `task.json`. This file specifies how to compile the project. We will be talking about these files in a coming section.

If everything goes well, you should see a "Hello World!" message in your debug console. If you are not sure where this message comes from, just look at the `Program.cs` file. This is how your `Program.cs` file looks:

```
using System;

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

You will be rewriting this code later. But before you do that, you need to make some changes to the Visual Studio Code.

Installing Visual Studio Code Remote WSL Extension

If you have ever wondered how to use the Windows Subsystem for Linux (WSL) as your full-time development environment right from VSCode, Visual Studio Code Remote WSL extension is the answer. It allows you to use a Linux-based environment, including the toolchains and utilities, and run and debug Linux-based applications from Windows. Sound good?

If you get the prompt shown in Figure 4-6 when you open your project in VSCode, go ahead and install it.

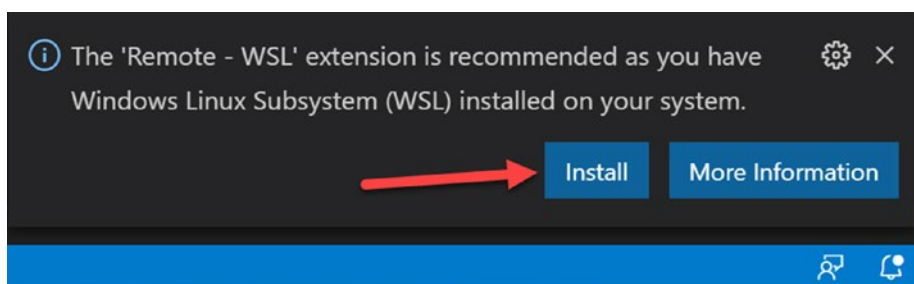


Figure 4-6. *Install Remote WSL*

You can also install it from the Extensions page in VSCode, as shown in Figure 4-7.

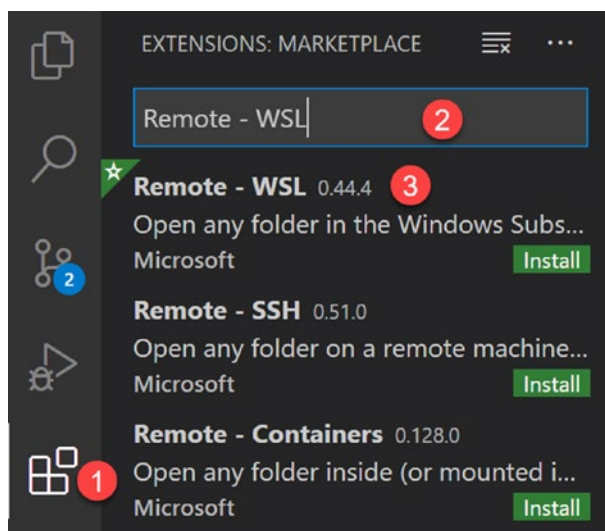


Figure 4-7. *Install Remote WSL from the Extensions page*

After the installation, reopen your VSCode. If you get a prompt saying that “Required assets are missing,” as in Figure 4-8, select Yes to add them.

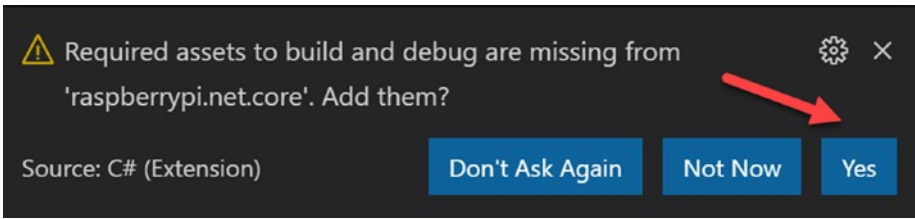


Figure 4-8. Required assets to build and debug

The project should already be opened in the VSCode, so we need to open it in WSL. To do that, press F1 and begin typing Remote-WSL: Reopen in WSL. As you type the words, the option should appear and you can select it. If the project is not opened in VSCode, you can directly open it in WSL by typing and selecting Remote-WSL: New Window Using Distro. See Figure 4-9.



Figure 4-9. Reopen the folder in WSL

This will reopen the VSCode in WSL and you should see the indication in your VSCode, as shown in Figure 4-10.

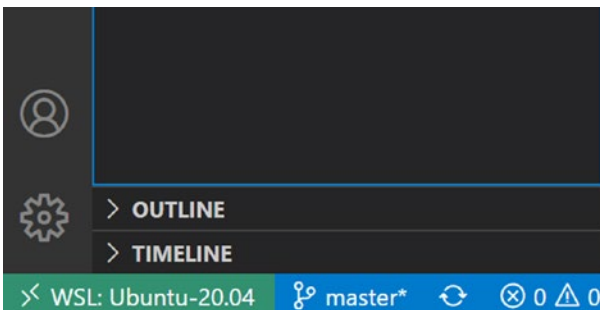


Figure 4-10. Editing on WSL Ubuntu

Rewriting the Application

In this section, we rewrite the `Program.cs` file to read the CPU temperature of the Raspberry Pi device. This is how your file will look.

```
using System;
using Iot.Device.CpuTemperature;
using System.Threading;
namespace raspberrypi.net.core
{
    class Program
    {
        private static CpuTemperature rpiCpuTemp = new
            CpuTemperature();
        static void Main(string[] args)
        {
            while (true)
            {
                if (rpiCpuTemp.IsAvailable)
                {
                    Console.WriteLine($"The CPU temperature
                        at { DateTime.Now } is { rpiCpuTemp.
                            Temperature.Celsius }");
                }
                Thread.Sleep(1000);
            }
        }
    }
}
```


Deploying the App to Raspberry Pi

The first thing to do is to make sure that your Raspberry Pi is switched on and connected to the Wi-Fi. We will be editing the `launch.json` file first. Let's talk about that file now. It contains attributes such as name, type, and request. These are mandatory attributes of a `launch.json` file.

The Name Attribute

This attribute provides a meaningful name to your configuration. It's found in the Debug launch configuration drop-down, as shown in Figure 4-11.

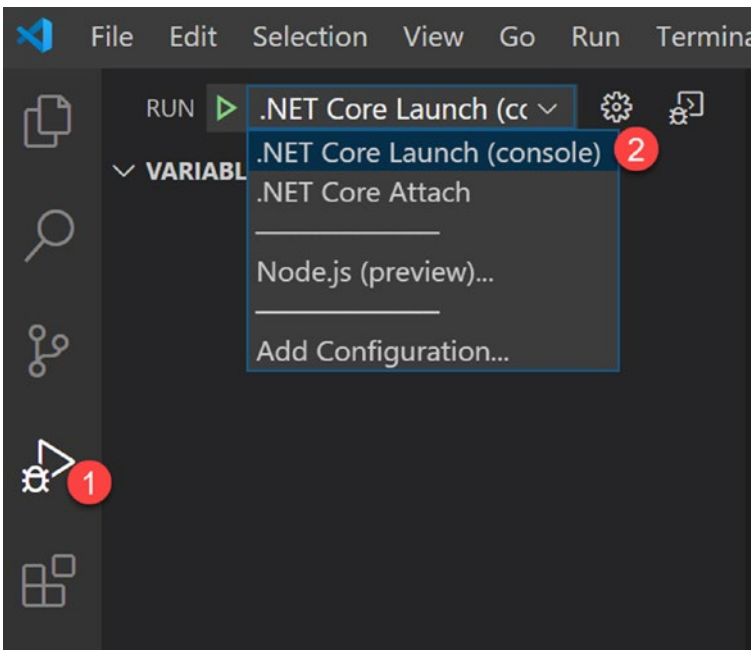


Figure 4-11. Launch JSON name attribute

The Request Attribute

Currently (as of June 2020), `request` has two supported values—`launch` and `attach`. The easy way to explain the difference between `launch` and `attach` is to think of a `launch` configuration as a recipe for how to start your application in debug mode before VSCode attaches to it, while an `attach` configuration is a recipe for how to connect VSCode's debugger to a process that is already running.

The Type Attribute

This attribute sets the type of debugger to use with this launch configuration. It depends on the environment. As you select an environment, each debug extension has a different type. For example, the `node` type is for the built-in node debugger.

Another important attribute in the `launch.json` file is `preLaunchTask`. This launches a task before the start of a debug session. We will define this task in the `task.json` file and use that task name here.

To deploy the application to Raspberry Pi, we must compile this application for Linux arm, and we can do that by configuring our VSCode. We do this configuration using the `task.json` file. Once we compile the application, we will copy our program to Raspberry Pi, using `Rsync`.

The Rsync Attribute

There are many ways that you can transfer your files from your computer to Raspberry Pi. Two popular ways are using `SCP` and `Rsync`. There are significant differences between those two. Let's cover a few of them:

- `SCP` stands for Secure Copy Protocol. It reads the source file and writes to the destination by performing a plain linear copy. `Rsync` also does the same, but with an efficient algorithm and a few optimizations that make the transfer faster.

- Instead of just copying, Rsync will check for the file sizes and modification timestamps to make sure only changes or differences are copied. This makes the transfer a lot faster. It's also easier to synchronize the files on the source and destination.
- Rsync has an option for resuming the transfer if it is interrupted, whereas SCP doesn't have this feature.
- Last but not the least, according to OpenSSH developers on Wikipedia in 2019, the SCP protocol is outdated, and they recommend using modern protocols like Rsync.

Feel free to check out more about Rsync at <https://rsync.samba.org/>.

Let's open our `task.json` file and update the code with Rsync. This file is responsible for the following jobs:

- Compiling the application.
- Copying the code to Raspberry Pi (yes, we use Rsync for this purpose).

Now you can edit the `task.json` file as follows.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "RpiPublish",
      "command": "sh",
      "type": "shell",
      "problemMatcher": "$msCompile",
      "args": [
        "-c",
```

```

        "\"dotnet publish -r linux-arm -c
        Debug -o ./bin/linux-arm/publish
        ./${workspaceFolderBasename}.csproj && rsync -rvuz
        ./bin/linux-arm/publish/ pi@192.168.0.80:~/${wo
        rkspaceFolderBasename}\"",
    ]
}
]
}
}

```

Here, `RpiPublish` is our task name and you might have noticed that we use the `rsync -rvuz` command. This copies the files from our computer to the Raspberry Pi.

Let's use this task in the `launch.json` file now. The `launch.json` file is responsible for these jobs:

- Calling the build tasks.
- Asking Raspberry Pi to start the Visual Studio Code debugger.
- Loading the application.

In the end, this is how your `launch.json` file will look.

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Rpi Publish and Debug",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "RpiPublish",
      "program": "~/${workspaceFolderBasename}/${workspace
      FolderBasename}",
    }
  ]
}

```

```

        "cwd": "~/${workspaceFolderBasename}",
        "stopAtEntry": false,
        "console": "internalConsole",
        "pipeTransport": {
            "pipeCwd": "${workspaceRoot}",
            "pipeProgram": "/usr/bin/ssh",
            "pipeArgs": [
                "pi@192.168.0.80"
            ],
            "debuggerPath": "~/vsdbg/vsdbg"
        }
    }
]
}

```

Note that I used `pi@192.168.0.80` everywhere, because I am running WSL version 1. If you are running in Linux or using WSL2, you should change that to `pi@raspberrypi.local`.

Variables in VSCode

You can see that we are using the `${workspaceRoot}`, `${workspaceFolder}`, and `${workspaceFolderBasename}` variables in our `task.json` and `launch.json` files. These are VSCode's predefined variables. The syntax to use them is `${variablename}`. Some of the other variables are listed in Table 4-1.

Table 4-1. VSCode Variables

Variable Name	Description
<code>\${workspaceFolder}</code>	The path of the folder opened in VSCode
<code>\${workspaceFolderBasename}</code>	The name of the folder opened in VSCode without any slashes (/)
<code>\${file}</code>	The current opened file
<code>\${relativeFile}</code>	The current opened file relative to workspaceFolder
<code>\${relativeFileDirname}</code>	The currently opened file's dirname relative to workspaceFolder
<code>\${fileBasename}</code>	The currently opened file's basename
<code>\${fileBasenameNoExtension}</code>	The currently opened file's basename with no file extension
<code>\${fileDirname}</code>	The currently opened file's dirname
<code>\${fileExtname}</code>	The currently opened file's extension
<code>\${cwd}</code>	The task runner's current working directory on startup
<code>\${lineNumber}</code>	The currently selected line number in the active file
<code>\${selectedText}</code>	The currently selected text in the active file
<code>\${execPath}</code>	The path to the running VSCode executable
<code>\${defaultBuildTask}</code>	The name of the default build task

You can also create a new task to view the values of each variable. So let's create a new task in the `task.json` file.

```
{
    "label": "Echo VSCode Variables",
    "type": "shell",
    "command": "echo ${workspaceRoot} | echo ${work
        spaceFolder} | echo ${workspaceFolderBasename}"
}
```

Now press the F1 button and type `Tasks: Run Task`. Select the `Echo VSCode Variables` and the `Continue Without Scanning the Output` options. The variables' values will appear in the terminal, as shown in [Figure 4-12](#).

```
> Executing task: echo C:\Sibeesh\Github\raspberrypi.net.core | echo C:\Sibeesh\Github\raspberrypi.net.core | echo raspberrypi.net.core <
raspberrypi.net.core
```

Figure 4-12. Showing the VSCode variables

Debugging the App from Raspberry Pi

Keep in mind that you should install the `ms-dotnettools.charp` extension in your WSL too. See [Figure 4-13](#).



Figure 4-13. Install the `ms-dotnettools` extension

Now it's time to run the application and deploy it to the Raspberry Pi. Click Run and add a breakpoint on any line you wish in `Program.cs`. Select the Rpi Publish and Debug configuration and then click the green icon on the left side of the configuration. See Figure 4-14.

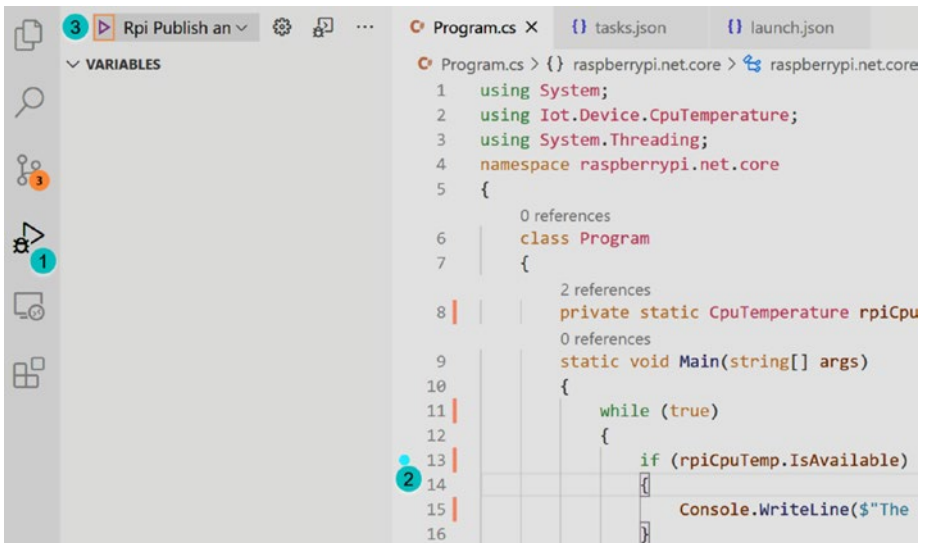


Figure 4-14. How to debug

Once you click the green icon, you will see a lot of things happening in the terminal. Figures 4-15 and 4-16 show the output of the debug session.

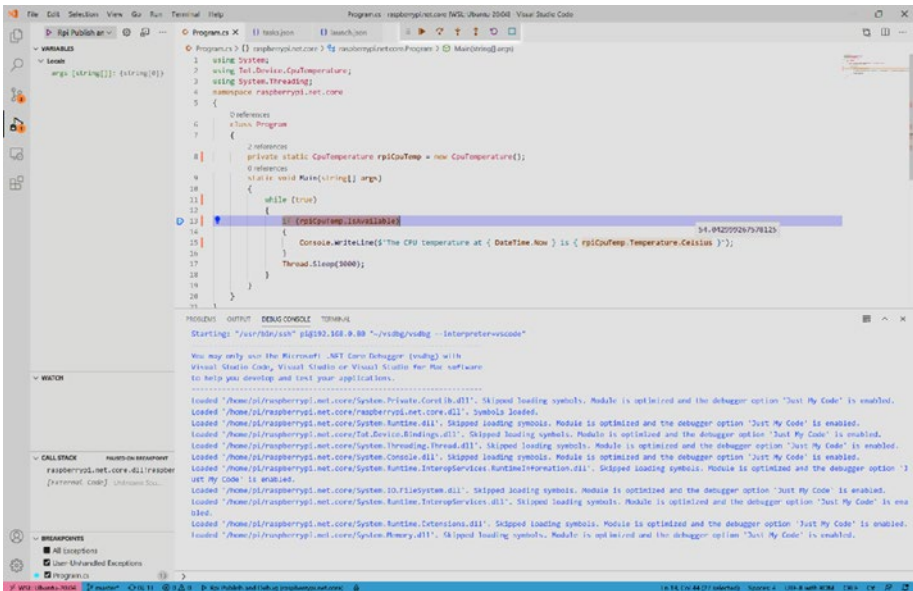


Figure 4-15. The Debug window

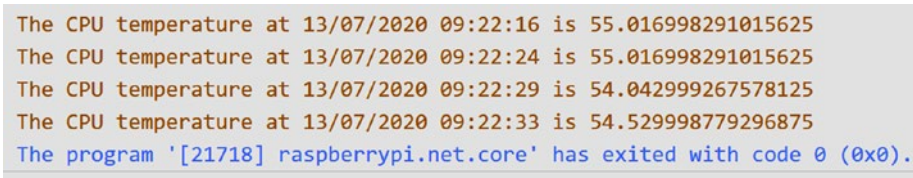


Figure 4-16. The Debug window console

Summary

Wow, that was amazing, right? In this chapter, you learned the following:

- How to create a .NET Core application?
- How to install Visual Studio Code Remote with the WSL extension?

CHAPTER 4 CREATING AND DEPLOYING A .NET CORE APPLICATION TO RASPBERRY PI

- How to read the CPU temperature of the Raspberry Pi device?
- How to deploy the application to Raspberry Pi?
- What Rsync is?
- The variables in VSCode.
- How to debug the application running in the Raspberry Pi device in VSCode?

Are you excited for the next chapter? I can't wait to see you there.

CHAPTER 5

Playing with Azure IoT Hub and Our Application

In the last chapter, you learned how to read the CPU temperature of your Raspberry Pi device and write the data to the console. The plan was to send this data to Azure IoT Hub, as there are many advantages to doing so. In this chapter, you will create an Azure IoT Hub and make your application send the temperature data to it.

Using Azure IoT Hub

Azure IoT Hub is an Azure service that helps you ingest telemetry data (for example, the CPU temperature we read from Raspberry Pi) from your IoT devices into the cloud to store and process. It acts as a central hub for bi-directional communication (from the device to the cloud and from the cloud to the device) between your IoT application and the managed devices.

Once the data is in the cloud, we can do many things with it. We will discuss these uses in the coming sections. Before you create an Azure resource, you should have a valid Azure subscription. If you don't have a subscription, no worries, you can always create one for free. All you have to do is go to <https://azure.microsoft.com/en-us/free/> and click the Start Free button. You will have to sign in with your Microsoft account or sign up for a new one.

Creating an Azure IoT Hub

Creating an Azure IoT Hub is as easy as drinking a glass of water. Lol, I mean it. There are two ways that you can create it.

- Using Azure Cloud Shell.
- Using Azure Portal.

We will discuss both of these options so that you can select the one that is more convenient for you.

Using Azure Cloud Shell

To create Azure services using a Cloud Shell, you must sign in to your Azure Portal (<https://portal.azure.com/>) and go to <https://shell.azure.com/>. You will be asked to select a directory if you have multiple directories in your Azure Portal. See Figure 5-1.

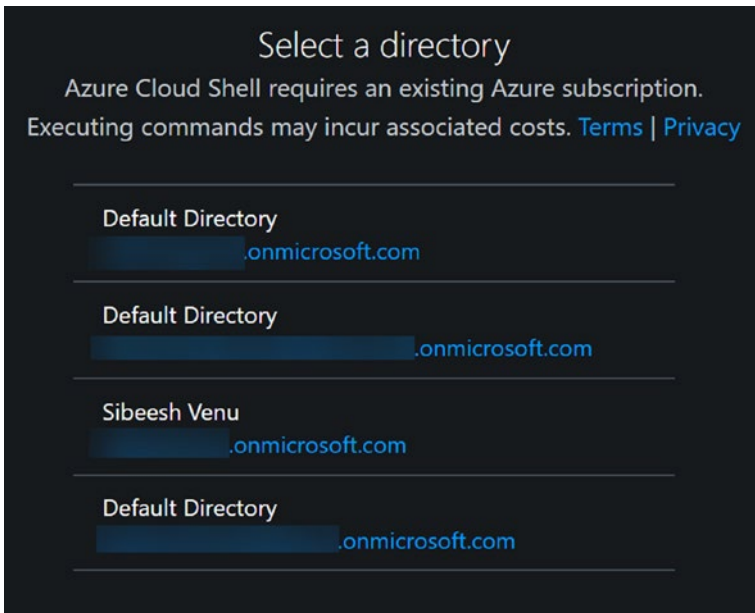


Figure 5-1. *Select a directory in the shell*

Clicking the directory will start the process. If you get the message saying that "No storage account mounted", you will have to create a storage account. If you are wondering why you need a storage account, it's used to persist the files of this shell. Note that creating a storage account will incur a small monthly cost. You can always see the pricing details at <https://azure.microsoft.com/en-us/pricing/details/storage/files/>.

The default subscription will be selected on the screen, but you can also change this. Click the Show Advanced Settings link, which is where you set the right resource group, storage, and so on. See Figure 5-2. You can use the existing resources or create a new resource. A resource group is just a collection of resources that share the same lifecycle, permissions, and policies.

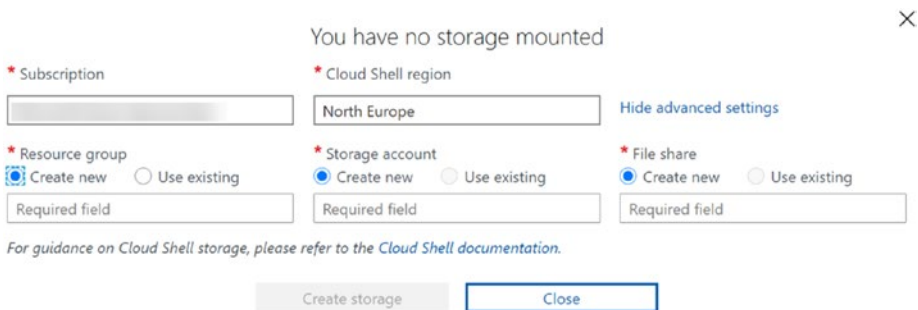


Figure 5-2. Cloud Shell advanced settings

Once you fill out the details, click the Create Storage button. This will start a new Cloud Shell for you, as shown in Figure 5-3.

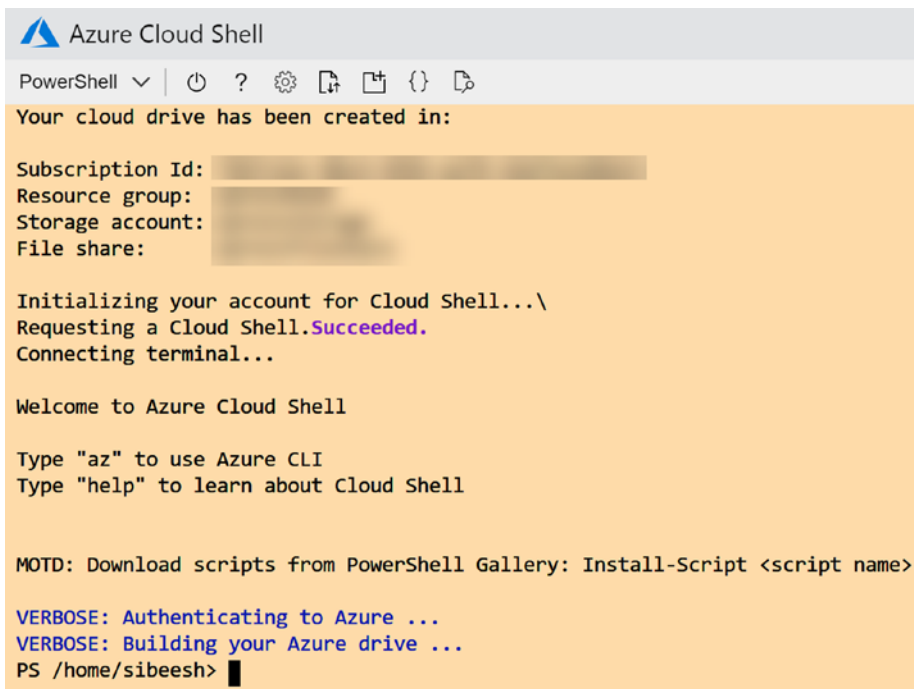


Figure 5-3. Azure Cloud Shell first login

Now it's time to run the command to create an IoT Hub. Paste the following command in the shell and press Enter. (Remember to change the resource group name to the one you chose in the previous section.)

```
az iot hub create --name apressiothub --resource-group
apressbook --sku S1
```

The command will run for a few minutes and a JSON will appear in the shell, with all the details of your IoT. You can also delete the IoT Hub with the following command.

```
az iot hub delete --name apressiothub --resource-group
apressbook
```

Using Azure Portal

Log in to Azure Portal(<https://portal.azure.com/>) and, from the Home page, click the +Create a Resource button. Now search for the keywords IoT Hub in the search box provided. Click the Create button on the next page (see Figure 5-4).

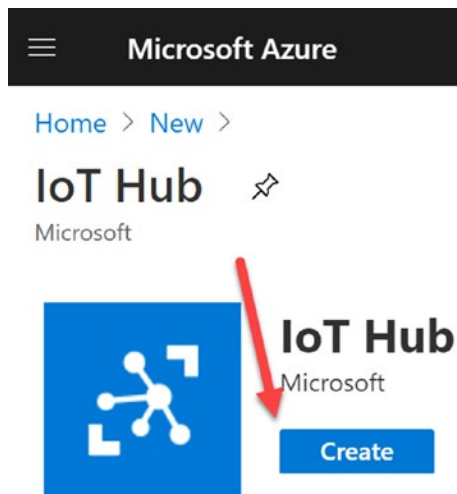


Figure 5-4. Creating IoT Hub

This will redirect you to the page where you can create IoT Hub. In the first step, you will be asked to select the subscription you have. You can either create a new resource group here or you can select the existing one. Next, select the region and give a valid name to your IoT Hub. (Keep in mind that you should select the region closest to you.) Boom, you did it! See Figure 5-5 for your reference.

[Home](#) > [New](#) > [IoT Hub](#) >

IoT hub

Microsoft

Basics Networking Size and scale Tags Review + create

Create an IoT hub to help you connect, monitor, and manage billions of your IoT assets. [Learn more](#)

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ	<input type="text"/>
Resource group * ⓘ	<input type="text" value="ApressBook"/> Create new
Region * ⓘ	<input type="text" value="North Europe"/>
IoT hub name * ⓘ	<input type="text" value="apressiothub"/>

Figure 5-5. *Creating IoT Hub, Step 1*

You can either click the Review + Create button to get to the last step or click the Next button. If you click the Next button, you will see an option for selecting how your IoT Hub should be connected, whether using public or private endpoints. See Figure 5-6.

[Home](#) > [New](#) > [IoT Hub](#) >

IoT hub

Microsoft

Basics Networking Size and scale Tags Review + create

Network connectivity

Connect to your IoT Hub using public or private endpoints.

Connectivity method * ⓘ

- Public endpoint (all networks)
- Public endpoint (selected IP ranges)
- Private endpoint
- i** All networks will have access to this IoT hub.
[Learn more about connectivity methods.](#)

Figure 5-6. *Creating IoT Hub, Step 2 Networking*

In the next step, you will learn how to size and scale IoT Hub (see Figure 5-7). You should be sure to select the right pricing and scale tier. There are some significant differences between them. For example, the standard tier enables all features, including bi-directional communication capabilities, whereas the basic tier provides only a subset of features and doesn't include bi-directional communication. It is also worth mentioning that, with the free tier, you can only send 8,000 messages per day and can have 500 devices connected. Each Azure subscription can create only one IoT Hub in the free tier. Table 5-1 shows the supported capabilities.

Table 5-1. *IoT Hub Tier Capabilities*

Capability	Basic Tier	Free/Standard Tier
Device-to-cloud telemetry	Yes	Yes
Per-device identity	Yes	Yes
Message routing, message enrichments, and event grid integration	Yes	Yes
HTTP, AMQP, and MQTT protocols	Yes	Yes
Device provisioning service	Yes	Yes
Monitoring and diagnostics	Yes	Yes
Cloud-to-device messaging		Yes
Device twins, module twins, and device management		Yes
Device streams (preview)		Yes
Azure IoT Edge		Yes
IoT plug-and-play preview		Yes

Home > New > IoT Hub >

IoT hub

Microsoft

Basics Networking Size and scale Tags Review + create

Each IoT hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

Scale tier and units

Pricing and scale tier * ⓘ

S1: Standard tier

[Learn how to choose the right IoT hub tier for your solution](#)

Number of S1 IoT hub units ⓘ



1

Determines how your IoT hub can scale. You can change this later if your needs increase.

Azure Security Center

 OnTurn on Azure Security Center for IoT and add an extra layer of threat protection to IoT Hub, IoT Edge, and your devices. [Learn more](#)

Pricing and scale tier ⓘ	S1	Device-to-cloud-messages ⓘ	Enabled
Messages per day ⓘ	400,000	Message routing ⓘ	Enabled
Cost per month	1652.41 INR	Cloud-to-device commands ⓘ	Enabled
Azure Security Center ⓘ	0.06609625 INR per device per month	IoT Edge ⓘ	Enabled
		Device management ⓘ	Enabled

Figure 5-7. *Creating IoT Hub, Step 3 Size and Scale*

The IoT Hub Units is the number of messages allowed per unit, per day. This depends on the selection of the pricing tier. With one S1 IoT Hub unit, you can send 400,000 messages per day. If you wish to send more, you can add an S1 IoT Hub unit, which will give you another 400,000 messages. Keep in mind that the price will also increase as you choose the units. If you choose free tier, this option will not be enabled.

The Azure Security Center is an extra layer of threat protection and security. This is not available on the free tier.

The Device-to-Cloud-Partitions under the Advanced settings relate the device-to-cloud message to the number of simultaneous readers of the messages.

Once you select the options you want, click the Next button, which will give you an option to add tags, as shown in Figure 5-8.

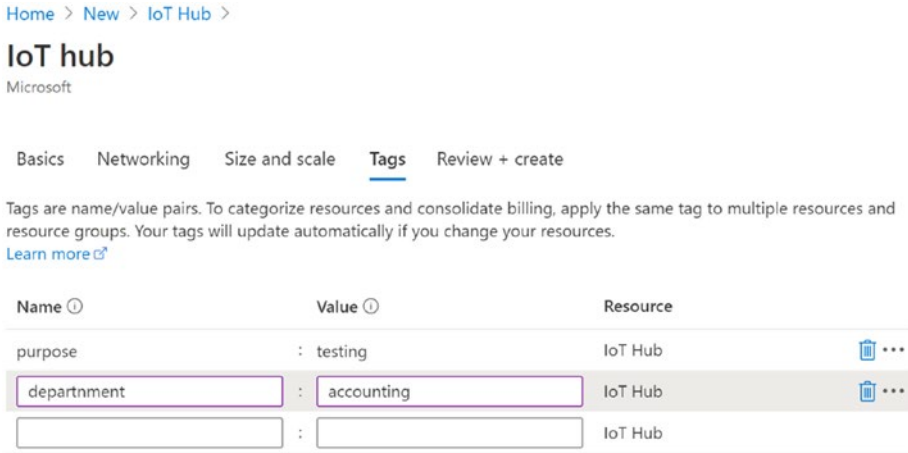


Figure 5-8. *Creating IoT Hub, Step 4 Tags*

Tags are the name/value pairs that categorize the resources and resource groups and consolidate in billing. You can apply the same tags to multiple resources and resource groups. Clicking the Next button will give you a screen with the values you have selected. See Figure 5-9.

[Home](#) > [New](#) > [IoT Hub](#) >

IoT hub

Microsoft

[Basics](#) [Networking](#) [Size and scale](#) [Tags](#) [Review + create](#)

Basics

Subscription	
Resource group	ApressBook
Region	North Europe
IoT hub name	apressiothub

Networking

Connectivity method	Public endpoint (all networks)
IP filter rules	None
Private endpoint connections	None

Size and scale

Pricing and scale tier	S1
Number of S1 IoT hub units	1
Messages per day	400,000
Device-to-cloud partitions	2
Cost per month	1652.41 INR
Azure Security Center	See the Azure Security Center pricing

Tags

purpose	testing
department	accounting

[Create](#)[< Previous: Tags](#)[Next >](#)[Automation options](#)

Figure 5-9. Creating IoT Hub, Step 5 Review and Create

Now click the Create button, which will initialize your IoT Hub. After a while, your resource will be ready for action.

Registering a Device in the IoT Hub

A device must be registered in your IoT Hub before it can connect. To create a device, go to your IoT Hub and then go to the IoT Devices section, as shown in Figure 5-10.

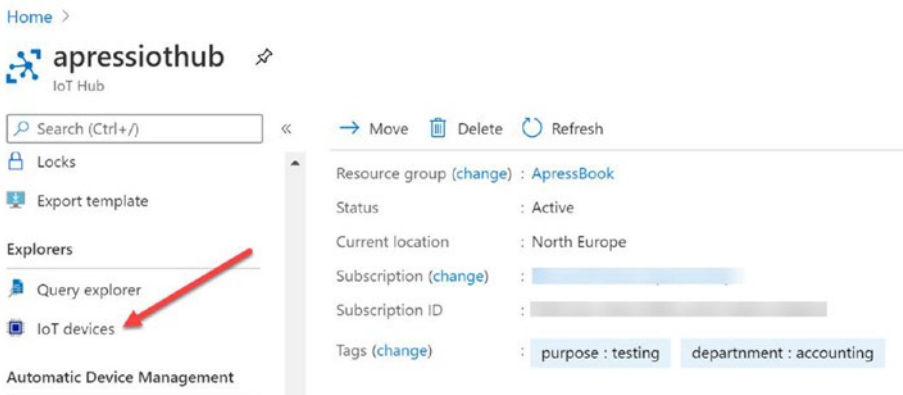





Figure 5-10. *The IoT Devices menu*

Now click the +New button on the top, which will open a new page where you can register a device. See Figure 5-11.

[Home](#) > [apressiothub | IoT devices](#) >

Create a device

 Find Certified for Azure IoT devices in the Device Catalog 

Device ID * 


 

Figure 5-11. Create a device

Here, the Device ID is the name of your device. It's used for device authentication and access control. Clicking the Save button will create a device and the page will be redirected to the device list.

Connecting Raspberry Pi to Azure IoT Hub

To connect your Raspberry Pi to Azure IoT Hub, you must add a package named `Microsoft.Azure.Devices.Client` to the solution. You can add this by running the following command.

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

If you check your `.csproj` file now, you should see that the package reference entry is been added there.

```
<PackageReference Include="Microsoft.Azure.Devices.Client"  
Version="1.27.0" />
```

You can also add an entry here if you don't link to run the previous command. It works both ways.

Let's create a model class for our telemetry data now. We'll call this class `DeviceData`, but feel free to give it any name you wish. We will have three properties in that class for now. Install the `Newtonsoft.Json` package in your application so that you can easily serialize and deserialize your data. Run the following command:

```
dotnet add package Newtonsoft.Json
```

Here is how your `DeveiceData` model class should look at this point.

```
using Newtonsoft.Json;  
namespace raspberrypi.net.core.Models  
{  
    public class DeviceData  
    {
```



```

    [JsonProperty(PropertyName="temperature")]
    public double Temperature { get; set; } = 0;
    [JsonProperty(PropertyName="messageid")]
    public int MessageId { get; set; } = 0;
    [JsonProperty(PropertyName="deviceid")]
    public string DeviceId {get;set;} = Program.DeviceId;
}
}

```

Now let's rewrite the program, as follows.

```

using System;
using System.Text;
using Iot.Device.CpuTemperature;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
using raspberrypi.net.core.Models;

namespace raspberrypi.net.core
{
    class Program
    {
        private static CpuTemperature _rpiCpuTemp = new
        CpuTemperature();
        private const string _deviceConnectionString = "";
        private static int _messageId = 0;
        private static DeviceClient _deviceClient =
        DeviceClient.CreateFromConnectionString(_
        deviceConnectionString, TransportType.Mqtt);
        public const string DeviceId = "";
    }
}

```

```
static async Task Main(string[] args)
{
    while (true)
    {
        if (_rpiCpuTemp.IsAvailable)
        {
            await SendToIoTHub(_rpiCpuTemp.Temperature.
                Celsius);
            Console.WriteLine("The device data has been
                sent");
        }
        Thread.Sleep(5000); // Sleep for 5 seconds
    }
}

private static async Task SendToIoTHub(double celsius)
{
    string jsonData = JsonConvert.SerializeObject(new
        DeviceData()
        {
            MessageId = _messageId++,
            Temperature = celsius
        });
    var messageToSend = new Message(Encoding.UTF8.
        GetBytes(jsonData));
    await _deviceClient.SendEventAsync(messageToSend).
        ConfigureAwait(false);
}
}
```

The `SentToIoTHub` method is responsible for sending the `Message` data to Azure IoT Hub by using the `SendEventAsync` function in the `DeviceClient`. Note that this `DeviceClient` is part of the `Microsoft.Azure.Devices.Client` namespace, so make sure to add it to the using statements.

Now we just need to update the connection string and device ID from IoT Hub. Go to your IoT Hub resource and click the IoT Devices menu under the Explorers section. You will see your device listed on the page, as shown in Figure 5-12.

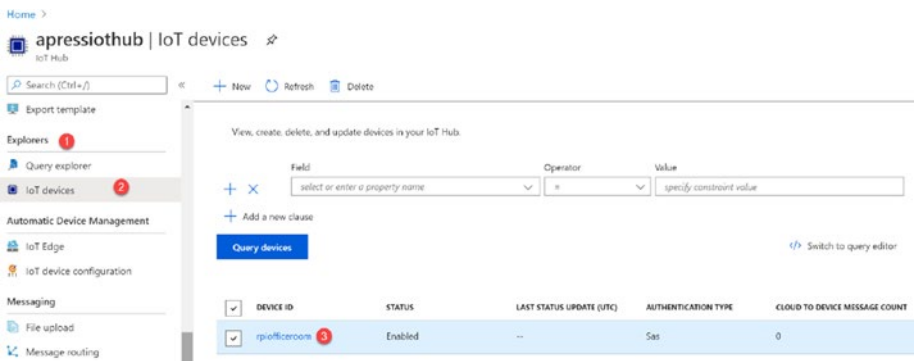


Figure 5-12. IoT Devices list

Click the device name to see a page with all the information about that device, as shown in Figure 5-13.

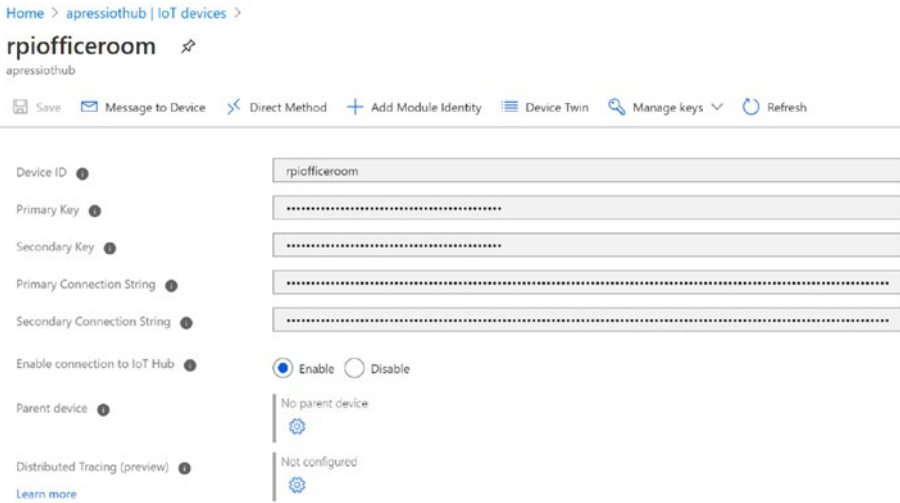


Figure 5-13. *IoT Device properties*

From those properties, you need both the Primary Connection String and the Device ID. Your connection string will look like this:

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

Now that you have updated the program with the connection string and device ID, all you have to do is run the build task. Do you remember how to do that? Just press F5 and make sure that you select the Rpi Publish and Debug task.

The new program will be redeployed to Raspberry Pi, will attach the debugger, and will run the application. If you still have the debugger, it will hit any issues and you can see the values. If there is no debugger attached, just double-click the left side of any line, See Figure 5-14.

```

20 |         static async Task Main(string[] args)
21 |         {
22 |             while (true)
23 |             {
24 |                 if (_rpiCpuTemp.IsAvailable)
25 |                 {
26 |                     await SendToIoTHub(_rpiCpuTemp.Temperature.Celsius);
27 |                     Console.WriteLine("The device data has been sent");
28 |                 }
29 |                 Thread.Sleep(5000); // Sleep for 5 seconds
30 |             }
31 |         }

```

Figure 5-14. Sending data to the IoT Hub Debug screen

Monitoring the Device Data and IoT Hub

To monitor data communication, Microsoft introduced an amazing extension called Azure IoT Tools. There are many things that you can do with this extension. First, let's install it. See Figure 5-15.

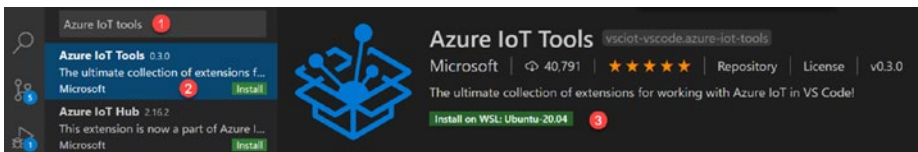


Figure 5-15. Install Azure IoT Tools

Once you install the extension, you have to restart your VSCode to load it. You should see a page like Figure 5-16 open once you have done that.

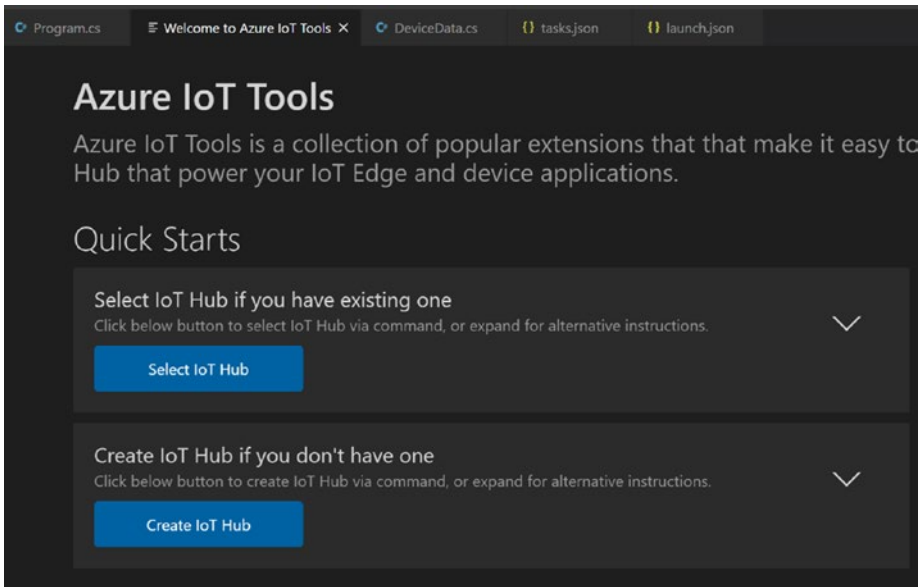


Figure 5-16. *Azure IoT Tools first page*

Click the Select IoT Hub button and log in with your account. You will see all of your subscriptions listed. Now select the Azure subscription where you created your IoT Hub, and then select IoT Hub. It is as simple as that. You should see your IoT Hub device in the tool now. See Figure 5-17.

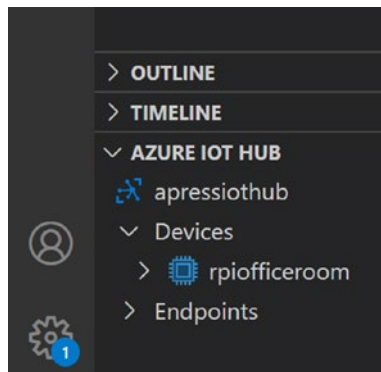


Figure 5-17. *IoT Hub Devices list*

You can also select the IoT Hub from the Azure IoT Hub menu, as shown in Figure 5-18.

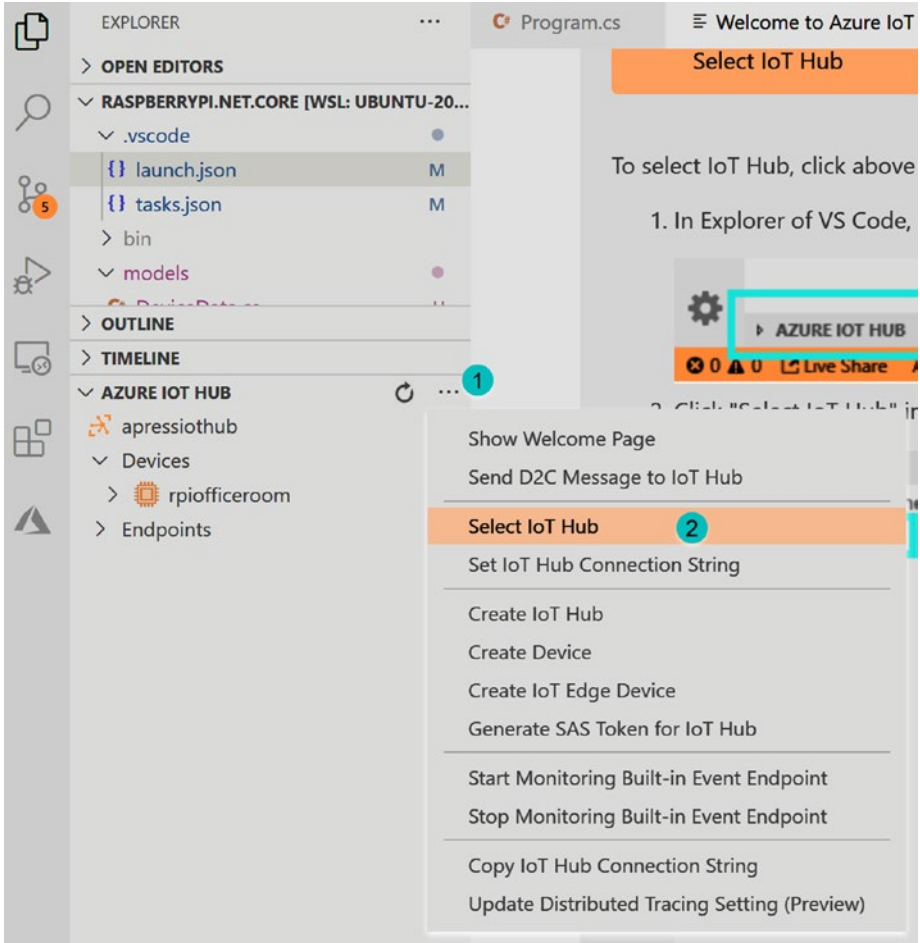


Figure 5-18. Select IoT Hub

You can easily interact with the device using this tool. Some of the options are given here:

- Send D2C messages to IoT Hub
- Send C2D messages to the device
- Start monitoring

For now, we can go ahead and start monitoring our device. What do you think? To do that, you just right-click the device name and choose the Start Monitoring Built-in Event Endpoint menu item (see Figure 5-19).

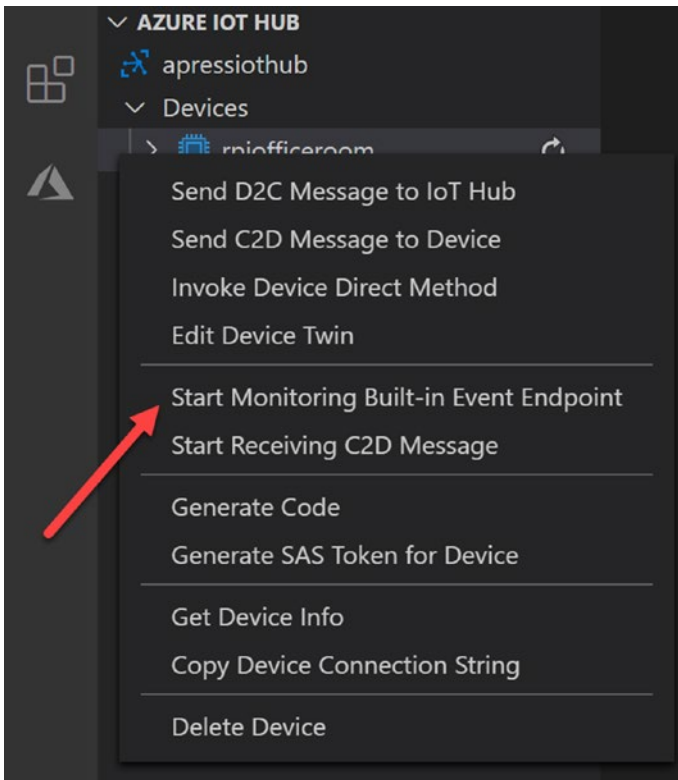


Figure 5-19. Azure IoT Hub Tool options

You should see the communication in the output window, as shown in Figure 5-20.


```

Subscription selected:
IoT Hub selected: apressiothub
[IoTHubMonitor] Start monitoring message arrived in built-in endpoint for device [rpiofficerroom] ...
[IoTHubMonitor] Created partition receiver [0] for consumerGroup [$Default]
[IoTHubMonitor] Created partition receiver [1] for consumerGroup [$Default]
[IoTHubMonitor] [12:10:37 PM] Message received from [rpiofficerroom]:
{
  "temperature": 54.042999267578125,
  "messageid": 0,
  "deviceid": "rpiofficerroom"
}

```

Figure 5-20. IoT Hub device monitoring

Adding Custom Event Message Properties

You can also add a custom event message property while you send the data to the Azure IoT Hub. Let's see how you do it. Update your `SendToIoTHub` function as follows:

```

private static async Task SendToIoTHub(double tempCelsius)
{
    string jsonData = JsonConvert.SerializeObject(new
        DeviceData()
        {
            MessageId = _messageId++,
            Temperature = tempCelsius
        });
    var messageToSend = new Message(Encoding.UTF8.
        GetBytes(jsonData));
    messageToSend.Properties.Add("TemperatureAlert",
        (tempCelsius > _temperatureThreshold) ? "true" :
        "false");
    await _deviceClient.SendEventAsync(messageToSend).
        ConfigureAwait(false);
}

```

Make sure to add a new variable called `_temperatureThreshold`.

```
private const double _temperatureThreshold = 40;
```

Now press F5 to see the output. You should see the JSON data, as shown in Figure 5-21.

The image shows a screenshot of a Visual Studio Output window. The window has a dark header with four tabs: "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", and "TERMINAL". The "OUTPUT" tab is selected. The output text is as follows:

```
[IoTHubMonitor] [2:38:25 PM] Message received from [rpiofficeroom]:  
{  
  "body": {  
    "temperature": 54.042999267578125,  
    "messageid": 0,  
    "deviceid": "rpiofficeroom"  
  },  
  "applicationProperties": {  
    "TemperatureAlert": "true"  
  }  
}
```

Figure 5-21. *Temperature alert*

Wow, isn't that cool?. I hope that you enjoyed playing with IoT Hub. There are many things that we need to do with it in subsequent chapters.

Summary

In this chapter, I hope you have learned the following:

- What Azure IoT Hub is?
- How to create Azure IoT Hub using the Azure Cloud Shell?
- How to create Azure IoT Hub using Azure Portal?
- How to connect Azure IoT Hub from your .NET Core application?
- How to monitor Azure IoT Hub communication using Azure's IoT tools?

Let's keep playing with IoT Hub. I will see you in the next chapter.

CHAPTER 6

Finally, A Windows Terminal That You Can Customize

In the last few chapters, you worked with your application and the IoT hub. In this chapter, I want to give you pro tip—using the Windows Terminal command-line tool, which you can customize. Yeah, you heard me right. In this chapter, I show how you can use and customize that tool. If you are not interested in using this tool, feel free to skip this chapter.

Using Windows Terminal

Windows Terminal is a new, fast, modern, efficient terminal application for users of command-line tools and shells, like command-prompt, PowerShell, and WSL. You can easily download the terminal from Microsoft Store, as shown in Figure 6-1.

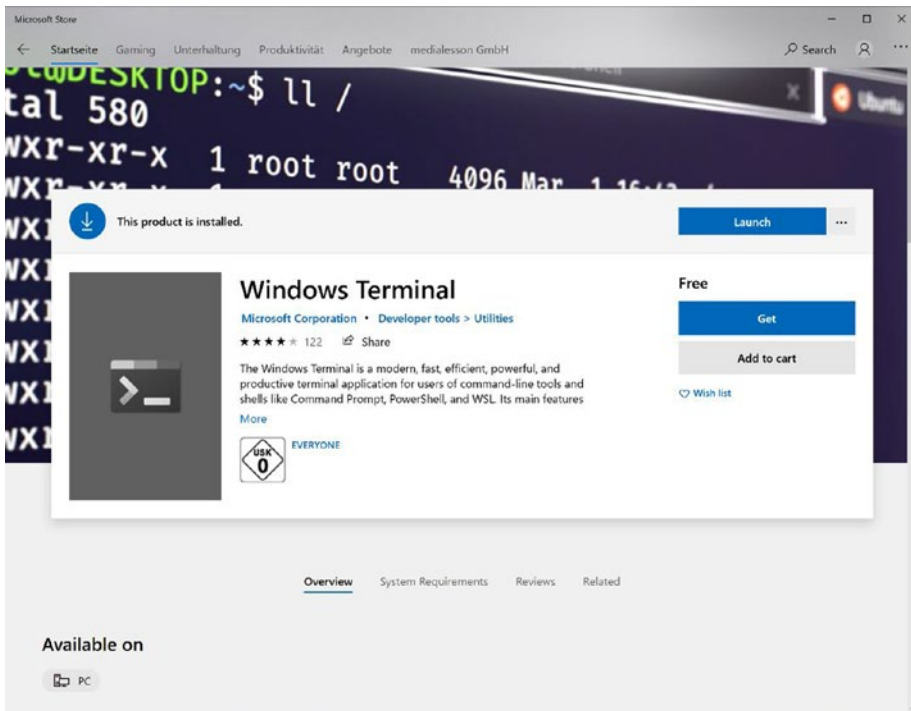


Figure 6-1. Windows Terminal install

Windows Terminal Key Features

Once you open Windows Terminal, you can see the fresh look of this new command-line tool. It has an updated UI and many new features, some of which we discuss next.

Support for Multiple Tabs

The new terminal supports multiple tabs, each connected to a command-line shell or app of your choice. See Figure 6-2.

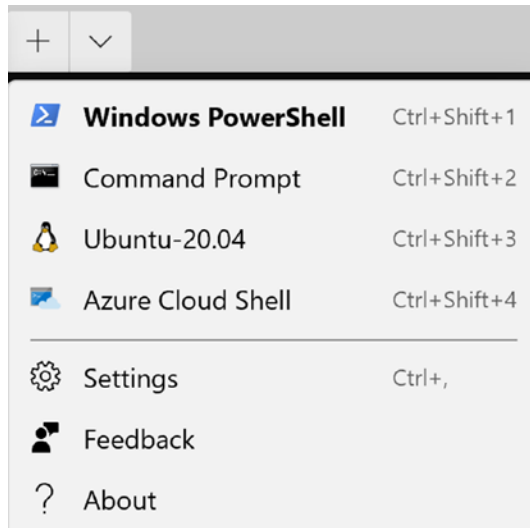


Figure 6-2. Windows Terminal's multiple tabs

Support for Emojis, Icons, and More

With the new terminal, you can display text characters, glyphs, and symbols present on your windows, including emojis, powerline symbols, icons, and more.

Configuring Windows Terminal

The new terminal gives you the option to customize your terminal, including:

- Multiple profiles for each shell/app/tools you use.
- Separate font styles, color themes, backgrounds, and transparency levels for each profile.

The configuration is stored in a structured text file so that anyone can easily edit it. To edit the settings file (called `settings.json`), just click the Settings button, as shown in Figure 6-3.

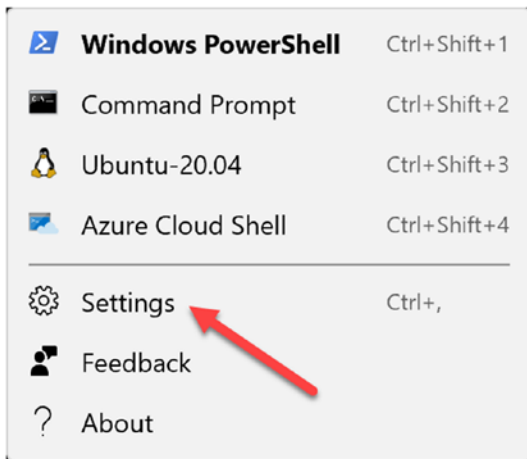


Figure 6-3. *Windows Terminal settings*

The `settings.json` file will open in your default code editor. You should see the profiles section with possible names Windows PowerShell, Command Prompt, Ubuntu-20.04, or Azure Cloud Shell. You can edit these profiles as you wish; for example, you could edit the Ubuntu-20.04 profile as follows:

```
{
    "guid": "{07b52e3e-de2c-5db4-bd2d-ba144ed6c273}",
    "hidden": false,
    "name": "Ubuntu-20.04",
    "source": "Windows.Terminal.Wsl",
    "background": "#fff",
    "startingDirectory": "\\wsl$\Ubuntu-20.04\home\sibeeshvenu",
    "colorScheme": "Campbell"
}
```

Now if you open the Ubuntu-20.04 shell, you'll see that all the settings are updated. Note that the starting directory has changed as well.

Windows Terminal Preview Version

There is also a preview version of Windows Terminal, planned to be released in July of 2020. This version offers many other features. Let's look at them now.

Open Folders in Windows Terminal

You can right-click any folder and select Open in Windows Terminal, which will launch Windows Terminal with your default profile in the directory you selected from File Explorer.

Font Weight Support

The preview version supports font weight as a new profile setting. The possible values of the `fontWeight` property are `normal`, `thin`, `extra-light`, `light`, `semi-light`, `medium`, `semi-bold`, `bold`, `extra-bold`, `black`, `extra-black`, or an integer corresponding to the numeric representation of the OpenType font weight. You place the values in quotes, as follows:

```
"fontWeight": "normal"
```

Support to Open a Profile as a Pane

If you want to open a profile as a pane in the current window, all you have to do is press and hold the Alt key and then click the profile. This will open the profile as a pane by using the auto-split feature. See Figure 6-4.

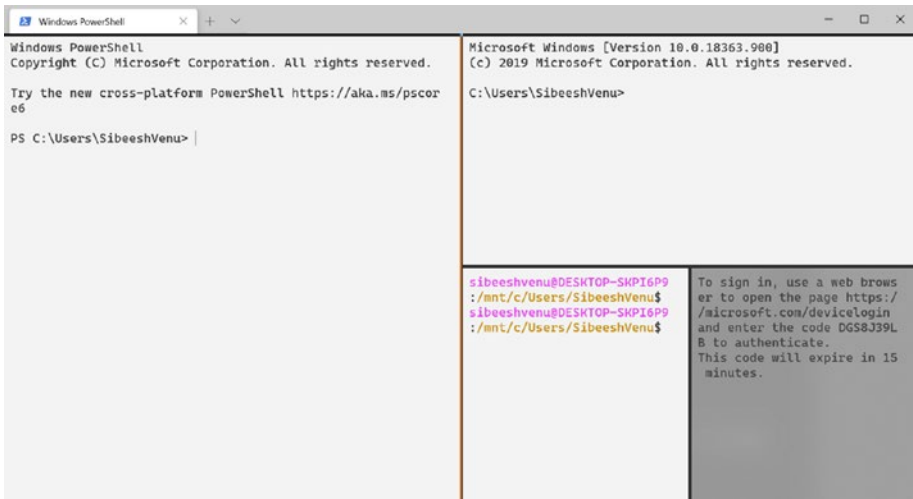


Figure 6-4. Windows Terminal's auto-split pane

Change the Tab Color

To change the color of a tab, just right-click the tab and select Color, which will open a color menu. Then you can select the color you wish. See Figure 6-5.

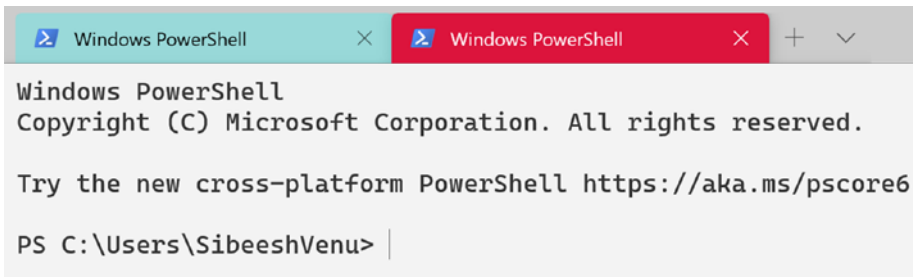


Figure 6-5. Windows Terminal tab colors

Rename a Tab

There is also an option to rename a tab. To do that, just right-click the tab and select Rename Tab. This will change your tab title to a textbox, where you can rename the tab for that terminal session.

Summary

Isn't it cool that you can design your terminal? So, in this chapter, you learned:

- What Windows Terminal is?
- How to install Windows Terminal?
- The key features of Windows Terminal.
- How to customize Windows Terminal?

CHAPTER 7

Cloud to Device Communication

In the last few chapters, we sent data from the device to the Azure IoT Hub and monitored the data using Azure IoT Tools. Now it's time to discuss sending messages or data from the cloud to the device. You might need to do this in the following scenarios:

- If the temperature is more than the threshold value, you may need to make sure that the room temperature is balanced. How do you do that automatically? You can send instructions from the cloud to the device to turn on the fan and air conditioner.
- Imagine that you need to install a new module or firm update, perhaps on millions of devices. It is impossible to manually do this on each device. Instead, you can send the instructions from the cloud to all the devices so that each device can be updated.
- Imagine that your camera-integrated IoT device needs to send a photo of the room when some conditions are met. To do this, you simply send the instructions from the Hub and the device will listen to it.

Now that you know why it's useful, let's see how to send data from the cloud to the device.

Cloud-to-Device Communication Options

The IoT Hub provides three ways to communicate to the device from the cloud:

- Direct methods.
- Twin's desired properties.
- Cloud-to-device messages.

Let's go through each of these methods.

Direct Methods

With these methods, you invoke direct functions on the device from the cloud. You can use this approach if you require an immediate confirmation of the result. A few of the examples are given here.

- To turn on lights or fans.
- To give an alert or a warning to the user.
- To take a photo of an intruder.
- To turn on a motor when the water level of a tank is low.

Direct methods represent a request-reply interaction with the device, similar to an HTTP call in that they succeed or fail immediately. To configure the direct method, the first thing we should do is create a handler for the method.

Rewrite the Main method as follows.

```
static async Task Main(string[] args)
{
    _deviceClient = DeviceClient.
    CreateFromConnectionString(_deviceConnectionString,
    TransportType.Mqtt);
    // Create a handler for the direct method call
    _deviceClient.SetMethodHandlerAsync(methodName,
    TurnOnLight, null).Wait();

    while (true)
    {
        if (_rpiCpuTemp.IsAvailable)
        {
            await SendToIoTHub(_rpiCpuTemp.Temperature.
            Celsius);
            Console.WriteLine("The device data has been
            sent");
        }
        Thread.Sleep(5000); // Sleep for 5 seconds
    }
}
```

Here, the method name is TurnOnLight.

```
private const string methodName = "TurnOnLight";
```

You can see that we are using the SetMethodHandlerAsync method to create the handler. Now let's write the custom handler as follows:

```
private static Task<MethodResponse> TurnOnLight(MethodRequest
methodRequest, object userContext)
{
    Console.WriteLine("Here is the call from cloud to
    turn of the light!");
}
```

```

var result = "{\"result\": \"Executed direct method:
" + methodRequest.Name + "\"}";
return Task.FromResult(new MethodResponse(Encoding.
UTF8.GetBytes(result), 200));
}

```

The idea here is that the handler we registered will invoke this task method. As we are using the IoT Hub Tool already, there is an easy way to call this direct method. Just right-click the device name and click the Invoke Device Direct Method menu option, as shown in Figure 7-1.

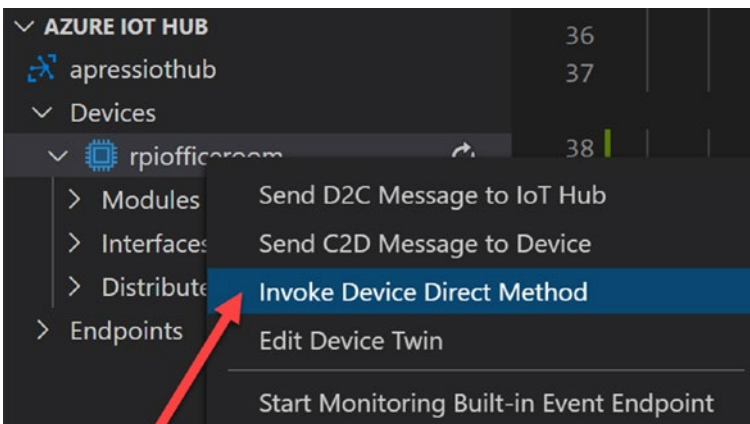
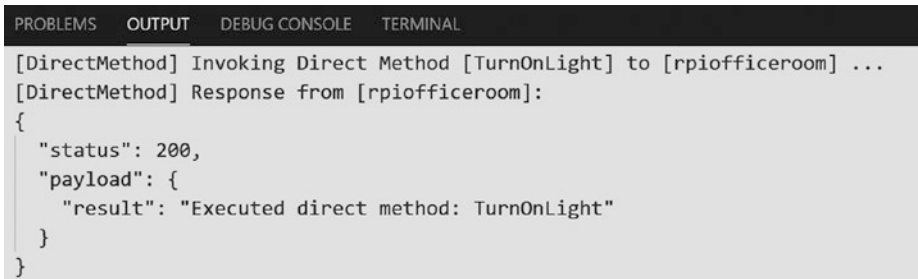


Figure 7-1. *Invoking the device direct method*

This will give you a textbox where you can enter the method name and payload (optional). Don't forget to enter the same method name that you used in the handler. Once you give that information, press Enter. The TurnOnLight method will be called and you will get output similar to what's shown in Figure 7-2.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[DirectMethod] Invoking Direct Method [TurnOnLight] to [rpiofficeroom] ...
[DirectMethod] Response from [rpiofficeroom]:
{
  "status": 200,
  "payload": {
    "result": "Executed direct method: TurnOnLight"
  }
}

```

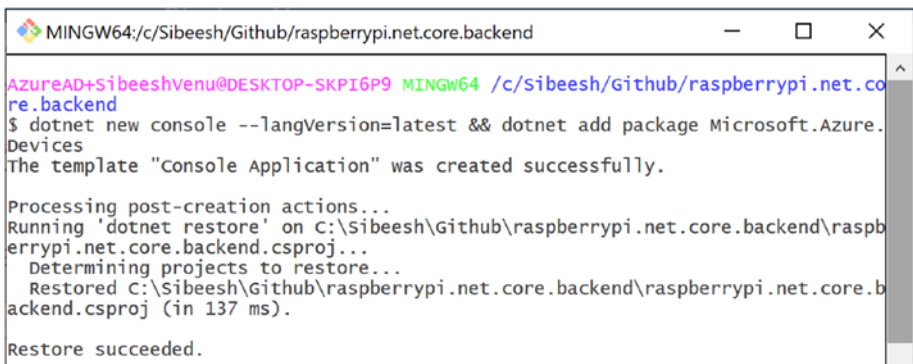
Figure 7-2. Executed direct method

You can also create another application, where you trigger this action instead of using the Azure IoT Tool.

Creating a Backend Application To Call the Direct Method

Let's create a new folder, called `raspberrypi.net.core.backend`, and run the following command to create a new console application. See Figure 7-3.

```
dotnet new console --langVersion=latest && dotnet add package Microsoft.Azure.Devices
```



```

MINGW64:/c:/Sibeesh/Github/raspberrypi.net.core.backend
AzureAD+SibeeshVenu@DESKTOP-SKPI6P9 MINGW64 /c:/sibeesh/Github/raspberrypi.net.core.backend
$ dotnet new console --langVersion=latest && dotnet add package Microsoft.Azure.Devices
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Sibeesh\Github\raspberrypi.net.core.backend\raspberrypi.net.core.backend.csproj...
  Determining projects to restore...
  Restored C:\Sibeesh\Github\raspberrypi.net.core.backend\raspberrypi.net.core.backend.csproj (in 137 ms).

Restore succeeded.

```

Figure 7-3. Creating a backend solution

Once the application is created, open it in a separate VSCode. We will rewrite the Program.cs file as follows.

```
using System;
using Microsoft.Azure.Devices;
using System.Threading.Tasks;

namespace raspberrypi.net.core.backend
{
    class Program
    {
        private static ServiceClient _serviceClient;
        private const string _deviceId = "rpiofficeroom";
        private const string methodName = "TurnOnLight";
        private const string _deviceConnectionString = "HostName
=apressiothub.azure-devices.net;SharedAccessKeyName=
service;SharedAccessKey=i2uJ9US+JaJQMDAGcwIkTYJ95JWDC7PT
004zyCAW8dQ=";
        static async Task Main(string[] args)
        {
            _serviceClient = ServiceClient.CreateFrom
                ConnectionString(_deviceConnectionString);
            await InvokeDirectMethod(methodName);
            Console.WriteLine("Hello World!");
        }

        private static async Task InvokeDirectMethod(string
methodName)
        {
            var invocation = new CloudToDeviceMethod(methodName)
            {
                ResponseTimeout = TimeSpan.FromSeconds(45)
            };
        }
    }
}
```

```

        invocation.SetPayloadJson("5");
        var response = await _serviceClient.
        InvokeDeviceMethodAsync(_deviceId, invocation);
        Console.WriteLine(response.GetPayloadAsJson());
    }
}
}
}

```

Note that the connection string we are using here is the service connection string, not the device connection string. To get the service connection string, go to your IoT Hub and then choose the Shared Access Policies menu option. See Figure 7-4.

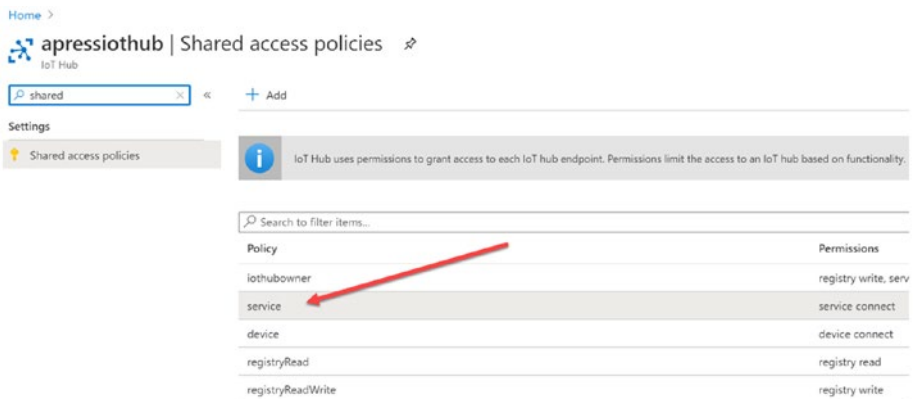
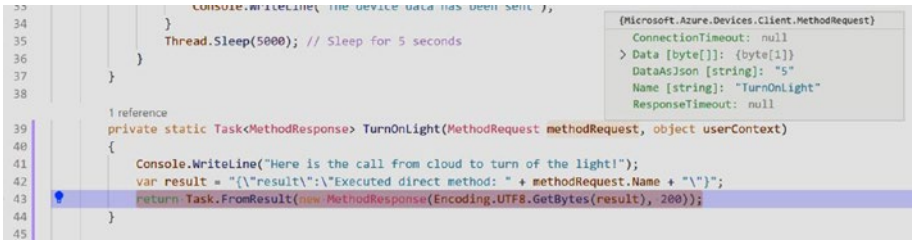


Figure 7-4. Service connection string

All we are doing is creating a new service client from the IoT Hub service connection string and then invoking the device method we configured. Now, let's debug both of the applications. Remember to run the first application (`raspberrypi.net.core`) in WSL.

Once we run the first application, we can put the debugger on the `TurnOnLight` method. Then we run our second application (`raspberrypi.net.core.backend`).

This second application will call the direct method; the first application will trigger the method via the handler and then return the response. The second application receives the response and shows it in the console. As you can see in Figure 7-5, the payload is also being received in the first application.



The screenshot shows a C# code editor with a debugger window. The code includes a `TurnOnLight` method that receives a `MethodRequest` and returns a `MethodResponse`. The debugger window shows the state of the `MethodRequest` object, including its `Name` property set to "TurnOnLight".

```
33 Console.WriteLine("The device data has been sent.");
34 }
35 Thread.Sleep(5000); // Sleep for 5 seconds
36 }
37 }
38
39 1 reference
40 private static Task<MethodResponse> TurnOnLight(MethodRequest methodRequest, object userContext)
41 {
42     Console.WriteLine("Here is the call from cloud to turn of the light!");
43     var result = $"{result}:\Executed direct method: " + methodRequest.Name + "\n";
44     return Task.FromResult(new MethodResponse(Encoding.UTF8.GetBytes(result), 200));
45 }
```

```
{Microsoft.Azure.Devices.Client.MethodRequest}
ConnectionTimeout: null
> Data [byte[]]: {byte[1]}
DataAsJson [string]: "5"
Name [string]: "TurnOnLight"
ResponseTimeout: null
```

Figure 7-5. Trigger Direct method debug

Figure 7-6 shows the console of the second application.

```

1 reference
20 private static async Task InvokeDirectMethod(string
21 {
22     var invocation = new CloudToDeviceMethod(methodName)
23     {
24         ResponseTimeout = TimeSpan.FromSeconds(45)
25     };
26     invocation.SetPayloadJson("5");
27     var response = await _serviceClient.InvokeDevice
28     Console.WriteLine(response.GetPayloadAsJson());
29 }
30 }
31 }
32 }
33 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

ger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.
'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.
on 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.
tion 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.
Debugger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.
Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.
r option 'Just My Code' is enabled.
{"result": "Executed direct method: TurnOnLight"}

```

Figure 7-6. Response from the direct method call

Twin's Desired Properties

This type of cloud-to-device communication enables long-running commands to put the device into a certain desired state. For example, to change the send telemetry interval. The device twins are a JSON document where the device information, such as metadata, configuration, and conditions, is saved. The Azure IoT Hub maintains a device twin for each device that you connect to IoT Hub. As we are using Azure IoT Tools already, we can easily see this file within our VSCode. Right-click your device name and select Edit Device Twin, as shown in Figure 7-7.

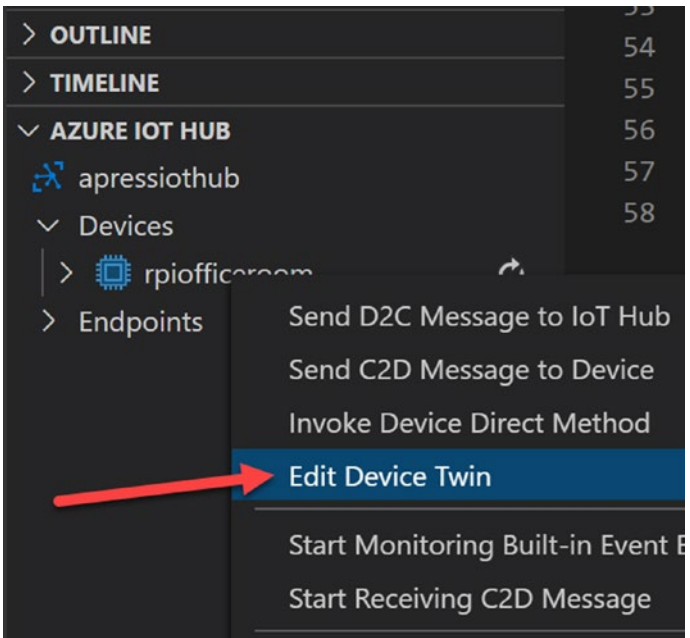


Figure 7-7. Edit device twin

A new JSON file with the name `azure-iot-device-twin.json` will be loaded and the contents of that file will look as follows:

```
{
  "deviceId": "rpiofficeroom",
  "etag": "AAAAAAAAAAE=",
  "deviceEtag": "NzgyOTM1NDc2",
  "status": "enabled",
  "statusUpdateTime": "0001-01-01T00:00:00Z",
  "connectionState": "Disconnected",
  "lastActivityTime": "2020-07-17T12:36:24.4328341Z",
  "cloudToDeviceMessageCount": 0,
  "authenticationType": "sas",
  "x509Thumbprint": {
    "primaryThumbprint": null,
    "secondaryThumbprint": null
  },
  "version": 2,
  "properties": {
    "desired": {
      "$metadata": {
        "$lastUpdated": "2020-07-14T10:47:29.8590777Z"
      },
      "$version": 1
    },
    "reported": {
      "$metadata": {
        "$lastUpdated": "2020-07-14T10:47:29.8590777Z"
      },
      "$version": 1
    }
  }
},
```

```
"capabilities": {  
  "iotEdge": false  
},  
"tags": {}  
}
```

You can use device twins to:

- Store device-specific locations, for example, the location of your Raspberry Pi.
- Report current state information about the device.
- Query your device configuration, state, or metadata.
- Synchronize the state of long-running workflows between the backend app and the device app. For example, when the backend app performs a firmware update to be installed on the device and the device app reports the stages of the update process.

You can also see this device twin JSON data in your Azure portal. Click the Query Explorer, and then select Device Twin from the Collections drop-down. Then click the Run button. Note that the default query is:

```
SELECT * FROM c
```

Figure 7-8 is for your reference.

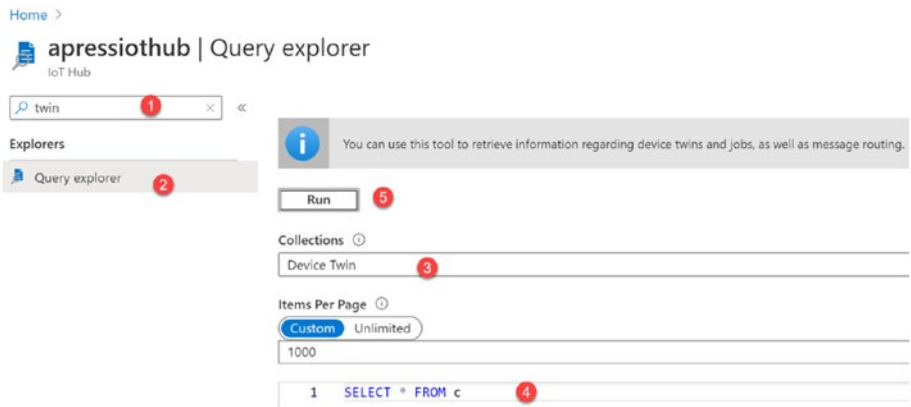


Figure 7-8. Device twin in the Azure Portal

Now let's discuss some of the properties of the device twins.

Reported Property

The reported property can be used in such scenarios as when the solution backend needs to know the last known value of a property. Here's an example of the reported property. Let's have a look.

```
"reported": {
  "telemetryConfig": {
    "sendFrequency": "5m",
    "status": "success"
  },
  "batteryLevel": 55,
  "$metadata" : {...},
  "$version": 4
}
```

Here, the `batteryLevel` property is the last battery level reported by the device app.

Desired Property

Here is an example of the desired property:

```
"desired": {
    "telemetryConfig": {
        "sendFrequency": "5m"
    },
    "$metadata" : {...},
    "$version": 1
}
```

The backend solution sets the desired property with the desired configuration values so that the device application can read it. For example, in the previous codeblock, `telemetryConfig` is the desired property. If the device is already connected, the changes will take effect immediately, if not, at the first reconnect. The device app can report the status in the reported property, as in the following codeblock.

```
"reported": {
    "telemetryConfig": {
        "sendFrequency": "5m",
        "status": "success"
    },
    "batteryLevel": 55,
    "$metadata" : {...},
    "$version": 4
}
```

Check the status property. I hope this is enough of an introduction to device twins. Let's see them in action. We will be updating the backend application that we created earlier, but before we do that, let's get a new connection string from the IoT Hub. Recall that the connection string we used earlier only has the Service Connect permission.

To work with this, we need a connection string that has both Service Connect and Registry Read connections. To change this, go to the Shared Access Policy section of your IoT Hub and click the +Add button. See Figure 7-9.

Add a shared access policy ×

apressiothub

Access policy name *

serviceRegistryRead ✓

Permissions

- Registry read ⓘ
- Registry write ⓘ
- Service connect ⓘ
- Device connect ⓘ

Figure 7-9. Service connect and registry read permissions

Once you are done, click the Create button. This will create a new policy. Click the policy and get the primary connection string, as shown in Figure 7-10.

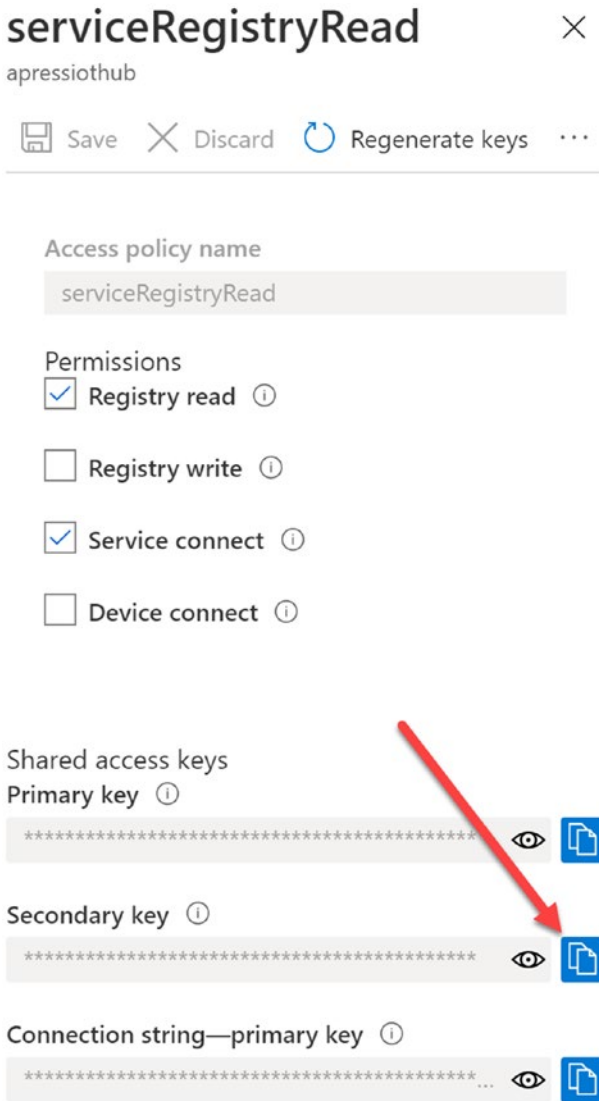


Figure 7-10. The connection string with service connect and registry read permissions

Replace the old connection string with the new one in the backend application. Note that you can also create a new console application with the new connection string, if you don't want to update this connection string and mess with the old application.

Now add a new property to the Registry Manager and initialize it.

```
private static RegistryManager _registryManager;
_registryManager = RegistryManager.CreateFromConnectionString
(_deviceConnectionString);
```

Let's now create a new function with the properties we need to update.

```
private static async Task UpdateTwin()
{
    var twin = await _registryManager.GetTwinAsync
        (_deviceId);
    var toUpdate = @"{
        tags:{
            location: {
                region: 'DE'
            }
        },
        properties: {
            desired: {
                telemetryConfig: {
                    sendFrequency: '5m'
                },
                $metadata: {
                    $lastUpdated:
                        '2020-07-14T10:47:29.8590777Z'
                },
                $version: 1
            }
        }
    }";
```

```

        }
    }";

    await _registryManager.UpdateTwinAsync(_deviceId,
    toUpdate, twin.ETag);
}

```

As you can see in the codeblock, we are updating the location tag and the desired properties with custom configuration values. All we have to do next with our backend application is call this method from the Main method. This is how the Main method should look now.

```

static async Task Main(string[] args)
{
    _serviceClient = ServiceClient.CreateFrom
    ConnectionString(_deviceConnectionString);
    _registryManager = RegistryManager.CreateFrom
    ConnectionString(_deviceConnectionString);
    await UpdateTwin();
    await InvokeDirectMethod(methodName);
    Console.WriteLine("Hello World!");
}

```

As we have already set up our backend, it's time to make some changes to the device application. Open the application and add a desired property change callback to the Main function, as follows.

```

// Set desired property update callback
    await _deviceClient.SetDesiredPropertyUpdateCall
    backAsync(OnDesiredPropertyChangedAsync, null).
    ConfigureAwait(false);

```

Now let's write the callback function.

```

private static async Task OnDesiredPropertyChangedAsync(TwinCollection desiredProperties, object userContext)
{
    Console.WriteLine($"New desired property is {desiredProperties.ToJson()}");
    TwinCollection reportedProperties, telemetryConfig;
    reportedProperties = new TwinCollection();
    telemetryConfig = new TwinCollection();
    telemetryConfig["status"] = "success";
    reportedProperties["telemetryConfig"] =
        telemetryConfig;
    await _deviceClient.UpdateReportedPropertiesAsync(
        reportedProperties).ConfigureAwait(false);
}

```

In the callback function, we are updating the reported property as an acknowledgment and passing the status value as success. The `UpdateReportedPropertiesAsync` function will update the properties.

Go back to your main method and write code to set the initial value of the reported property `sendFrequency` under `telemetryConfig`.

```

var twin = await _deviceClient.GetTwinAsync();
    Console.WriteLine($"Initial Twin: {twin.ToJson()}");

    TwinCollection reportedProperties, telemetryConfig;
    reportedProperties = new TwinCollection();
    telemetryConfig = new TwinCollection();
    telemetryConfig["sendFrequency"] = "5m";
    reportedProperties["telemetryConfig"] =
        telemetryConfig;

```

```

        await _deviceClient.UpdateReportedPropertiesAsync
            (reportedProperties).ConfigureAwait(false);
Console.WriteLine("Waiting 30 seconds for IoT Hub Twin
updates...");
        await Task.Delay(3 * 1000);

```

Here, `sendFrequency` is a property of the `telemetryConfig` twin collection. Here is the full code of the `Program.cs` class:

```

using System;
using System.Text;
using Iot.Device.CpuTemperature;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
using raspberrypi.net.core.Models;
using Microsoft.Azure.Devices.Shared;

namespace raspberrypi.net.core
{
    class Program
    {
        private static CpuTemperature _rpiCpuTemp = new
            CpuTemperature();
        private const string _deviceConnectionString =
            "HostName=apressiothub.azure-devices.net;
            DeviceId=rpiofficeroom;SharedAccessKey=Zz40yJ06odR5aLu6
            x9tzSpE8sUy3vBEfQThsRipN2WA=";
        private static int _messageId = 0;
        private static DeviceClient _deviceClient;
        private const double _temperatureThreshold = 40;
        public const string DeviceId = "rpiofficeroom";
        private const string methodName = "TurnOnLight";

```

```

static async Task Main(string[] args)
{
    _deviceClient = DeviceClient.
    CreateFromConnectionString(_deviceConnectionString,
    TransportType.Mqtt);
    // Create a handler for the direct method call
    _deviceClient.SetMethodHandlerAsync(methodName,
    TurnOnLight, null).Wait();
    // Set desired property update callback
    await _deviceClient.SetDesiredPropertyUpdateCall
    backAsync(OnDesiredPropertyChangedAsync, null).
    ConfigureAwait(false);

    var twin = await _deviceClient.GetTwinAsync();
    Console.WriteLine($"Initial Twin: {twin.ToJson()}");

    TwinCollection reportedProperties, telemetryConfig;
    reportedProperties = new TwinCollection();
    telemetryConfig = new TwinCollection();
    telemetryConfig["sendFrequency"] = "5m";
    reportedProperties["telemetryConfig"] =
    telemetryConfig;

    await _deviceClient.UpdateReportedPropertiesAsync
    (reportedProperties).ConfigureAwait(false);
    Console.WriteLine("Waiting 30 seconds for IoT Hub
    Twin updates...");
    await Task.Delay(3 * 1000);

    while (true)
    {
        if (_rpiCpuTemp.IsAvailable)
        {

```

```

        await SendToIoTHub(_rpiCpuTemp.Temperature.
            Celsius);
        Console.WriteLine("The device data has been
            sent");
    }
    Thread.Sleep(5000); // Sleep for 5 seconds
}
}

private static async Task OnDesiredPropertyChangedAsync
(TwinCollection desiredProperties, object userContext)
{
    Console.WriteLine($"New desired property is
        {desiredProperties.ToJson()}");
    TwinCollection reportedProperties, telemetryConfig;
    reportedProperties = new TwinCollection();
    telemetryConfig = new TwinCollection();
    telemetryConfig["status"] = "success";
    reportedProperties["telemetryConfig"] =
        telemetryConfig;
    await _deviceClient.UpdateReportedPropertiesAsync
        (reportedProperties).ConfigureAwait(false);
}

private static Task<MethodResponse>
TurnOnLight(MethodRequest methodRequest, object
userContext)
{
    Console.WriteLine("Here is the call from cloud to
        turn of the light!");
    var result = "{\\"result\":"\\"Executed direct method:
        " + methodRequest.Name + "\\"}";
}

```

```

        return Task.FromResult(new MethodResponse(Encoding.
            UTF8.GetBytes(result), 200));
    }

    private static async Task SendToIoTHub(double tempCelsius)
    {
        string jsonData = JsonConvert.SerializeObject(new
            DeviceData()
            {
                MessageId = _messageId++,
                Temperature = tempCelsius
            });
        var messageToSend = new Message(Encoding.UTF8.
            GetBytes(jsonData));
        messageToSend.Properties.Add("TemperatureAlert",
            (tempCelsius > _temperatureThreshold) ? "true" :
            "false");
        await _deviceClient.SendEventAsync(messageToSend).
            ConfigureAwait(false);
    }
}
}
}

```

Now put a debugger in the `OnDesiredPropertyChangedAsync` function and run the device application. Once it is running, run your backend application. The backend application will change the properties and the debugger will be called in the callback function. See Figure 7-11.



Figure 7-11. *OnDesiredPropertyChangedAsync* method debugger

As you can see in the debugger window, we passed these values from the backend application. Now, from the Azure IoT Tools, right-click the device name and select the Edit Device Twin option. Your `azure-iot-device-twin.json` file should now look like this:

```

{
  "deviceId": "rpiofficerroom",
  "etag": "AAAAAAAAAAAY=",
  "deviceEtag": "NzgyOTM1NDc2",
  "status": "enabled",
  "statusUpdateTime": "0001-01-01T00:00:00Z",
  "connectionState": "Disconnected",
  "lastActivityTime": "2020-07-19T12:12:02.2900013Z",
  "cloudToDeviceMessageCount": 0,
  "authenticationType": "sas",
  "x509Thumbprint": {
    "primaryThumbprint": null,
    "secondaryThumbprint": null
  },
  "version": 11,
  "tags": {
    "location": {
      "region": "DE"
    }
  }
}

```

```

    }
  },
  "properties": {
    "desired": {
      "telemetryConfig": {
        "sendFrequency": "5m"
      },
      "$metadata": {
        "$lastUpdated": "2020-07-19T12:10:44.6185564Z",
        "$lastUpdatedVersion": 4,
        "telemetryConfig": {
          "$lastUpdated": "2020-07-19T12:10:
          44.6185564Z",
          "$lastUpdatedVersion": 4,
          "sendFrequency": {
            "$lastUpdated": "2020-07-19T12:10:44.61
            85564Z",
            "$lastUpdatedVersion": 4
          }
        }
      }
    },
    "$version": 4
  },
  "reported": {
    "telemetryConfig": {
      "sendFrequency": "5m",
      "status": "success"
    },
    "$metadata": {
      "$lastUpdated": "2020-07-19T12:12:02.2743455Z",
      "telemetryConfig": {

```

```

        "$lastUpdated": "2020-07-19T12:12:02.
        2743455Z",
        "sendFrequency": {
            "$lastUpdated":
            "2020-07-19T12:07:31.3467328Z"
        },
        "status": {
            "$lastUpdated":
            "2020-07-19T12:12:02.2743455Z"
        }
    },
    "$version": 5
}
},
"capabilities": {
    "iotEdge": false
}
}

```

Note the status property in the reported property. Wow, that was amazing, right? Let's move on to the next session.

Cloud-to-Device Messages

This approach is used to send a notification to the device, and it is one-way communication. Note that most of these features are available only in the standard tier of IoT Hub.

To send a cloud-to-device message, we use a service-facing endpoint `/messages/devicebound`. The device will receive the message through a device-specific endpoint, called `/devices/{deviceid}/messages/devicebound`. Note that each device can hold a maximum of 50 cloud-to-device messages.

The devices can reject these kinds of messages from the cloud; in this case, IoT Hub will set this to the dead lettered state. The devices can also abandon these messages. IoT Hub will then put the message back in the queue, with the Enqueued state.

Enough for the introduction, let's rewrite the device application and the backend application. Here are the steps:

1. The backend application sends the cloud-to-device message.
2. The device application receives the cloud message.
3. The device application sends the delivery feedback.
4. The backend application receives the delivery feedback.

Sending the Cloud-to-Device Message

Let's create a function called `SendCloudToDeviceMessageAsync()` in our backend application to send the cloud message to the device.

```
private static async Task SendCloudToDeviceMessageAsync()
{
    var message = new Message(Encoding.ASCII.
        GetBytes("This is a message from cloud"));
    message.Ack = DeliveryAcknowledgement.Full; // This is to
                                                request the
                                                feedback
    await _serviceClient.SendAsync(_deviceId, message);
}
```

Here, the line `message.Ack = DeliveryAcknowledgement.Full;` requests feedback of the delivery of our cloud-to-device message. Don't forget to call this function in your main function. This is how your main function will look:

```

static async Task Main(string[] args)
{
    _serviceClient = ServiceClient.
    CreateFromConnectionString(_deviceConnectionString);
    _registryManager = RegistryManager.
    CreateFromConnectionString(_deviceConnectionString);
    await SendCloudToDeviceMessageAsync();
    await UpdateTwin();
    await InvokeDirectMethod(methodName);
    Console.WriteLine("Hello World!");
}

```

Receiving the Cloud-to-Device Message and Sending Feedback

The backend is now capable of sending the cloud-to-device message, so let's create a new function in the device application to receive this message.

```

private static async Task ReceiveCloudToDeviceMessageAsync()
{
    while (true)
    {
        var cloudMessage = await _deviceClient.
        ReceiveAsync();
        if (cloudMessage == null) continue;
        Console.WriteLine($"The received message is:
        {Encoding.ASCII.GetString(cloudMessage.
        GetBytes())}");
        await _deviceClient.
        CompleteAsync(cloudMessage); // Send feedback
    }
}

```

Don't forget to call this method in the main method. Let's comment out all the other code, so this is how the new main method looks.

```
static async Task Main(string[] args)
{
    _deviceClient = DeviceClient.
    CreateFromConnectionString(_deviceConnectionString,
    TransportType.Mqtt);
    // // Create a handler for the direct method call
    // _deviceClient.SetMethodHandlerAsync(methodName,
    TurnOnLight, null).Wait();
    // // Set desired property update callback
    // await _deviceClient.SetDesiredPropertyUpdateCa
    llbackAsync(OnDesiredPropertyChangedAsync, null).
    ConfigureAwait(false);

    // var twin = await _deviceClient.GetTwinAsync();
    // Console.WriteLine($"Initial Twin: {twin.
    ToJson()}");
    // TwinCollection reportedProperties,
    telemetryConfig;
    // reportedProperties = new TwinCollection();
    // telemetryConfig = new TwinCollection();
    // telemetryConfig["sendFrequency"] = "5m";
    // reportedProperties["telemetryConfig"] =
    telemetryConfig;

    // await _deviceClient.UpdateReportedPropertiesAsyn
    c(reportedProperties).ConfigureAwait(false);
    // Console.WriteLine("Waiting 30 seconds for IoT
    Hub Twin updates...");
    // await Task.Delay(3 * 1000);
}
```

```

        // while (true)
        // {
        //     if (_rpiCpuTemp.IsAvailable)
        //     {
        //         await SendToIoTHub(_rpiCpuTemp.
        //             Temperature.Celsius);
        //         Console.WriteLine("The device data has
        //             been sent");
        //     }
        //     Thread.Sleep(5000); // Sleep for 5 seconds
        // }
        await ReceiveCloudToDeviceMessageAsync();
        // Cloud to device receiver
    }

```

The `ReceiveAsync()` function returns null after a timeout period. When the app receives null, it should continue to wait for new messages. That is why we added a condition to check whether `cloudMessage` is null.

The `CompleteAsync()` function notifies IoT Hub that the message has been processed and can be safely removed.

Receiving Feedback from the Device

We have seen how to send cloud-to-device message feedback from the device, so let's learn how to receive that feedback in the backend application. The following `ReceiveDeliveryFeedback()` function does that job.

```

private static async Task ReceiveDeliveryFeedback()
{
    var feedbackReceiver = _serviceClient.
        GetFeedbackReceiver();
}

```

```
while (true)
{
    var feedback = await feedbackReceiver.
    ReceiveAsync();
    if (feedback == null) continue;
    Console.WriteLine($"The feedback status is:
    {string.Join(",", feedback.Records.Select(s =>
    s.StatusCode))}");
    await feedbackReceiver.CompleteAsync(feedback);
}
}
```

Don't forget to add this function to the main method.

Demo Application

Now let's run our device application and backend application. If everything goes well, you should see the output shown in Figures 7-12 and 7-13. Figure 7-12 is from the device application and Figure 7-13 is from the backend application.

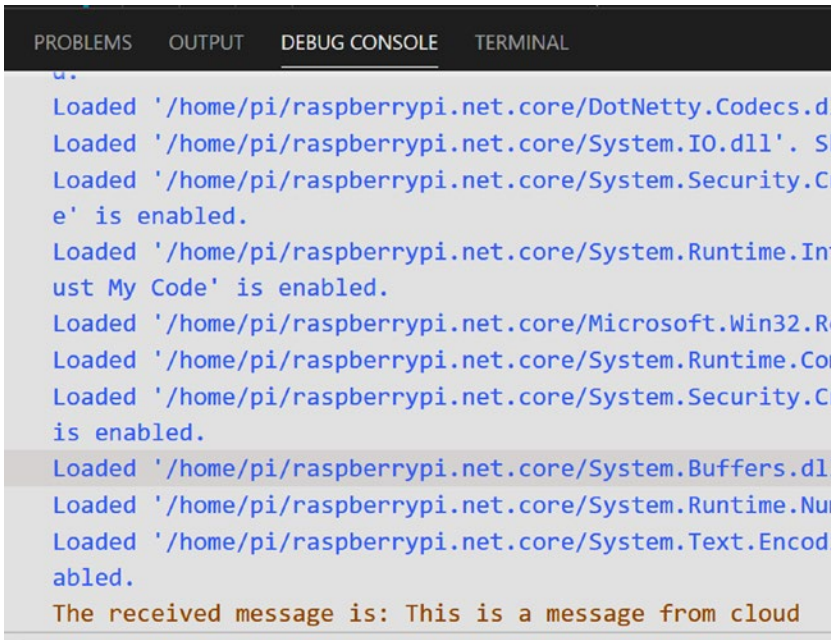


Figure 7-12. Cloud-to-device message



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Loaded 'C:\Program Files\dotnet\shared\Micro
t My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
t My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
gger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
er option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
'Just My Code' is enabled.
Loaded 'Anonymously Hosted DynamicMethods
Loaded 'C:\Program Files\dotnet\shared\Micro
option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
on 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Micro
r option 'Just My Code' is enabled.
The feedback status is: Success
```

Figure 7-13. Cloud-to-device message feedback

Summary

I hope you were able to follow along with this chapter hands-on. In this chapter, you learned:

- The ways you can communicate with the device from the cloud.
- What the direct method is and how to implement it?

CHAPTER 7 CLOUD TO DEVICE COMMUNICATION

- What device twins are and how to implement them?
- What a cloud-to-device message is and how to implement it?

Now let's move on to the next chapter.

CHAPTER 8

IoT Edge

In the last few chapters, you learned about bi-directional communication between the device and the IoT Hub. You also saw the actions in the demo. For a full understanding of IoT Hub, we need to also discuss IoT Edge. Therefore, in this chapter, we cover the following topics:

- What is IoT Edge?
- Why is it so popular?
- How does it work?

What do you think? Feeling excited? Let's start.

IoT Edge

The Azure IoT Edge is a managed service built on Azure IoT Hub. Usually, when we send the data to the cloud, the workloads happen in the cloud. But with the help of IoT Edge, we can move this workload to our device, via standard containers. These workloads can be artificial intelligence, your business logic, or any other Azure or third-party services. This way, the communication with the device and cloud is limited and the device can react more quickly to the data or to changes.

IoT Edge Runtime

The IoT Edge runtime is a collection of programs. These programs turn a device into an IoT Edge device. In short, the IoT Edge runtime components enable the IoT Edge devices to receive code to run at the edge and communicate the results.

IoT Edge Modules

The IoT Edge allows us to deploy and manage business logic on the edge in the form of modules. These modules are the smallest units managed by IoT Edge. They can contain any Azure services (such as Azure stream analytics), third-party services, or your code. Here are the elements of a module:

- A *module image* is a package containing the software.
- A *module instance* is the unit of computation running the module image on the IoT Edge; it is started by the IoT Edge runtime.
- A *module identity* is a piece of information stored in IoT Hub; it's associated with each module instance. This piece of information includes the security credentials.
- A *module twin* is a JSON document stored in IoT Hub. It contains the state information for a module instance. This can contain metadata, configurations, and conditions.

Module images exist on the cloud, and they can be updated, changed, and deployed in different solutions. The module images must handle only a single purpose. For example, don't create a module with the capability of AI and application insights. A new instance of the module will be created whenever a new module image is deployed to a device and started by the IoT Edge runtime.

Each time a module instance is created by the IoT Edge runtime, it gets a new corresponding module identity. This identity depends on the identity of the device and the module name. For example, if your module name is `Logger` and you deploy it on the `RpiOfficeRoom` device, the corresponding module identity will be `/devices/RpiOfficeRoom/modules/Logger`.

Each module instance will have its corresponding module twin that we can use to configure the module instance. The module twin is also a JSON document, just like the device twin, and it stores the module information and its configuration.

Capabilities of IoT Edge Runtime

Here are the responsibilities of the IoT Edge runtime:

- Install and update the workload on the device.
- Maintain Azure IoT Edge security standards on the device.
- Ensure that the IoT Edge modules are always running.
- Report module health to the cloud for remote monitoring.
- Manage communication between downstream devices and IoT Edge devices.
- Manage communication between modules on the IoT Edge device.
- Manage communication between the IoT Edge device and the cloud.

Creating an IoT Edge Device

The IoT Edge allows you to manage code on your device remotely. This makes it easier for you to send more of your workload to the edge. To start, we need an IoT Hub, which we already have. We need to create an IoT Edge device. As this device behaves and is managed differently, we need to create this device differently. Go to your IoT Hub in the Azure Portal and click the IoT Edge section. Then click + Add an IoT Edge Device, as shown in Figure 8-1.

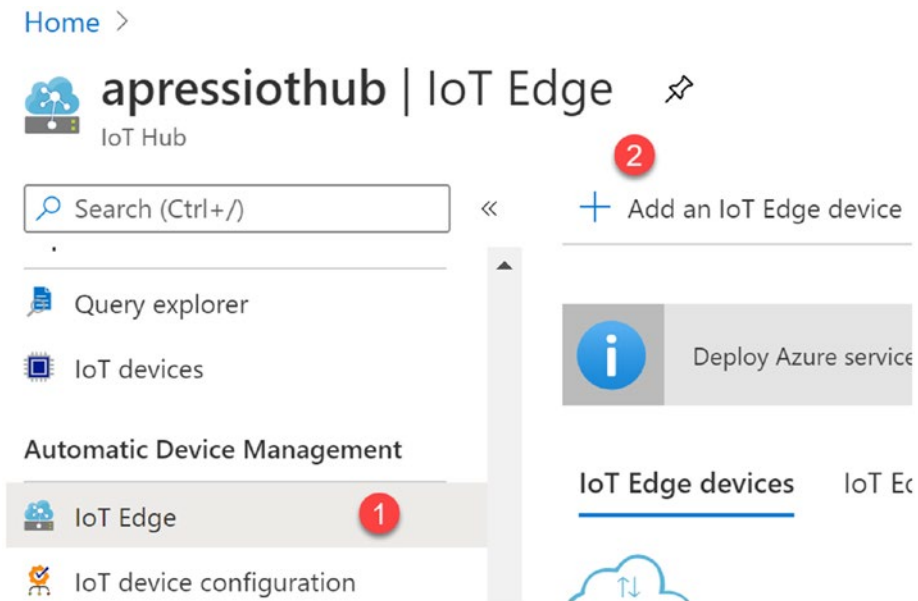





Figure 8-1. Adding an IoT Edge device

The create screen is the same one as with a normal IoT device, as shown in Figure 8-2.

Create a device

 Find Certified for Azure IoT devices in the Device Catalog 

Device ID * 


 

Figure 8-2. *Creating an IoT Edge device*

The device will be listed in the IoT Edge device list. Click that list to get the connection string. See Figure 8-3.

The screenshot shows the configuration page for a device named 'rpiedge' in the Azure IoT Hub portal. The 'Primary Connection String' field is highlighted, showing the connection string: `HostName=apressiohub.azure-devices.net,DeviceId=rpiedge,SharedAccessKey=0+Z5akF7N7huX2acu1SZ11jmNTJFLBMTBQUy9me4qc=`. Other fields include Device ID (rpiedge), Primary Key, Secondary Key, Secondary Connection String, IoT Edge Runtime Response (NA), and a checkbox for 'Enable connection to IoT Hub' which is checked.

NAME	TYPE	SPECIFIED IN DEPLOYMENT	REPORTED BY DEVICE	RUNTIME STATUS
\$edgeAgent	Module Identity	NA	NA	NA
\$edgeHub	Module Identity	NA	NA	NA

Figure 8-3. IoT Edge device connection string

As you can see in Figure 8-3, the connection string lists the IoT Hub name, device ID, and the shared access key to do the authentication.

Installing IoT Edge Runtime on Linux Systems

Since we are using Raspberry Pi, we need to install the IoT Edge runtime. Use the following command to register the Microsoft key and the software repository feed before you install the container runtime. SSH to your Raspberry Pi and run the following command:

```
curl https://packages.microsoft.com/config/debian/stretch/multiarch/prod.list > ./microsoft-prod.list
```

Now copy `./microsoft-prod.list` to `/etc/apt/sources.list.d/` by running the following command:

```
sudo cp ./microsoft-prod.list /etc/apt/sources.list.d/
```

Now install the Microsoft GPG public key:

```
curl https://packages.microsoft.com/keys/microsoft.asc |
gpg --dearmor > microsoft.gpg
sudo cp ./microsoft.gpg /etc/apt/trusted.gpg.d/
```

Once that is done, run `sudo apt-get update`. Figure 8-4 shows the command-line output.

```
pi@raspberrypi:~$ curl https://packages.microsoft.com/config/debian/stretch/multiarch/prod.list > ./microsoft-prod.list
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 104 100 104 0 0 19 0 0:00:05 0:00:05 --:--:-- 21
pi@raspberrypi:~$ sudo cp ./microsoft-prod.list /etc/apt/sources.list.d/
pi@raspberrypi:~$ curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg
sudo cp ./microsoft.gpg /etc/apt/trusted.gpg.d/ % Total % Received % Xferd Average Speed Time Time Time C
urrent
Dload Upload Total Spent Left Speed
100 983 100 983 0 0 4890 0 --:--:-- --:--:-- --:--:-- 4890
pi@raspberrypi:~$ sudo cp ./microsoft.gpg /etc/apt/trusted.gpg.d/
pi@raspberrypi:~$ sudo apt-get update
Hit:1 http://archive.raspberrypi.org/debian buster InRelease
Get:2 http://rasbian.raspberrypi.org/raspbian buster InRelease [15.0 kB]
Get:3 https://packages.microsoft.com/debian/stretch/multiarch/prod stretch InRelease [29.8 kB]
Get:4 http://rasbian.raspberrypi.org/raspbian buster/main armhf Packages [13.0 MB]
Get:5 https://packages.microsoft.com/debian/stretch/multiarch/prod stretch/main armhf Packages [7,125 B]
Get:6 https://packages.microsoft.com/debian/stretch/multiarch/prod stretch/main all Packages [546 B]
Get:7 https://packages.microsoft.com/debian/stretch/multiarch/prod stretch/main arm64 Packages [3,904 B]
Get:8 https://packages.microsoft.com/debian/stretch/multiarch/prod stretch/main amd64 Packages [5,696 B]
Fetched 13.1 MB in 7s (1,783 kB/s)
Reading package lists... Done
```

Figure 8-4. *IoT Edge device prerequisites*

All the prerequisites to install the container runtime are ready, so let's install the Moby engine (`mobyproject.org`). It is worth it to mention that the Moby engine is the only container engine officially supported by Azure IoT Edge. Run the following command to install the `moby-engine`.

```
sudo apt-get install moby-engine
```

Now we can install the IoT Edge Security Daemon. The IoT Edge Security Daemon provides and maintains the security standards on the Edge device. It starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

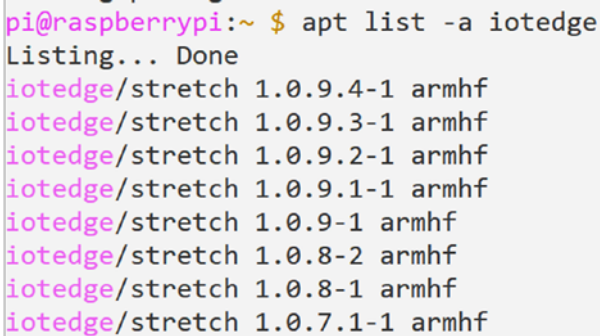
Update the package lists on your device as follows:

```
sudo apt-get update
```

You can check which versions of IoT Edge are available by running the following command.

```
apt list -a iotedge
```

You should get a list of available versions, as shown in Figure 8-5.



```
pi@raspberrypi:~ $ apt list -a iotedge
Listing... Done
iotedge/stretch 1.0.9.4-1 armhf
iotedge/stretch 1.0.9.3-1 armhf
iotedge/stretch 1.0.9.2-1 armhf
iotedge/stretch 1.0.9.1-1 armhf
iotedge/stretch 1.0.9-1 armhf
iotedge/stretch 1.0.8-2 armhf
iotedge/stretch 1.0.8-1 armhf
iotedge/stretch 1.0.7.1-1 armhf
```

Figure 8-5. IoT Edge available versions

If you want to install a specific version of the security daemon, you can mention that in the command along, with the `libiothsm-std` package version.

```
sudo apt-get install iotedge=1.0.9* libiothsm-std=1.0.9*
```

The `sudo apt-get install iotedge` command will install the latest version of both security daemon and `libiothsm-std`.

```
sudo apt-get install iotedge
```

Once IoT Edge is installed at `/etc/iotedge/`, the command window will show you a message about updating the configuration file, as shown in Figure 8-6.

```

Setting up iotedge (1.0.9.4-1) ...
=====
                                Azure IoT Edge

IMPORTANT: Please update the configuration file located at:

    /etc/iotedge/config.yaml

with your device's provisioning information. You will need to restart the
'iotedge' service for these changes to take effect.

To restart the 'iotedge' service, use:

    'systemctl restart iotedge'

- OR -

    /etc/init.d/iotedge restart

These commands may need to be run with sudo depending on your environment.

=====
Created symlink /etc/systemd/system/sockets.target.wants/iotedge.mgmt.socket →
Created symlink /etc/systemd/system/multi-user.target.wants/iotedge.service → /
Created symlink /etc/systemd/system/sockets.target.wants/iotedge.socket → /lib/
Processing triggers for man-db (2.8.5-2) ...
Processing triggers for systemd (241-7~deb10u4+rpil) ...

```

Figure 8-6. *Install IoT Edge*

This configuration file is available at `/etc/iotedge/`. This is the file we can use to provision the device. To see the contents of this file, run the following command.

```
sudo nano /etc/iotedge/config.yaml
```

There are two ways you can provision your device:

- Manually
- Automatically

As we have only one device, we will use manual provisioning. Automatic provisioning can be done using the Device Provisioning Service, which will automatically provision your devices. Figure 8-7 shows how your configuration file looks.

```

pi@raspberrypi: ~
GNU nano 3.2 /etc/iotedge/config.yaml
#####
#                               IoT Edge Daemon configuration
#####
#
# This file configures the IoT Edge daemon. The daemon must be restarted to
# pick up any configuration changes.
#
# Note - this file is yaml. Learn more here: http://yaml.org/refcard.html
#
#####
# Provisioning mode and settings
#####
#
# Configures the identity provisioning mode of the daemon.
#
# Supported modes:
#   manual   - using an iot hub connection string
#   dps      - using dps for provisioning
#   external - the device has been provisioned externally.
#             Uses an external provisioning endpoint to get device specific i
#
# DPS Settings
#   scope_id      - Required. Value of a specific DPS instance's ID scope
#   registration_id - Required for TPM and symmetric key provisioning flows.
#                   Optional for X.509 provisioning. Registration ID of a
#                   specific device in DPS.
#                   For more information regarding DPS registration ids
#                   please see https://docs.microsoft.com/en-us/azure/iot-dp
#   symmetric_key - Optional. This entry should only be specified when
#                   provisioning devices configured for symmetric key
#                   attestation. Device specific symmetric key.
#   identity_cert - Optional. The Edge device identity X.509 certificate

```

Figure 8-7. IoT Edge configuration file

This configuration file is a `.yaml` file, so spacing and indentation are very important. Make sure you uncomment the manual provisioning configuration and comment out the other provisioning sections. See Figure 8-8.

```

pi@raspberrypi: ~
GNU nano 3.2 /etc/iotedge/config.
# Manual provisioning configuration
provisioning:
  source: "manual"
  device_connection_string: "<ADD DEVICE CONNECTION STRING HERE>"

# DPS TPM provisioning configuration
# provisioning:
#   source: "dps"
#   global_endpoint: "https://global.azure-devices-provisioning.net"
#   scope_id: "<SCOPE_ID>"
#   attestation:
#     method: "tpm"
#     registration_id: "<REGISTRATION_ID>"

# DPS symmetric key provisioning configuration
# provisioning:
#   source: "dps"
#   global_endpoint: "https://global.azure-devices-provisioning.net"
#   scope_id: "<SCOPE_ID>"
#   attestation:
#     method: "symmetric_key"
#     registration_id: "<REGISTRATION_ID>"
#     symmetric_key: "<SYMMETRIC_KEY>"

# DPS X.509 provisioning configuration
# provisioning:
#   source: "dps"
#   global_endpoint: "https://global.azure-devices-provisioning.net"
#   scope_id: "<SCOPE_ID>"
#   attestation:
#     method: "x509"
#     registration_id: "<OPTIONAL REGISTRATION ID. LEAVE COMMENTED OUT>"
#     identity_cert: "<REQUIRED URI TO DEVICE IDENTITY CERTIFICATE>"
#     identity_pk: "<REQUIRED URI TO DEVICE IDENTITY PRIVATE KEY>"

```

Figure 8-8. IoT Edge manual provisioning

You need to update the value of `device_connection_string` with your IoT Edge device connection string from the Azure Portal.

```
# Manual provisioning configuration
provisioning:
  source: "manual"
  device_connection_string: "<ADD DEVICE CONNECTION STRING
  HERE>"
```

Once you update it with the connection string, just save the file and close it (Ctrl+X, Type Y, and then press Enter).

Now restart the daemon by running the following command.

```
sudo systemctl restart iotedge
```

If you have successfully finished the provisioning, you can check the status of your IoT Edge daemon by running the following command.

```
systemctl status iotedge
```

If everything goes well, you should see the output in Figure 8-9.

```
pi@raspberrypi:~$ systemctl status iotedge
● iotedge.service - Azure IoT Edge daemon
   Loaded: loaded (/lib/systemd/system/iotedge.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2020-07-30 16:56:18 BST; 6min ago
     Docs: man:iotedged(8)
    Main PID: 2324 (iotedged)
      Tasks: 11 (limit: 4915)
     Memory: 3.3M
    CGroup: /system.slice/iotedge.service
            └─2324 /usr/bin/iotedged -c /etc/iotedge/config.yaml

Jul 30 17:01:53 raspberrypi iotedged[2324]: 2020-07-30T16:01:53Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:01:58 raspberrypi iotedged[2324]: 2020-07-30T16:01:58Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:03 raspberrypi iotedged[2324]: 2020-07-30T16:02:03Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:08 raspberrypi iotedged[2324]: 2020-07-30T16:02:08Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:13 raspberrypi iotedged[2324]: 2020-07-30T16:02:13Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:18 raspberrypi iotedged[2324]: 2020-07-30T16:02:18Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:23 raspberrypi iotedged[2324]: 2020-07-30T16:02:23Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:28 raspberrypi iotedged[2324]: 2020-07-30T16:02:28Z [INFO] - [mgmt] - - - [2020-07-30 16:0
Jul 30 17:02:32 raspberrypi iotedged[2324]: 2020-07-30T16:02:32Z [INFO] - Checking edge runtime status
Jul 30 17:02:32 raspberrypi iotedged[2324]: 2020-07-30T16:02:32Z [INFO] - Edge runtime is running.
pi@raspberrypi:~$
```

Figure 8-9. IoT Edge status

It is relevant to mention that you can always check the daemon logs by running this command:

```
journalctl -u iotedge --no-pager --no-full
```

You can also troubleshoot your IoT Edge device by running the check command. It runs a collection of configuration and connectivity tests.

```
sudo iotedge check
```

The previous command will check both configuration and connectivity checks. Some of them are shown in Figures 8-10 and 8-11.

```
pi@raspberrypi:~ $ sudo iotedge check
Configuration checks
-----
√ config.yaml is well-formed - OK
√ config.yaml has well-formed connection string - OK
√ container engine is installed and functional - OK
√ config.yaml has correct hostname - OK
√ config.yaml has correct URIs for daemon mgmt endpoint - OK
√ latest security daemon - OK
√ host time is close to real time - OK
√ container time is close to host time - OK
```

Figure 8-10. IoT Edge configuration check

```
Connectivity checks
-----
√ host can connect to and perform TLS handshake with IoT Hub AMQP port - OK
√ host can connect to and perform TLS handshake with IoT Hub HTTPS / WebSockets port - OK
√ host can connect to and perform TLS handshake with IoT Hub MQTT port - OK
√ container on the default network can connect to IoT Hub AMQP port - OK
√ container on the default network can connect to IoT Hub HTTPS / WebSockets port - OK
√ container on the default network can connect to IoT Hub MQTT port - OK
√ container on the IoT Edge module network can connect to IoT Hub AMQP port - OK
√ container on the IoT Edge module network can connect to IoT Hub HTTPS / WebSockets port - OK
√ container on the IoT Edge module network can connect to IoT Hub MQTT port - OK
18 check(s) succeeded.
```

Figure 8-11. IoT Edge connectivity check

If you ever want to see all the modules on your IoT Edge device, you can use the `list` command:

```
sudo iotedge list
```

Figure 8-12 shows the modules list.

```
pi@raspberrypi:~$ sudo iotedge list
NAME          STATUS      DESCRIPTION          CONFIG
edgeAgent     running     Up 43 minutes       mcr.microsoft.com/azureiotedge-agent:1.0
pi@raspberrypi:~$
```

Figure 8-12. IoT Edge modules list

As you can see, only one module is running at this time, which is `edgeAgent`. But no worries, there will be other modules once you do your first deployment.

Deploying a Module to IoT Edge Device

Now that you have set up your IoT Edge device, it's time to deploy some modules to it from the cloud. Are you ready? See Figure 8-13.

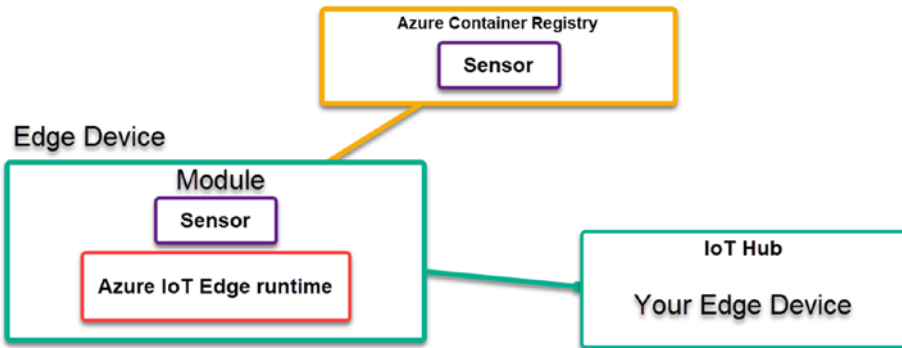


Figure 8-13. IoT Edge deploy module

As discussed, the modules are executable packages implemented as containers. Log in to your Azure Portal and go to the IoT Hub that you created. From the menu on the left pane, under Automatic Device Management, select IoT Edge, and then click the device that you created. See Figure 8-14.

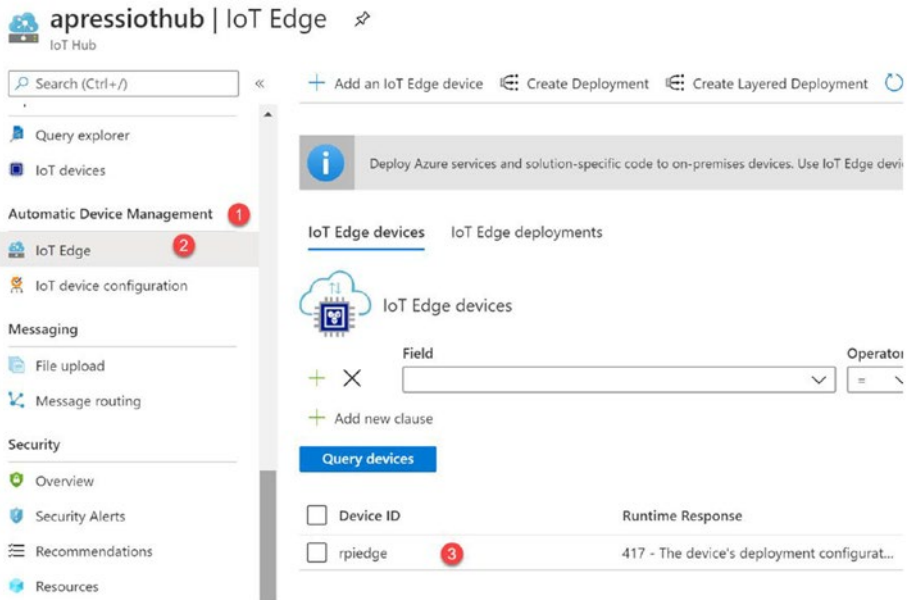


Figure 8-14. *IoT Edge automatic deploy module*

On the next screen, click the Set Modules button, as shown in Figure 8-15.

Home > apressiothub | IoT Edge >

rpiedge ✕
apressiothub

Save Set Modules Manage Child Devices Device Twin Manage keys Refresh

Device ID ⓘ	rpiedge
Primary Key ⓘ
Secondary Key ⓘ
Primary Connection String ⓘ
Secondary Connection String ⓘ
IoT Edge Runtime Response ⓘ	417 -- The device's deployment configuration is not set

Figure 8-15. IoT Edge set module

There are two ways you can deploy a module to your IoT Edge device:

- Using the Container Registry (see Figure 8-16)
- Using IoT Edge modules

Home > apressiothub | IoT Edge > rpiedge >

Set modules on device: rpiedge

apressiothub

Modules Routes Review + create

Container Registry Credentials

You can specify credentials to container registries hosting module images. Listed credentials are used to retrieve modules with a matching URL. The Edge Agent will report error code 500 if it can't find a container registry setting for a module.

NAME	ADDRESS	USER NAME	PASSWORD
<input type="text" value="Name"/>	<input type="text" value="Address"/>	<input type="text" value="User name"/>	<input type="text" value="Password"/>

IoT Edge Modules

An IoT Edge module is a Docker container you can deploy to IoT Edge devices. It communicates with other modules and sends data to the IoT Edge runtime. Using this UI you can import Azure Service IoT Edge modules or specify the settings for an IoT Edge module. Setting modules on each device will be counted towards the quota and throttled based on the IoT Hub tier and units. For example, for S1 tier, modules can be set 10 times per second if no other updates are happening in the IoT Hub. [Learn more](#)

+ Add Runtime Settings

	DESIRED STATUS
NA + IoT Edge Module	
TH + Marketplace Module	
+ Azure Stream Analytics Module	

Figure 8-16. IoT Edge set module page

The Azure Container Registry stores and manages private Docker container images. You might have already worked with Docker Hub (hub.docker.com), which stores and manages the public Docker container images. To get your private modules from your Azure Container Registry, you have to provide your credentials. With the help of credentials, the Registry will retrieve the modules with a matching URL. Note that the Edge Agent will report error code 500 if it cannot find a Container Registry setting for a module.

The IoT Edge module is a Docker container that you can deploy to your IoT Edge device. Luckily, there are enough pre-built modules in the IoT Edge Module section of the Azure Marketplace (azuremarketplace.microsoft.com). In this section, we will use one from the marketplace that simulates a sensor and sends generated data.

Click + Add, and then select Marketplace Module from the IoT Edge Modules section, as shown in Figure 8-17.

IoT Edge Modules

An IoT Edge module is a Docker container you Service IoT Edge modules or specify the settir example, for S1 tier, modules can be set 10 tir [Learn more](#)

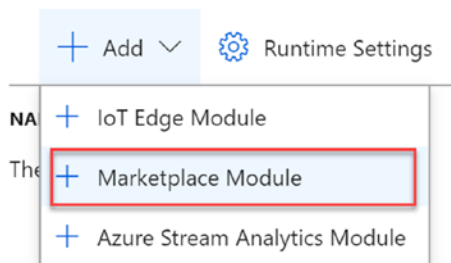


Figure 8-17. *IoT Edge marketplace Module*

Once the IoT Edge Marketplace Module is loaded, search for the Simulated Temperature Sensor module and add it (see Figure 8-18).

IoT Edge Module Marketplace

Marketplace

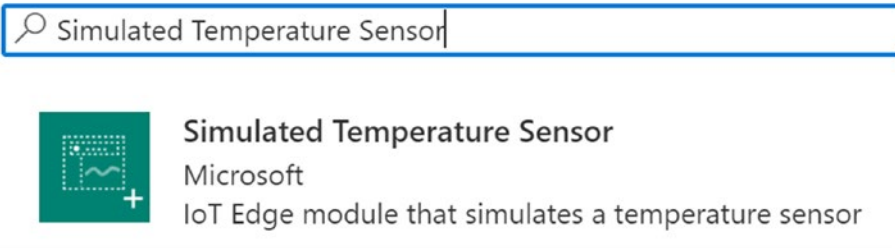


Figure 8-18. *IoT Edge simulated module*

The Simulated Temperature Sensor module is added with the desired state. Click the Next button to set the routes, as shown in Figure 8-19.

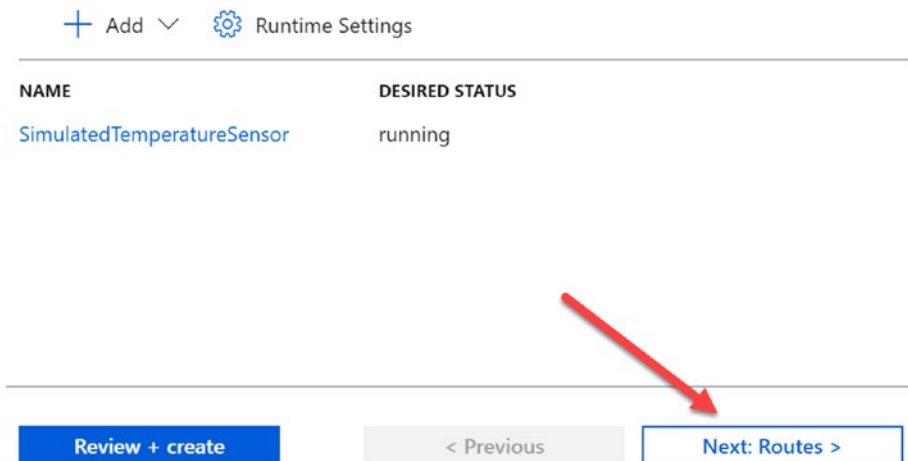


Figure 8-19. *IoT Edge added module*



The routes determine how messages are passed between modules and the IoT Hub. With the help of routes, we can send the data to other services, if necessary. As you can see in Figure 8-20, these routes are name/value pairs, and you should see two routes there now. One is the default route, which sends all the messages to the IoT Hub. The second one is created automatically when you add the Simulated Temperature Sensor module. Let's check the value of that route.

```
FROM /messages/modules/SimulatedTemperatureSensor/* INTO $upstream
```

Once that is done, click the Next button.

Modules Routes **Review + create**

Routes
 You can set routes between modules, which gives you the flexibility to send messages where they need to go without the need for additional services to process messages or to write additional code.
[Learn more](#)

NAME	VALUE	
route 1	FROM /messages/* INTO \$upstream	
2 SimulatedTemperatureSensor...	FROM /messages/modules/SimulatedTemperatureSensor/* INTO \$...	
Route name	FROM /messages/* INTO \$upstream	

Review + create < Previous **Next: Review + create >**




Figure 8-20. Module routes

On the next screen, you can preview the JSON file that defines all the modules that are deployed to your IoT Edge device. If you want to make any changes, this is when to do it. Here's the example JSON file.

```
{
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "modules": {
          "SimulatedTemperatureSensor": {
            "settings": {
              "image": "mcr.microsoft.com/
azureiotedge-simulated-temperature-
sensor:1.0",
              "createOptions": ""
            },
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "version": "1.0"
          }
        },
        "runtime": {
          "settings": {
            "minDockerVersion": "v1.25"
          },
          "type": "docker"
        },
        "schemaVersion": "1.0",
        "systemModules": {
          "edgeAgent": {
            "settings": {
```

```

        "image": "mcr.microsoft.com/
        azureiotedge-agent:1.0",
        "createOptions": ""
    },
    "type": "docker"
},
"edgeHub": {
    "settings": {
        "image": "mcr.microsoft.com/
        azureiotedge-hub:1.0",
        "createOptions": "{\"HostConfig\":{\"
        \"PortBindings\":{\"443/tcp\":[{\\"H
        ostPort\":\\\"443\\\"}],\\\"5671/tcp\":[{\
        \\\"HostPort\":\\\"5671\\\"}],\\\"8883/tcp\
        \":[{\\"HostPort\":\\\"8883\\\"}]}}}"
    },
    "type": "docker",
    "status": "running",
    "restartPolicy": "always"
}
}
},
"$edgeHub": {
    "properties.desired": {
        "routes": {
            "route": "FROM /messages/* INTO $upstream",
            "SimulatedTemperatureSensorToIoTHub":
            "FROM /messages/modules/
            SimulatedTemperatureSensor/* INTO
            $upstream"
        },
    },

```



```

        "schemaVersion": "1.0",
        "storeAndForwardConfiguration": {
            "timeToLiveSecs": 7200
        }
    },
    "SimulatedTemperatureSensor": {
        "properties.desired": {
            "SendData": true,
            "SendInterval": 5
        }
    }
}
}
}

```

You can see the module that you had added as well as two runtime modules—`edgeAgent` and `edgeHub`. Click the Create button when you are done reviewing. It is worth it to mention that, when you submit a new deployment to your IoT Edge, nothing is pushed to your device. Instead, the device queries the IoT Hub regularly. If it finds an updated manifest, it pulls the updated module image from the cloud and starts running those modules locally.

This submission will take a few seconds, and once it is done, you will see the "Successfully updated IoT Edge settings for device `rpiedge`" notification. You can now check all the modules in the modules list, as shown in Figure 8-21.

Modules	IoT Edge Hub connections	Deployments				
NAME	TYPE	SPECIFI...	REPORT...	RUNTI...	EXIT CO...	
\$edgeAgent	IoT Edge System M...	✓ Yes	✓ Yes	running	0	
\$edgeHub	IoT Edge System M...	✓ Yes	✓ Yes	running	0	
SimulatedTemperatureSensor	IoT Edge Custom M...	✓ Yes	✓ Yes	running	0	

Figure 8-21. *Running modules*

Viewing Sent Messages

Since we have an IoT Edge device with modules running in it, let's SSH into that device and see the messages getting sent. See [Figure 8-22](#).

```
pi@raspberrypi:~ $ sudo iotedge list
NAME                STATUS      DESCRIPTION      CONFIG
SimulatedTemperatureSensor  running    Up 8 minutes     mcr.microsoft.com/azureiot
edge-simulated-temperature-sensor:1.0
edgeAgent           running    Up 2 days        mcr.microsoft.com/azureiot
edge-agent:1.0
edgeHub             running    Up 7 minutes     mcr.microsoft.com/azureiot
edge-hub:1.0
```

Figure 8-22. *New modules in the device*

To see the messages being sent from the simulator module, use the following command.

```
sudo iotedge logs SimulatedTemperatureSensor -f
```

This will give you the output shown in [Figure 8-23](#).

```

pi@raspberrypi:~$ sudo iotedge logs SimulatedTemperatureSensor -f
[2020-08-02 12:45:26 +00:00]: Starting Module
SimulatedTemperatureSensor Main() started.
Initializing simulated temperature sensor to send 500 messages, at an interval of 5 seconds.
To change this, set the environment variable MessageCount to the number of messages that should be sent (set it to -1 to send unlimited messages).
Information: Trying to initialize module client using transport type [Amqp_Tcp_Only].
Information: Successfully initialized module client of transport type [Amqp_Tcp_Only].
08/02/2020 12:46:38> Sending message: 1, Body: [{"machine":{"temperature":22.016715180951504,"pressure":1.1158283117539689},"ambient":{"temperature":21.208730870722203,"humidity":25},"timeCreated":"2020-08-02T12:46:38.6041514Z"}]
08/02/2020 12:46:44> Sending message: 2, Body: [{"machine":{"temperature":22.436544679541395,"pressure":1.1636569888085133},"ambient":{"temperature":21.318105262153832,"humidity":26},"timeCreated":"2020-08-02T12:46:44.1677571Z"}]
08/02/2020 12:46:49> Sending message: 3, Body: [{"machine":{"temperature":23.618484604343998,"pressure":1.2983083726467846},"ambient":{"temperature":21.241491453136081,"humidity":26},"timeCreated":"2020-08-02T12:46:49.2075935Z"}]

```

Figure 8-23. Simulator module messages

Isn't this cool? We were able to use a module in our device without having to configure that module specifically. So in the future, we can easily add as many modules as required, without thinking about the device.

Summary

Wow, you have finished this chapter. I hope you have learned the following topics from this chapter:

- What IoT Edge is and the benefits of using it?
- What IoT Edge runtime is?
- What IoT Edge modules are?
- The capabilities of IoT Edge runtime.
- How to create an IoT Edge device?
- How to install IoT Edge runtime on Linux systems?
- How to deploy a module to an IoT Edge device and see the data being sent to the IoT Hub?

I will see you in the next chapter.

CHAPTER 9

Developing IoT Edge Modules

In the last chapter, you successfully deployed a module to your IoT Edge device, but you used a module that was already in the marketplace. In this chapter, you will learn how to develop your own module and deploy it. Sound interesting? I cannot wait to show you how to do that. Let's start.

Prerequisites

Before we start, it is a good idea to list all the prerequisites you'll need. As usual, we will be using the Visual Studio Code for development. I am developing the module on a Windows computer intending to target a Linux device running IoT Edge, the Raspberry Pi. Other things to note are:

- Your development machine should support nested virtualization. This is needed to run a container engine. We use Docker Desktop to develop the containers. You can easily switch between Linux containers and Windows containers using Docker

Desktop (see Figure 9-1). This gives you the power to create modules for different types of IoT Edge devices. You can install Docker Desktop from the Docker Hub (docs.docker.com/docker-for-windows/install).

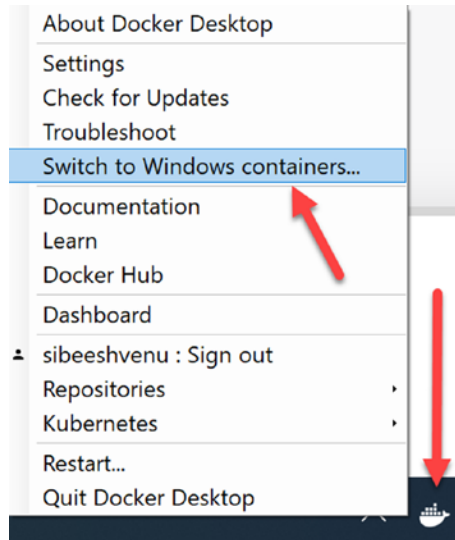


Figure 9-1. Docker switch to Windows containers

- You should install Git, in order to pull the module template package C# extension powered by OmniSharp. Remember that we added this to Visual Studio?

Setting Up VSCode

Because of the Azure IoT Tools extension, creating IoT solutions with VSCode has never been easier. With this extension, you can easily create projects using the given project templates, automate the deployment, and monitor and manage IoT devices. Go to the Extensions tab in VSCode and install Azure IoT Tools (see Figure 9-2).

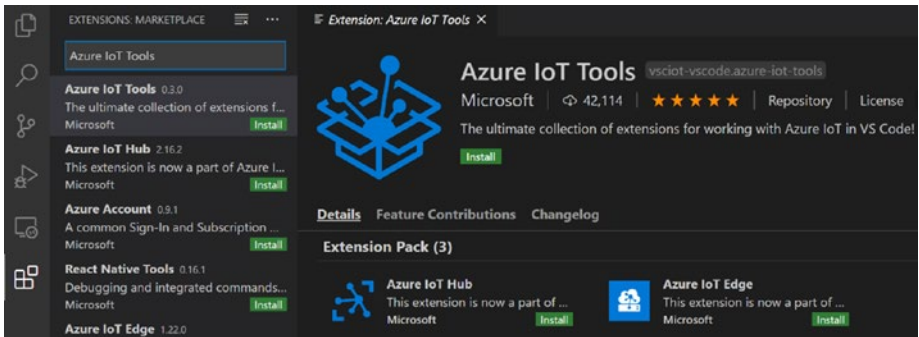


Figure 9-2. *Azure IoT Tools*

Once the extension is installed, go to the View menu and click Command Palette. Search for and select Azure: Sign in. Azure will ask you to sign in with your existing Azure account; remember to sign in with the account you used to create the Azure IoT Hub.

Once you sign in, open the Command Palette again. Search for and select Azure IoT Hub: Select IoT Hub, which will give you an option to select your IoT Hub. When you are finished, you can see your devices in the VSCode Explorer (choose View ► Explorer), as shown in Figure 9-3.

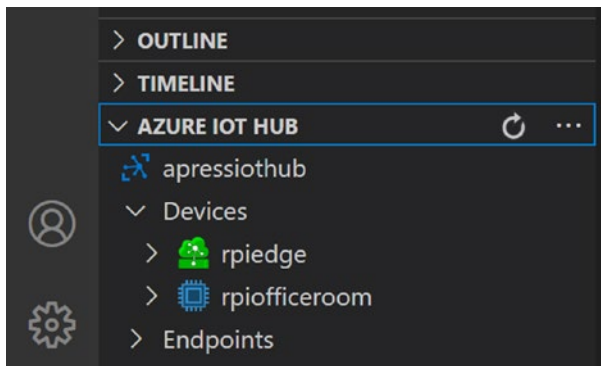


Figure 9-3. *Azure IoT Explorer*

Creating an Azure Container Registry

This chapter aims to deploy the container image of the modules created by using the Azure IoT Tool in any Docker-compatible registry. We will use the Azure Container Registry in this example, but you can also use Docker Hub. Let's create an Azure Container Registry in the Azure Portal now. Search for the Container Registry service and create it. Creating this service is as easy as creating other services in Azure. It is recommended to create the resource in the same resource group.

All the SKUs provide the same programmatic capabilities; however, choosing the higher SKU will provide more performance and scale. Table 9-1 shows some of the differences between the SKUs.

Table 9-1. *Container Registry Capabilities and Differences*

Capability	Basic	Standard	Premium
Price per day	\$0.17	\$0.67	\$1.67
Included storage (GiB)	10	100	500
			Offers enhanced throughput for docker pulls across multiple, concurrent nodes
Total web hooks	2	10	500
Geo Replication	Not Supported	Not Supported	Supported
			\$1.667 per replicated region

In this example, we will use the Basic SKU. When you are done, review and create the resource. See Figure 9-4.

Create container registry

Validation passed

Basics Networking Encryption Tags Review + create

Registry details

Basics

Registry name	apresscr
Subscription	
Resource Group	ApressBook
Location	North Europe
SKU	Basic

Networking

Allow public network access	Yes
-----------------------------	-----

Encryption

Customer-Managed Key	Disabled
Identity	None
Key Vault	None
Encryption key	None
Version	None

Tags

purpose	book
---------	------

[Create](#) [< Previous](#) [Next >](#)

Figure 9-4. Azure Container Registry

Now go to the resource you just created and click the Access Keys from the menu located under settings. From the given page, enable the Admin user so that you can use the registry name as the username and use the password generated to log in to your container registry. See Figure 9-5.

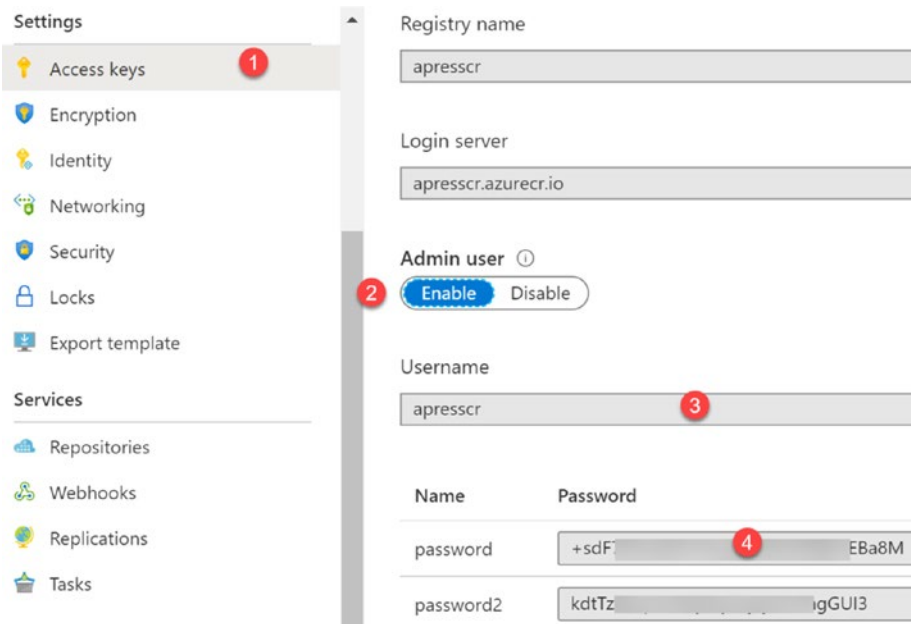


Figure 9-5. Container Registry admin access

Creating a New Project

As discussed earlier, it is very easy to create a new project with the IoT Tools. To do this, open the Command Palette and search for and select Azure IoT Edge: New IoT Edge Solution. Once you press Enter, it will ask you to select the folder where the files should be saved and then to provide a solution name. I named it `raspberrypi.edge`. Select C# Module when asked to select the module template. You can name your module anything; I named this example `SendTelemetry`. When you're asked for the Docker

image repository for the module, provide the Login Server value that you see in the Access Key section of your Azure Container Registry. It looks similar to `apresscr.azurecr.io/sendtelemetry`. You can always edit this value in the `module.json` file. See Figure 9-6.

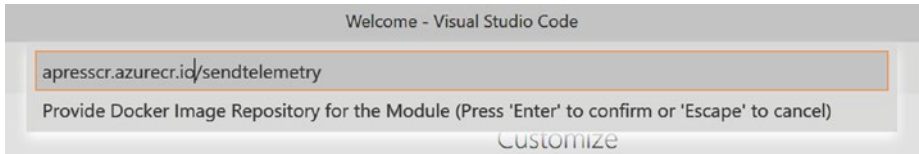


Figure 9-6. Image repository

Once the solution is created, go to the module. You'll see the `module.json` file there. This is how it looks:

```
{
  "$schema-version": "0.0.1",
  "description": "",
  "image": {
    "repository": "apresscr.azurecr.io/sendtelemetry",
    "tag": {
      "version": "0.0.1",
      "platforms": {
        "amd64": "./Dockerfile.amd64",
        "amd64.debug": "./Dockerfile.amd64.debug",
        "arm32v7": "./Dockerfile.arm32v7",
        "arm32v7.debug": "./Dockerfile.arm32v7.debug",
        "arm64v8": "./Dockerfile.arm64v8",
        "arm64v8.debug": "./Dockerfile.arm64v8.debug",
        "windows-amd64": "./Dockerfile.windows-amd64"
      }
    }
  },
}
```

```

        "buildOptions": [],
        "contextPath": "./"
    },
    "language": "csharp"
}

```

The `modules` folder will contain all the modules. Right now, there is only one module, which is where you write all of the main program code, module metadata, and Dockerfiles.

The `.env` file stores the credentials of your container registry. These credentials will be shared with your IoT Edge device to pull the container images.

To deploy these modules, we need a deployment manifest file. This file defines which modules will be deployed, how they should be configured, and how they will communicate with each other and with the cloud. The `deployment.template.json` and `deployment.debug.template.json` files help create this manifest file. Here are the contents of these files:

`deployment.debug.template.json`:

```

{
  "$schema-template": "2.0.0",
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "apresscr": {

```

```

    "username": "$CONTAINER_REGISTRY_USERNAME_
    aprescr",
    "password": "$CONTAINER_REGISTRY_PASSWORD_
    aprescr",
    "address": "aprescr.azurecr.io"
  }
}
},
"systemModules": {
  "edgeAgent": {
    "type": "docker",
    "settings": {
      "image": "mcr.microsoft.com/azureiotedge-
      agent:1.0",
      "createOptions": {}
    }
  },
  "edgeHub": {
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "mcr.microsoft.com/azureiotedge-
      hub:1.0",
      "createOptions": {
        "HostConfig": {
          "PortBindings": {
            "5671/tcp": [
              {
                "HostPort": "5671"
              }
            ]
          }
        }
      }
    }
  }
}
}
}

```

```

        ],
        "8883/tcp": [
            {
                "HostPort": "8883"
            }
        ],
        "443/tcp": [
            {
                "HostPort": "443"
            }
        ]
    ]
}
}
}
}
},
"modules": {
    "SendTelemetry": {
        "version": "1.0",
        "type": "docker",
        "status": "running",
        "restartPolicy": "always",
        "settings": {
            "image": "${MODULES.SendTelemetry.debug}",
            "createOptions": {}
        }
    },
    "SimulatedTemperatureSensor": {
        "version": "1.0",
        "type": "docker",

```

```

    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "mcr.microsoft.com/azureiotedge-
        simulated-temperature-sensor:1.0",
      "createOptions": {}
    }
  }
}
},
"$edgeHub": {
  "properties.desired": {
    "schemaVersion": "1.0",
    "routes": {
      "SendTelemetryToIoTHub": "FROM /messages/modules/
        SendTelemetry/outputs/* INTO $upstream",
      "sensorToSendTelemetry": "FROM /messages/modules/
        SimulatedTemperatureSensor/outputs/temperatureOutput
        INTO BrokeredEndpoint(\"/modules/SendTelemetry/
        inputs/input1\")"
    },
    "storeAndForwardConfiguration": {
      "timeToLiveSecs": 7200
    }
  }
}
}
}
}

```

deployment.template.json:

```
{
  "$schema-template": "2.0.0",
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "apresscr": {
                "username": "$CONTAINER_REGISTRY_USERNAME_
                  apresscr",
                "password": "$CONTAINER_REGISTRY_PASSWORD_
                  apresscr",
                "address": "apresscr.azurecr.io"
              }
            }
          }
        }
      },
    },
    "systemModules": {
      "edgeAgent": {
        "type": "docker",
        "settings": {
          "image": "mcr.microsoft.com/azureiotedge-
            agent:1.0",
          "createOptions": {}
        }
      }
    }
  }
}
```

```

},
"edgeHub": {
  "type": "docker",
  "status": "running",
  "restartPolicy": "always",
  "settings": {
    "image": "mcr.microsoft.com/azureiotedge-
hub:1.0",
    "createOptions": {
      "HostConfig": {
        "PortBindings": {
          "5671/tcp": [
            {
              "HostPort": "5671"
            }
          ],
          "8883/tcp": [
            {
              "HostPort": "8883"
            }
          ],
          "443/tcp": [
            {
              "HostPort": "443"
            }
          ]
        }
      }
    }
  }
},

```



```

"modules": {
  "SendTelemetry": {
    "version": "1.0",
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "${MODULES.SendTelemetry}",
      "createOptions": {}
    }
  },
  "SimulatedTemperatureSensor": {
    "version": "1.0",
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "mcr.microsoft.com/azureiotedge-
simulated-temperature-sensor:1.0",
      "createOptions": {}
    }
  }
}
},
"$edgeHub": {
  "properties.desired": {
    "schemaVersion": "1.0",
    "routes": {
      "SendTelemetryToIoTHub": "FROM /messages/modules/
SendTelemetry/outputs/* INTO $upstream",

```

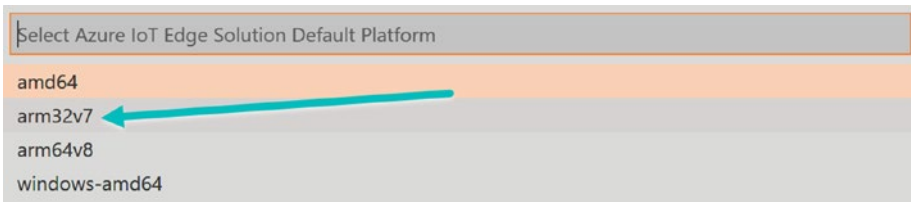



Figure 9-7. Select the default architecture

You should now see the message "azure-iot-edge.setDefaultPlatform: The default platform is arm32v7 now." in the output window. The solution that we just created includes sample code for an IoT Edge module. The sample code demonstrates how communication between modules works. The IoT Hub running on the device routes messages from the output of one module to the input of one or more modules. This communication is controlled or performed using the routes configured in `$edgeHub`. You can see the desired properties of `$edgeHub` in the `deployment.template.json` file:

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

As you can see, there are two routes—one sends the output of the SendTelemetry module to IoT Hub, and the other sends the output of SimulatedTemperatureSensor to the input of the SendTelemetry module. Every route should have a source and sink property. We can also have a where condition in the route to filter the messages, but that is optional. Here is the syntax of the route:

```
"$edgeHub": {
  "properties.desired": {
    "routes": {
      "route1": "FROM <source> WHERE <condition> INTO
<sink>",
      "route2": "FROM <source> WHERE <condition> INTO
<sink>"
    },
  },
}
```

The source field specifies where the message comes from. Table 9-2 shows the possible values for the source.

Table 9-2. Possible Source Properties

Source	Description
/*	All device-to-cloud messages or twin change notifications from any module or leaf device
/twinChangeNotifications	Any twin change (reported properties) coming from any module or leaf device
/messages/*	Any device-to-cloud message sent by a module through some or no output, or by a leaf device

(continued)

Table 9-2. (continued)

Source	Description
/messages/modules/*	Any device-to-cloud message sent by a module through some or no output
/messages/ modules/<moduleId>/*	Any device-to-cloud message sent by a specific module through some or no output
/messages/modules/ <moduleId>/outputs/*	Any device-to-cloud message sent by a specific module through some output
/messages/ modules/<moduleId>/ outputs/<output>	Any device-to-cloud message sent by a specific module through a specific output

Here is an example of a route with a filter in it:

```
FROM /messages/* WHERE NOT IS_DEFINED($connectionModuleId) INTO
$upstream
```

All the messages coming from the modules include a system property called `connectionModuleId`, so if you want to exclude module messages, you can use this query with the condition.

The sink defines where the messages are sent. There are a few things to note:

- Only modules and IoT Hub can receive messages
- Messages can't be routed to other devices
- There are no wildcard options in the sink property

Table 9-3. Possible Sink Property Values

Sink	Description
\$upstream	Send the message to IoT Hub
BrokeredEndpoint("/modules/<moduleId>/inputs/<input>")	Send the message to a specific input of a specific module

If you look in the `Program.cs` file, which is inside the `modules\SendTelemetry` folder, you should see a handler called `SetInputMessageHandlerAsync`, which will be called when a message is received by the module.

```
await ioTHubModuleClient.SetInputMessageHandlerAsync("input1",
PipeMessage, ioTHubModuleClient);
```

The `SetInputMessageHandlerAsync` method registers a new delegate for the particular input. If a delegate is already associated with the input, it will be replaced with the new one. Here is the `Init` method:

```
static async Task Init()
{
    MqttTransportSettings mqttSetting = new Mqtt
    TransportSettings(TransportType.Mqtt_Tcp_Only);
    ITransportSettings[] settings = { mqttSetting };

    // Open a connection to the Edge runtime
    ModuleClient ioTHubModuleClient = await
    ModuleClient.CreateFromEnvironmentAsync(settings);
    await ioTHubModuleClient.OpenAsync();
    Console.WriteLine("IoT Hub module client
    initialized.");
}
```

```

        // Register callback to be called when a message is
        // received by the module
        await ioTHubModuleClient.SetInputMessageHandler
        Async("input1", PipeMessage, ioTHubModuleClient);
    }

```

As you can see, a message handler is the second parameter of the `SetInputMessageHandlerAsync` method. This message handler pipes the message and sends the event to the IoT device. Here is the `PipeMessage` method:

```

static async Task<MessageResponse> PipeMessage(Message
message, object userContext)
{
    int counterValue = Interlocked.Increment(ref
counter);

    var moduleClient = userContext as ModuleClient;
    if (moduleClient == null)
    {
        throw new InvalidOperationException("UserContext
doesn't contain " + "expected values");
    }

    byte[] messageBytes = message.GetBytes();
    string messageString = Encoding.UTF8.GetString
(messageBytes);
    Console.WriteLine($"Received message:
{counterValue}, Body: [{messageString}]");

    if (!string.IsNullOrEmpty(messageString))
    {
        using (var pipeMessage = new
Message(messageBytes))

```

```

    {
        foreach (var prop in message.Properties)
        {
            pipeMessage.Properties.Add(prop.Key,
                prop.Value);
        }
        await moduleClient.SendEventAsync
            ("output1", pipeMessage);

        Console.WriteLine("Received message sent");
    }
}
return MessageResponse.Completed;
}

```

The `SendEventAsync` method processes the messages and sets up an output queue (`output1`) to pass them. As we have gone through the files and code, now we can try building our solution and generate the manifest. Before we build the container image, we must perform a Docker login with our Azure Container Registry credentials. Run the following command in the Terminal (choose `View` ► `Terminal`).

```
docker login -u aadresscr -p +sdF70FxKZ9C7NvyBZZHFDJHFPjBJ5EBa8M
aadresscr.azurecr.io
```

You should now see the `Login Succeeded` message in the terminal window. Let's log in to the Azure Container Registry, right after performing `az login`. When you run the `az login` command, it will ask you to log in with your Azure account. After you do that, run the following command.

```
az acr login -n aadresscr
```


If you get this error:

'az' is not recognized as an internal or external command, operable program or batch file

You need to install Azure CLI. The command uses the token created when you executed `az login`. To build the solution, right-click the `deployment.template.json` file and select Build and Push IoT Edge Solution, as shown in Figure 9-8.

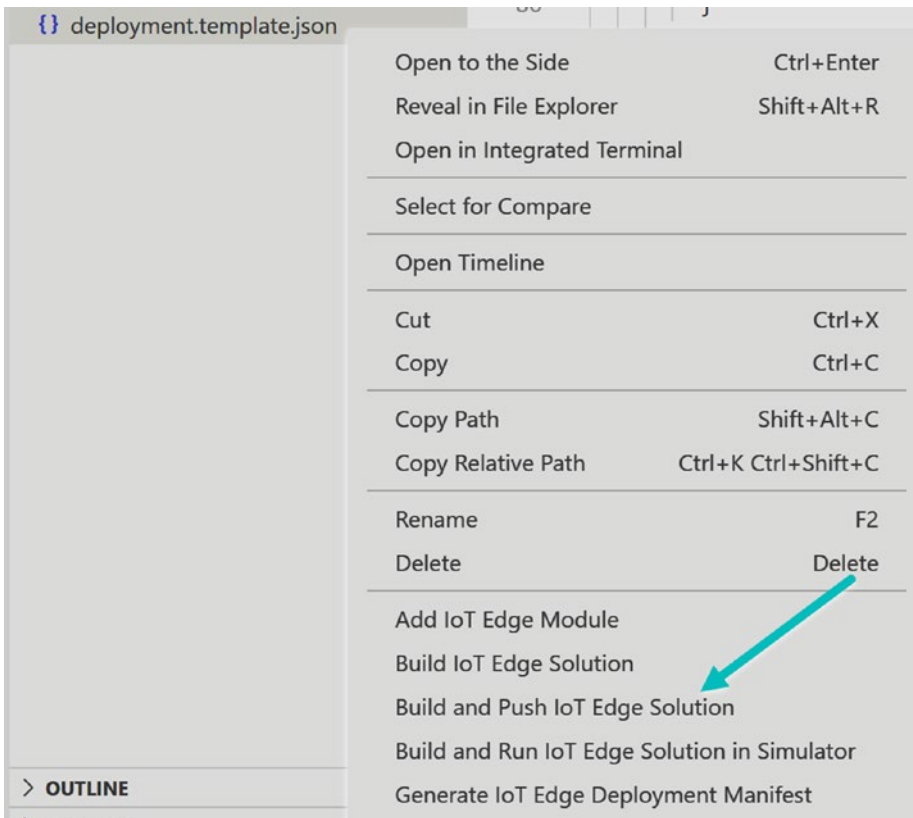


Figure 9-8. Choose Build and Push IoT Edge Solution

This will run the Docker commands and you'll be able to see the progress in the terminal window. The build and push command performs three operations:

- Creates a config folder that contains the full deployment manifest, built out of information in the deployment template.
- Runs a Docker build to build the container image based on the appropriate Dockerfile for your target architecture. Since we set the default architecture to Arm32, it will use the `Dockerfile.arm32v7` Dockerfile.
- In the end, it pushes the image repository to your container registry by running the Docker push command.

We also have to change the base image in the `Dockerfile.arm32v7` file, since we are using a Windows 64 bit machine and our target device has an arm32 architecture. You can easily check this by running the following commands:

```
pi@raspberrypi:~ $ uname -a
Linux raspberrypi 4.19.118-v7l+ #1311 SMP Mon Apr 27 14:26:42
BST 2020 armv7l GNU/Linux
pi@raspberrypi:~ $ uname -m
armv7l
```

If the `uname -m` command says `armv7l`, it is 32-bit. In my case, it is `armv7l`. So I had to build an `arm32` container image on my 64-bit Windows host machine and use it on my Raspberry Pi 4. Luckily, you can always build `arm32` and `arm64` images on `x64` machines, but will not be able to run them.

CHAPTER 9 DEVELOPING IOT EDGE MODULES

As we already have a device running on Raspberry Pi, running is not a problem for us. Open the `Dockerfile.arm32v7` file and change the first line from:

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster-arm32v7 AS build-env
```

to:

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build-env
```

This is how your Dockerfile should look now:

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build-env
WORKDIR /app

COPY *.csproj ./
RUN dotnet restore

COPY . ./
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/runtime:3.1-buster-slim-
arm32v7
WORKDIR /app
COPY --from=build-env /app/out ./

RUN useradd -ms /bin/bash moduleuser
USER moduleuser

ENTRYPOINT ["dotnet", "SendTelemetry.dll"]
```

Note that it may take a while the first time and will be faster when you run it the next time. A new file called `deployment.arm32v7.json` is created in the `config` folder; if you check the contents of this file, you can see that all the credential values are updated from the `.env` file. Here is the sample generated file:

```
{
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "apresscr": {
                "username": "apresscr",
                "password": "+sdF70FxKZGDGD9C7NvyBZZM2DPjBJ5
EBa8M",
                "address": "apresscr.azurecr.io"
              }
            }
          }
        }
      }
    },
    "systemModules": {
      "edgeAgent": {
        "type": "docker",
        "settings": {
          "image": "mcr.microsoft.com/azureiotedge-
agent:1.0",
```

```

        "createOptions": "{}"
    }
},
"edgeHub": {
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
        "image": "mcr.microsoft.com/azureiotedge-
            hub:1.0",
        "createOptions": "{\"HostConfig\":{\"Port
            Bindings\":{\"5671/tcp\":[{\"HostPort\":
            \\\"5671\\\"}],\\\"8883/tcp\":[{\"HostPort\":
            \\\"8883\\\"}],\\\"443/tcp\":[{\"HostPort\":
            \\\"443\\\"}]}}}"
    }
}
},
"modules": {
    "SendTelemetry": {
        "version": "1.0",
        "type": "docker",
        "status": "running",
        "restartPolicy": "always",
        "settings": {
            "image": "apresscr.azurecr.io/sendtelemetry:
                0.0.2-arm32v7",
            "createOptions": "{}"
        }
    }
},
"SimulatedTemperatureSensor": {
    "version": "1.0",

```


Now log in to the Azure Portal and open the Container Registry resource that you created. Click the Repositories menu item on the left pane. This will show the module name there, which is `sendtelemetry`. See Figure 9-9.

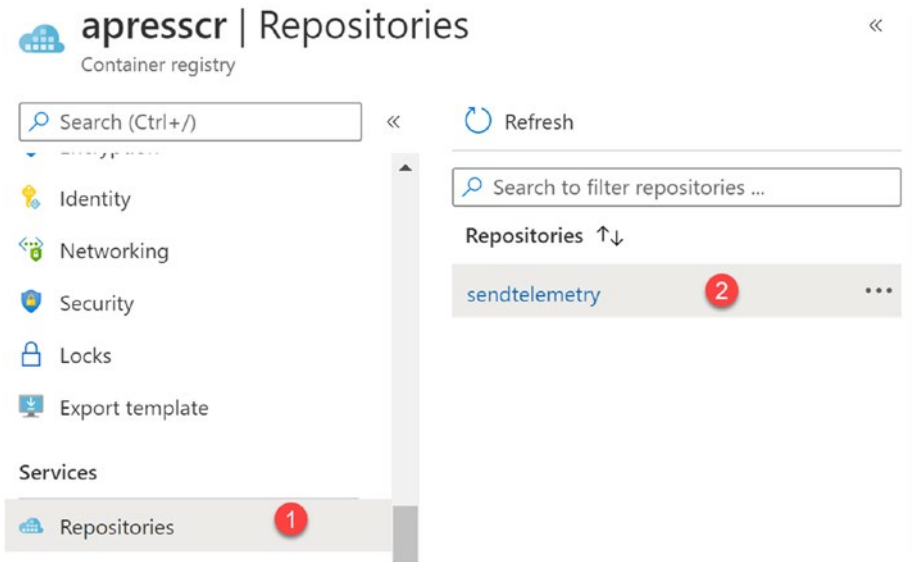


Figure 9-9. Azure container registry repository

Now click the `sendtelemetry` repository. You should see your image with the right version number and tag there, as shown in Figure 9-10.

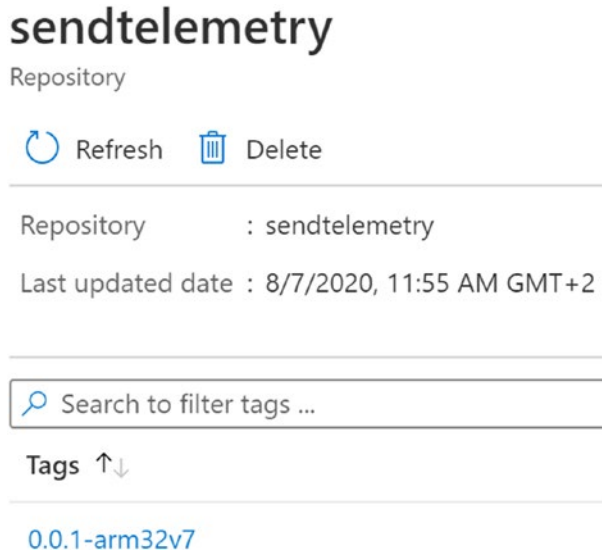
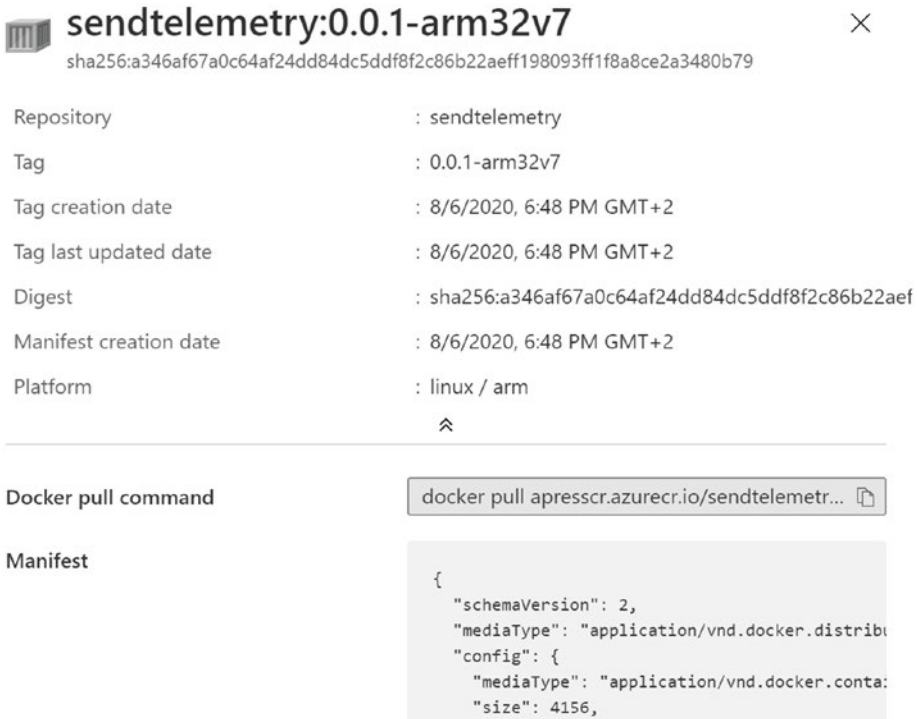


Figure 9-10. *Send telemetry image*

If you click the tag, you'll see full details of your image with the manifest; see Figure 9-11.



sendtelemetry:0.0.1-arm32v7 ×

sha256:a346af67a0c64af24dd84dc5ddf8f2c86b22aeff198093ff1f8a8ce2a3480b79

Repository	: sendtelemetry
Tag	: 0.0.1-arm32v7
Tag creation date	: 8/6/2020, 6:48 PM GMT+2
Tag last updated date	: 8/6/2020, 6:48 PM GMT+2
Digest	: sha256:a346af67a0c64af24dd84dc5ddf8f2c86b22aeef
Manifest creation date	: 8/6/2020, 6:48 PM GMT+2
Platform	: linux / arm

Docker pull command `docker pull apresscr.azurecr.io/sendtelemetry...`

Manifest

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distrib...",
  "config": {
    "mediaType": "application/vnd.docker.conta...",
    "size": 4156,
```

Figure 9-11. *Send telemetry image details*

You can also change the version number of these container images. This will help you have a separate set of functionalities in each version and test the functionalities in a small set of devices before you deploy the changes to the production. To change this version number, open the `module.json` file inside your module folder—in our case, it is the `modules\SendTelemetry` folder—and update the property version to the new number. See Figure 9-12.

```

{} module.json ×
modules > SendTelemetry > {} module.json > {} image > {} tag > [v] version
1  {
2    "$schema-version": "0.0.1",
3    "description": "",
4    "image": {
5      "repository": "apresscr.azurecr.io/sendtelemetry",
6      "tag": [
7        "version": "0.0.2",
8      "platforms": {
9        "amd64": "./Dockerfile.amd64",
10       "amd64.debug": "./Dockerfile.amd64.debug",
11       "arm32v7": "./Dockerfile.arm32v7",
12       "arm32v7.debug": "./Dockerfile.arm32v7.debug",
13       "arm64v8": "./Dockerfile.arm64v8",
14       "arm64v8.debug": "./Dockerfile.arm64v8.debug",
15       "windows-amd64": "./Dockerfile.windows-amd64"
16     }
17   },
18   "buildOptions": [],
19   "contextPath": "./"
20 },
21 "language": "csharp"
22 }
23

```

Figure 9-12. Change module version number

When you are done, right-click the `deployment.template.json` file and select **Build and Push IoT Edge Solution**. This will build a new image and push it to the container registry. Note that if you don't change this version number, it will overwrite the previous image in the container registry.

When it's done, go to your repository and refresh it. You should see two images with appropriate version numbers, as shown in Figure 9-13.

sendtelemetry

Repository

 Refresh  Delete

Repository : sendtelemetry

Tag count : 2

Last updated date : 8/7/2020, 11:55 AM GMT+2

Manifest count : 2

⤴

 Search to filter tags ...

Tags ↑↓

0.0.2-arm32v7

0.0.1-arm32v7

Figure 9-13. *New docker image in the container registry*

Deploying the Modules to the Device

The container images are ready for action in our Container Registry, so it is time to deploy them to our device. The question is, are you ready? If you are ready, make sure that your device is up and running.

Go to the Visual Studio Code Explorer. Under the Azure IoT Hub, expand the Devices menu, which will list all of your devices. Right-click the device and select Create Deployment for Single Device, as shown in Figure 9-14.

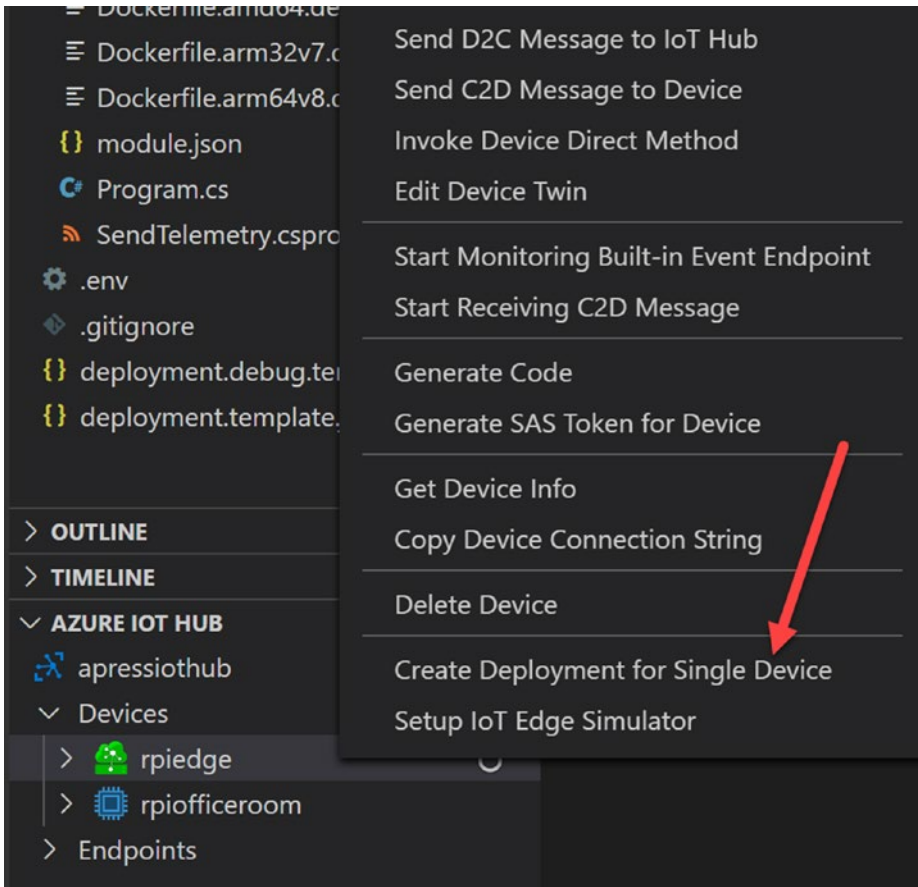


Figure 9-14. *Deploy to the device*

This will open a Windows Explorer window. Navigate to the config folder and select `deployment.arm32v7.json`. You should now see the following output in the output window.

```
[Edge] Start deployment to device [rpiedge]
[Edge] Deployment succeeded.
```

This will also add the Azure Container Registry credentials to the Set Module page of your IoT Edge device (choose IoT Hub ► IoT Edge ► IoT Edge Device ► Set Module). See Figure 9-15.

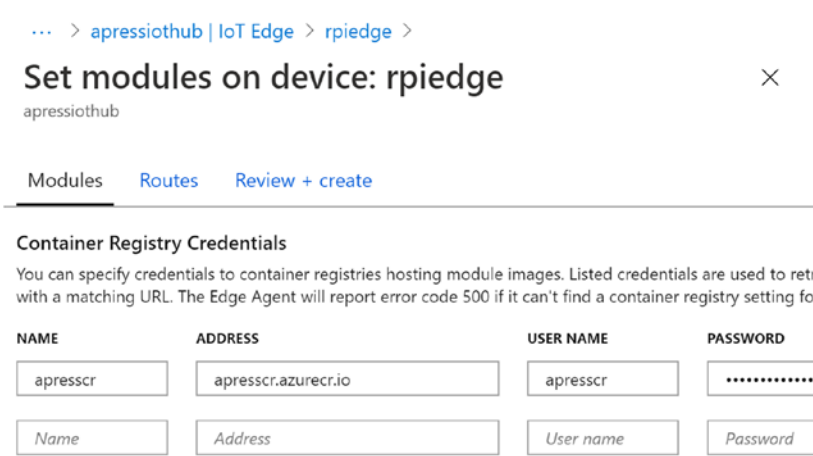


Figure 9-15. Container registry credentials

Now, click the device from the Azure IoT Hub section, and then click the Refresh button. Go to the Modules section under your device; you should see that the modules are running, as shown in Figure 9-16.

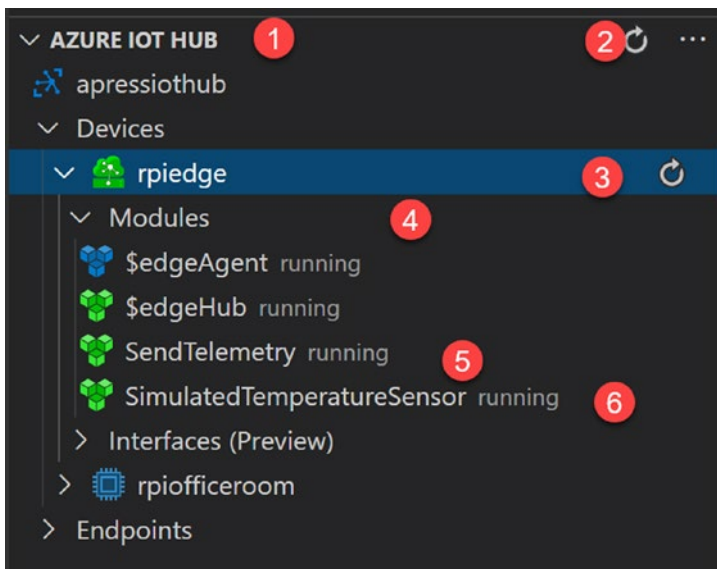


Figure 9-16. Modules running on the IoT Edge device

It may take a few minutes to start the modules, as the IoT Edge runtime needs to get the new manifest and then update the new images from the Azure Container Registry.

If you notice that your modules are in the “backoff” status, make sure that you set the default architecture correctly and updated the Dockerfile accordingly. You can also try restarting the Docker and IoT Edge services.

```
sudo systemctl restart docker
sudo systemctl restart iotedge
```

Viewing Device Messages

One amazing thing about the IoT Tools extension is that you can see everything in one place. To see the device messages, right-click the device and select Start Monitoring Built-in Event Endpoint, as shown in Figure 9-17.

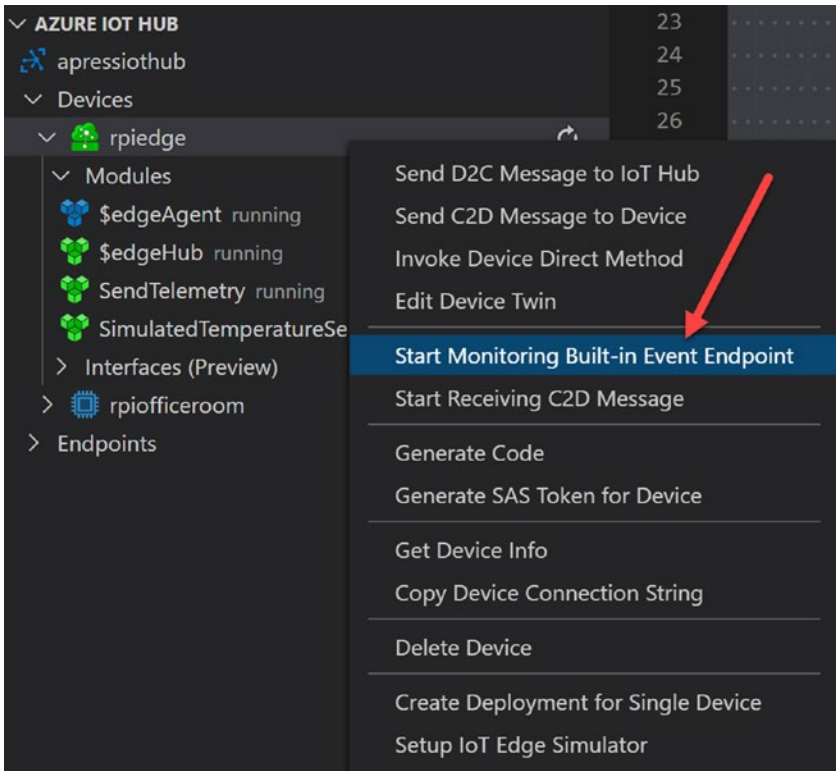
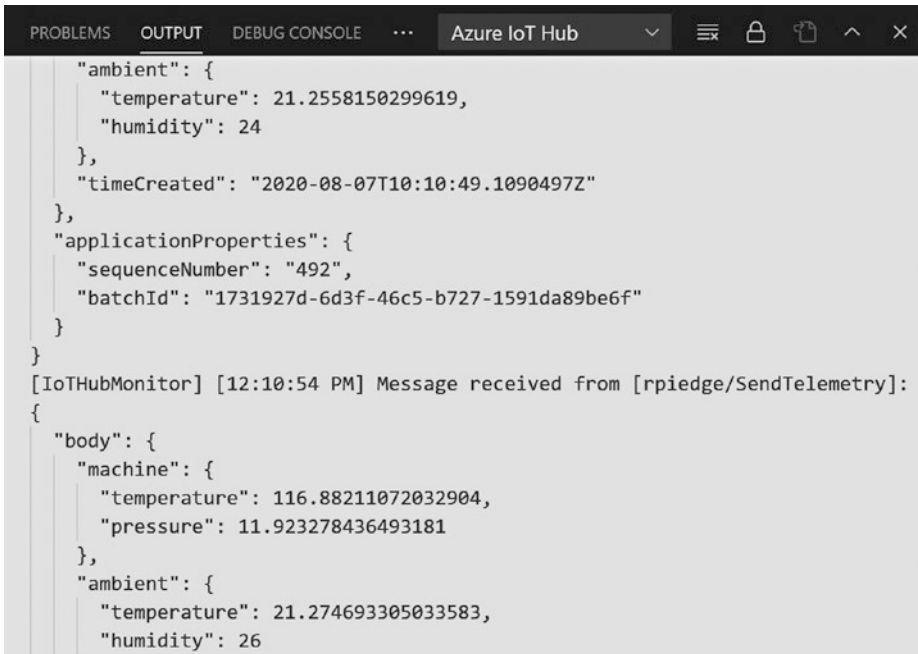


Figure 9-17. Viewing the device messages

You should now see the messages in the output window, as shown in Figure 9-18.



```

"ambient": {
  "temperature": 21.2558150299619,
  "humidity": 24
},
"timeCreated": "2020-08-07T10:10:49.1090497Z"
},
"applicationProperties": {
  "sequenceNumber": "492",
  "batchId": "1731927d-6d3f-46c5-b727-1591da89be6f"
}
}
[IoTHubMonitor] [12:10:54 PM] Message received from [rpiedge/SendTelemetry]:
{
  "body": {
    "machine": {
      "temperature": 116.88211072032904,
      "pressure": 11.923278436493181
    },
    "ambient": {
      "temperature": 21.274693305033583,
      "humidity": 26
    }
  }
}

```

Figure 9-18. View device messages output

You can also SSH to your Raspberry Pi and see the logs there. Let's do that now. The IoT Edge list command will show you all the modules in the device, as shown in Figure 9-19.



```

pi@raspberrypi:~$ iotedge list

```

NAME	STATUS	DESCRIPTION	CONFIG
SendTelemetry	running	Up an hour	apresscr.azurecr.io/sendtelemetry:0.0.2-arm32v7
SimulatedTemperatureSensor:1.0	running	Up 2 hours	mcr.microsoft.com/azureiotedge-simulated-temperature-sens
edgeAgent	running	Up 2 hours	mcr.microsoft.com/azureiotedge-agent:1.0
edgeHub	running	Up 2 hours	mcr.microsoft.com/azureiotedge-hub:1.0

Figure 9-19. IoT Edge list

To see the logs, run the `iotedge logs SendTelemetry` command. Note that the module name is case-sensitive. See Figure 9-20.

```
pi@raspberrypi:~$ iotedge logs SendTelemetry
IoT Hub module client initialized.
Received message: 1, Body: [{"machine":{"temperature":21.221405432546234,"pressure":1.0252234037077987},"ambient":{"temperature":20.584866617845776,"humidity":24},"timeCreated":"2020-08-07T09:13:23.1632881Z"}]
Received message sent
Received message: 2, Body: [{"machine":{"temperature":21.872761375677197,"pressure":1.0994285111530984},"ambient":{"temperature":21.102864242905223,"humidity":26},"timeCreated":"2020-08-07T09:13:28.7604971Z"}]
Received message sent
Received message: 3, Body: [{"machine":{"temperature":22.385136688191043,"pressure":1.1578003821989795},"ambient":{"temperature":21.166818618619264,"humidity":24},"timeCreated":"2020-08-07T09:13:33.7850087Z"}]
```

Figure 9-20. Send Telemetry module logs

Summary

Wow, this was a long chapter, I know. I hope you found it interesting and that you learned a lot about the following topics:

- The prerequisites for creating IoT Edge modules.
- How to set up Visual Studio Code to build IoT Edge Solutions with custom modules?
- How to create an Azure Container Registry?
- How to create a new Azure IoT Edge Solution?
- How modules are connected through routes?
- How to deploy modules to a device?
- How to view messages from a device?

Are you ready for the next chapter? Just grab a coffee or a beer and join me. I will wait for you there.

CHAPTER 10

Azure IoT Central

We are nearing the end of the book. In the previous chapter, you learned a lot about IoT Edge. In this chapter, you will learn about IoT Central, what it is, and why it is needed. If you are ready, let's move on!

Azure IoT Central

With IoT projects, it is all about connecting things and handling the device's data. Once we get the data, we display it somewhere so we can easily check it every time. This can be a web application using Angular, React JS, Vue JS, or ASP.NET MVC, and so on. To get real-time updates in a web application, we can use the Azure Signal R service. But in this chapter, we will be using an easier option so that we don't have to worry about the web application. We will use the SaaS offering from Microsoft, which is Azure IoT Central.

What Is Azure IoT Central

Azure IoT Central is an SaaS offering from Microsoft that reduces the cost of developing, managing, and maintaining enterprise-grade IoT solutions. It comes with a Web UI that allows you to monitor device conditions, create rules, and manage millions of devices easily. There are over 30

Azure services integrated with Azure IoT Central, which makes it very user friendly. In the end, from the user's perspective, everything you need is there. Are you excited to build one?

IoT Hub vs. IoT Central

Though IoT Central was built on the PaaS offering IoT Hub, there are certain differences between them:

- Azure IoT Hub is a PaaS offering that can connect millions of devices securely and scale, whereas IoT Central is an SaaS offering that can connect, manage, and monitor devices at scale. We get an additional dashboard with a customizable UI. We will discuss the functionalities of this UI in the coming sections.
- The device-provisioning service setup is required separately in IoT Hub, but IoT Central has a built-in device-provisioning service.
- IoT Hub provides built-in event hub, service bus queue, service bus topic, and storage endpoints to export the data. Users can also use the message routing for the same functionality. IoT Central provides data exports to Azure Blob Storage, Azure Event Hubs, and Azure Service Bus.

Creating an IoT Central Application

Creating an IoT Central application is as easy as creating any other service in Azure. Go to <https://azure.microsoft.com/en-au/services/iot-central/> and click the Build a Solution button. This will redirect you to <https://apps.azureiotcentral.com>, where you can log in with your Microsoft account. You should see some ready-to-use templates available for some industries, as shown in Figure 10-1.

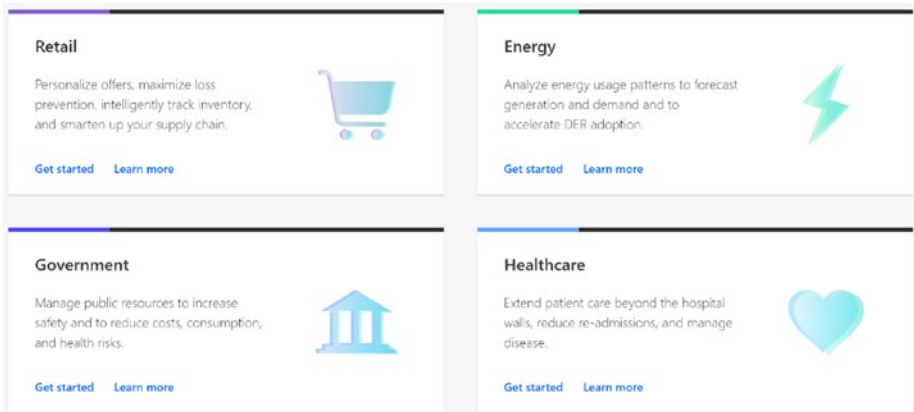


Figure 10-1. Industry-based IoT Central solutions

Let's create a custom application. To do this, just click the Create a Custom App button. You can also do this by clicking the Burger menu, and then choosing the Build menu item. You should see a custom template, as shown in Figure 10-2. Just click it.

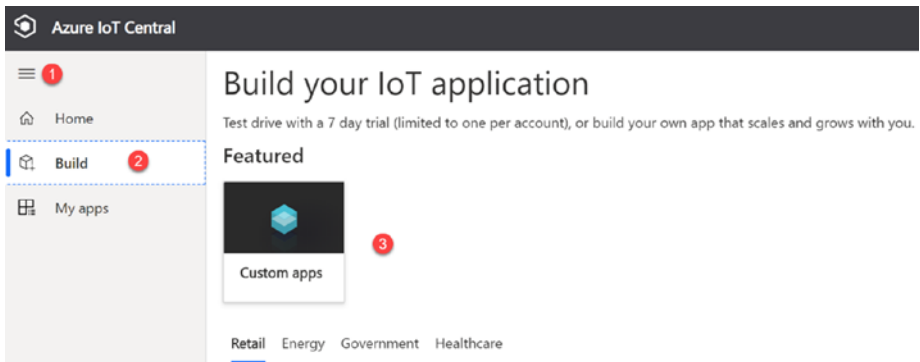


Figure 10-2. Custom IoT Central solutions

You should be given a form that you need to fill out with the application name, pricing plan, contact info, and so on. You can choose the Free plan, which will give you access for seven days with no commitments. You will also be given a chance to convert your plan to a paid one.

Keep in mind that you will not be able to recover your application once the trial period is over. You will have to create a new application in that case, and if you choose to go with any plans other than free, you should have a valid Azure subscription. When you are done filling out the form, click the Create button.

It's now time to create a device template. A device template defines the capabilities of a device that connects to Azure IoT Central. Click the Device templates menu under App Settings, as shown in Figure 10-3.

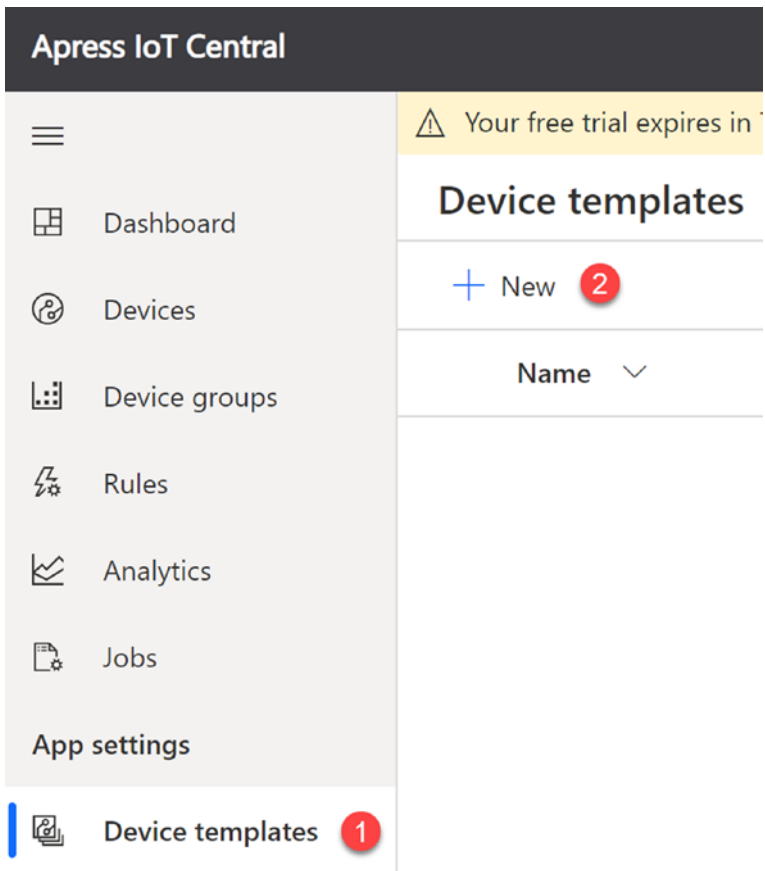


Figure 10-3. Device templates

Clicking the +New button will start the process of setting up the device template. The first step will be to select the device type. Select the IoT device template type, as shown in Figure 10-4.

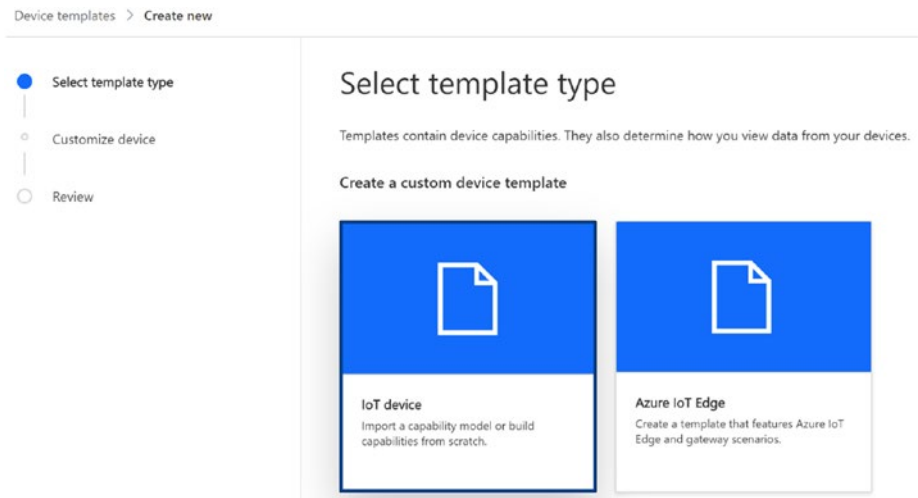


Figure 10-4. Select the template type

Click the Next: Customize button and give your device template a proper name. I call this one Raspberry Pi. Click the Next: Review button and then click the Create button.

The next step is to create a *capability model*. Click the Custom tab on the next screen, as you are going to build the capability model from scratch. You can also import the capability model if you already have one. You will see how to export a capability model in the coming sections. See Figure 10-5.

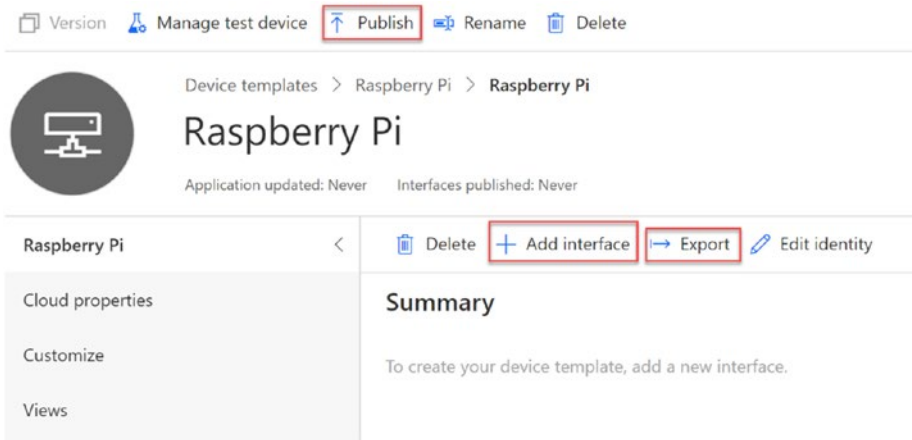


Figure 10-5. Create a custom template

Now you can add the specifications to the new device template. To do so, click the +Add interface button and then select Custom, as shown in Figure 10-6.

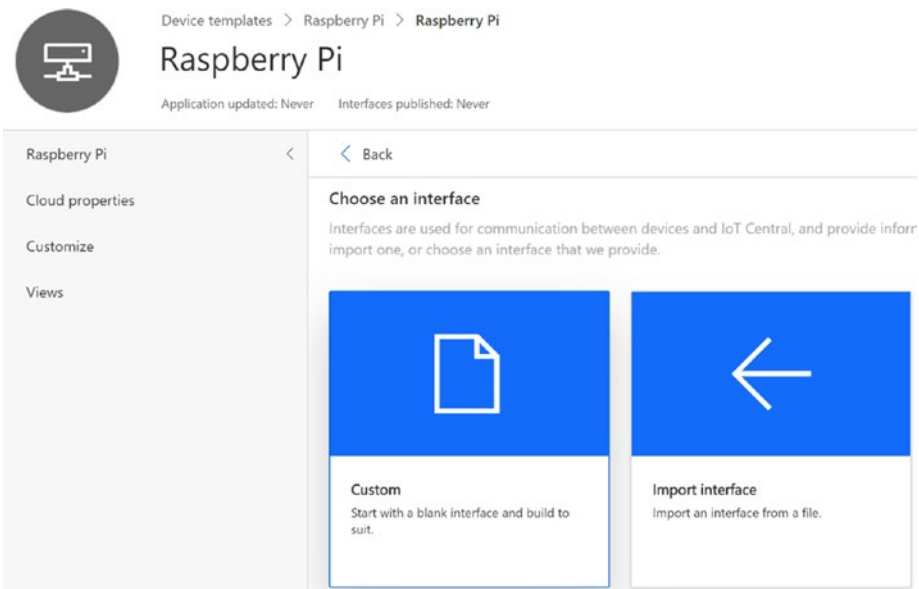


Figure 10-6. Custom interface

An interface must define the capabilities of a device. Let's start building from the blank interface. On the next screen, we will be adding the capabilities to our interface. One of the capabilities of our device is that it needs to send temperature data. Click the +Add Capability button and provide the details on the form, as shown in Figure 10-7.

The screenshot shows the 'Capabilities' configuration page in Azure IoT Central. At the top, there is a navigation bar with buttons for 'Save', '+ Add capability', 'Edit identity', 'Version', 'Export', and 'Delete'. A red arrow points to the '+ Add capability' button. Below the navigation bar, the page title is 'Capabilities' and the status is 'Draft'. A descriptive paragraph explains the purpose of the form. The form itself contains several input fields: 'Display name' (Temperature), 'Name' (Temperature), 'Capability type' (Telemetry), 'Semantic type' (Temperature), 'Schema' (Double), 'Unit' (°C), 'Display unit', 'Comment', and 'Description' (Temperature data).

Figure 10-7. Add capability

A *property* of a device is a constant value. It will be sent to the IoT Central application when the communication is initiated. In this case, I placed my Raspberry device in my office. I have only one office in my home, so I can call the property `OfficeRoom`. In a real-world scenario, imagine that your device will send the location of your car, and in that case, you can create the property with your car number, which is unique. To add the property, click +Add Capability again and set the capability type to property, as shown in Figure 10-8.

The screenshot shows the 'Add property' configuration page in Azure IoT Central. The form fields are: 'Display name' (Room), 'Name' (Room), 'Capability type' (Property), 'Semantic type' (None), 'Schema' (String), 'Writable' (checked), 'Unit' (None), 'Display unit', 'Comment', and 'Description'.

Figure 10-8. Add property

This property can also be configuration data; for example, in our initial application, we set the maximum temperature value for the alerts. This value can be configured as a property. As this value is not a constant, the property value will be changed. This kind of property is called “writable.”

A *command* is sent by the operator of the IoT Central application to the remote device. The only difference between a writable property and a command is that the writable property is limited to a single value, whereas the command can contain any number of input fields. Imagine that your device is in a car, sending location data, and the IoT Central application can send the command to the device to make you remember to stop at a location. Or it can send a command to take a picture. If you turn on the Request option, you can add more data to the command. I hope you got the idea. Click the +Add Capability and set the capability type to command, as shown in Figure 10-9.

The screenshot shows the configuration interface for adding a command in Azure IoT Central. At the top, there are three input fields: a text box containing "Take the picture", another text box containing "TakeThePicture", and a dropdown menu set to "Command". Below these is a light gray configuration panel with several sections:

- Command type:** A dropdown menu set to "Synchronous".
- Request:** A toggle switch that is turned on (blue).
- Display name:** A text box containing "Image type" with a close button (X).
- Name:** A text box containing "ImageType".
- Schema:** A dropdown menu set to "String" with a "Define" link and icon.
- Unit:** A dropdown menu set to "None".
- Response:** A toggle switch that is turned off (gray).

There are also empty text boxes for "Comment" and "Description" in the top right of the configuration panel, and another empty "Comment" box at the bottom right.

Figure 10-9. Add command

When you are done, click the Save button. Once the capability is added, it is time to create a device visualization view. Click the device template, and then select the newly created device template. Click the Views button. From the given options, select Visualizing the Device, as shown in Figure 10-10.

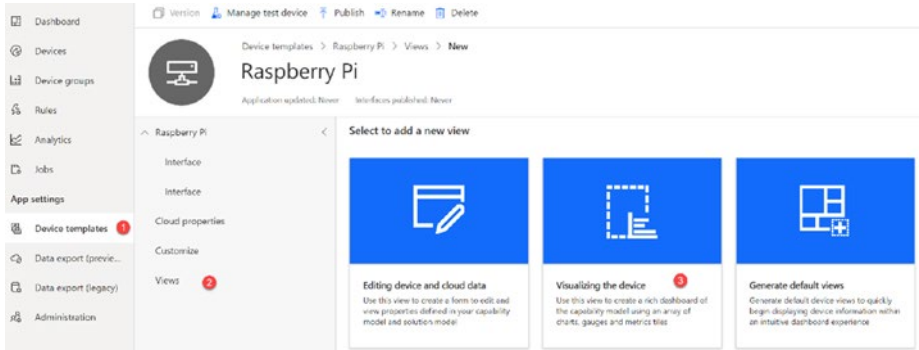






Figure 10-10. *Visualizing the device*

On the next screen, choose the telemetry capability that we created before. Click the Add Tile button and then click Save, as shown in Figure 10-11.

 Save  Delete  Configure preview device

^ View settings

View name * 

^ Add a tile

Drag single elements (like Humidity) onto your view, or select multiple elements from a single category and then click **Add tile**.

^ Telemetry

Create chart, state, event, last known value (LKV), and key performance indicator (KPI) tiles.

Temperature

^ Properties

Create list (grid) or map tiles that display basic information about your device.

Figure 10-11. Add tile

Click the Views and, from the right side, select Editing Device and Cloud Data, as shown in Figure 10-12.

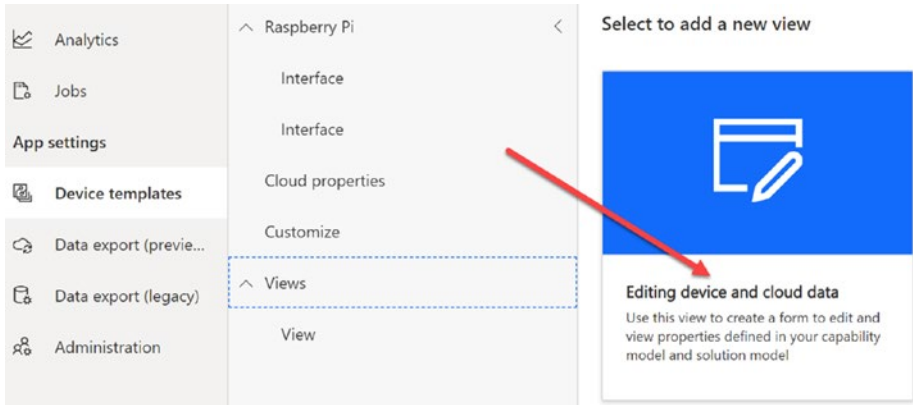







Figure 10-12. *Editing device and cloud data*

From the given form, select the properties that you created and then click the Add Section button. Then click the Save button. See Figure 10-13.

 Save  Delete


Form name * 

Page layout 

1 column layout 

^ Property

Room

 Cloud properties

No capabilities available

[Add section](#)

Figure 10-13. *Select properties*

When you are done, you can publish the device template; you can connect the device only after publishing the device template. It is worth mentioning that you can make only limited changes to the device capability model after publishing. To modify an interface, you have to create a new one. Click the Publish button on the top, as shown in Figure 10-14.

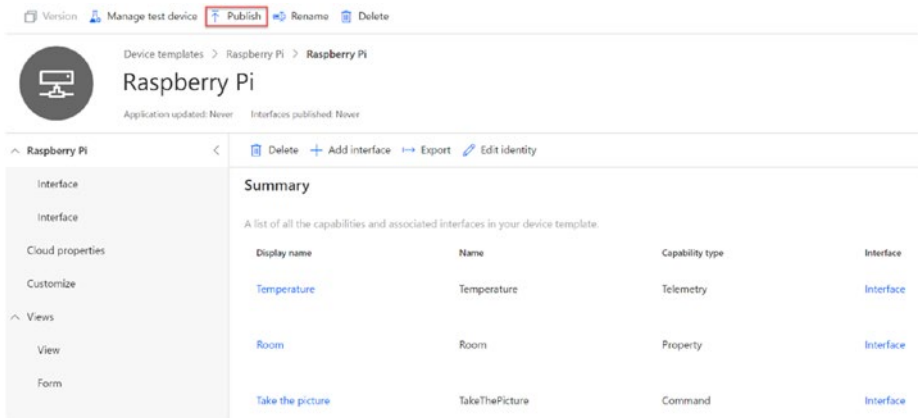


Figure 10-14. Publish the device template

Before publishing, make sure that you don't have an interface that doesn't have any capabilities. Figure 10-15 is an example interface without any capabilities.

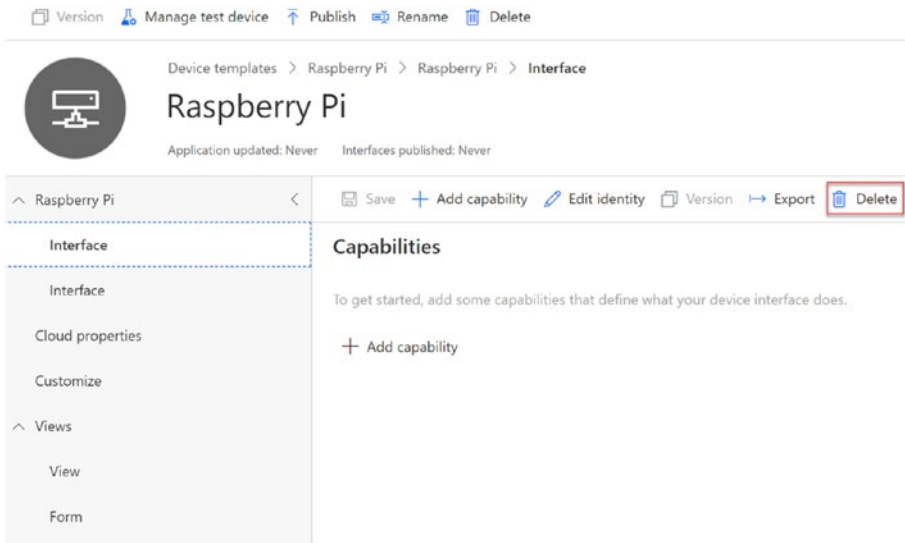


Figure 10-15. *Interface without capabilities*

When the publishing process is finished, you can see that the capabilities are grayed out and you will no longer be able to edit them. You can always export the interface and the device template for future uses. Click the Export button, available at the top of the device template. See Figure 10-16.

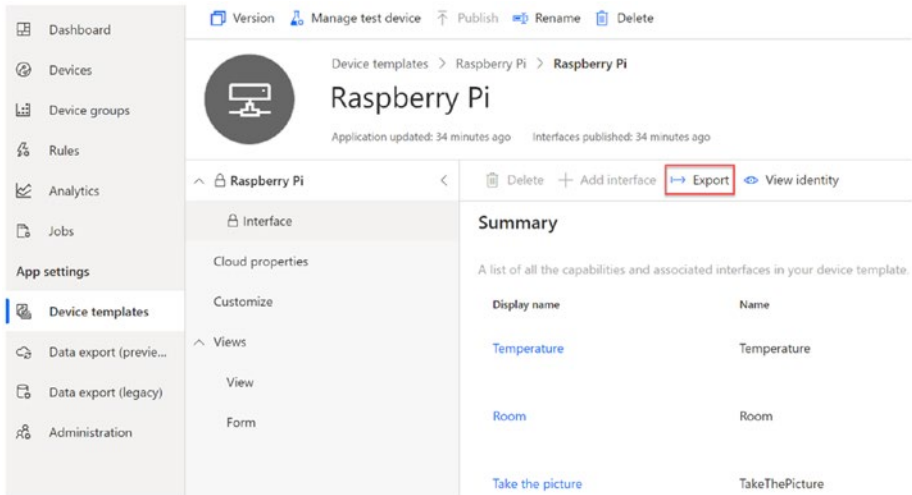


Figure 10-16. Export device template

This will generate a JSON file with your device template name; in my case, it is `Raspberry Pi.json`. Here are the contents of that file:

```
{
  "@id": "urn:apressIotCentral:RaspberryPi_6th:1",
  "@type": "CapabilityModel",
  "implements": [
    {
      "@id": "urn:apressIotCentral:RaspberryPi_6th:t2nevx5f:1",
      "@type": "InterfaceInstance",
      "displayName": {
        "en": "Interface"
      },
      "name": "RaspberryPi_7fp",
      "schema": {
        "@id": "urn:apressIotCentral:RaspberryPi_7fp:1",
        "@type": "Interface",
```



```

    "displayName": {
      "en": "Interface"
    },
    "contents": [
      {
        "@id": "urn:apressIotCentral:RaspberryPi_7fp:
        Temperature:1",
        "@type": [
          "Telemetry",
          "SemanticType/Temperature"
        ],
        "description": {
          "en": "Temperature data"
        },
        "displayName": {
          "en": "Temperature"
        },
        "name": "Temperature",
        "schema": "double",
        "unit": "celsius"
      },
      {
        "@id": "urn:apressIotCentral:RaspberryPi_7fp:Room:1",
        "@type": "Property",
        "displayName": {
          "en": "Room"
        },
        "name": "Room",
        "writable": true,
        "schema": "string"
      }
    ],

```

```

{
  "@id": "urn:apressIotCentral:RaspberryPi_7fp:Take
ThePicture:1",
  "@type": "Command",
  "commandType": "synchronous",
  "durable": false,
  "request": {
    "@id": "urn:apressIotCentral:RaspberryPi_7fp:Take
ThePicture:ImageType:1",
    "@type": "SchemaField",
    "displayName": {
      "en": "Image type"
    },
    "name": "ImageType",
    "schema": "string"
  },
  "displayName": {
    "en": "Take the picture"
  },
  "name": "TakeThePicture"
}
]
}
],
"displayName": {
  "en": "Raspberry Pi"
},
"@context": [
  "http://azureiot.com/v1/contexts/IoTModel.json"
]
}

```

Creating a Device

As you have already created a device template, it is time to create a real device that sends telemetry to IoT Central. Click the devices in the left menu and then click the device template you just created. You should now see a +New button, as shown in Figure 10-17. Click it.

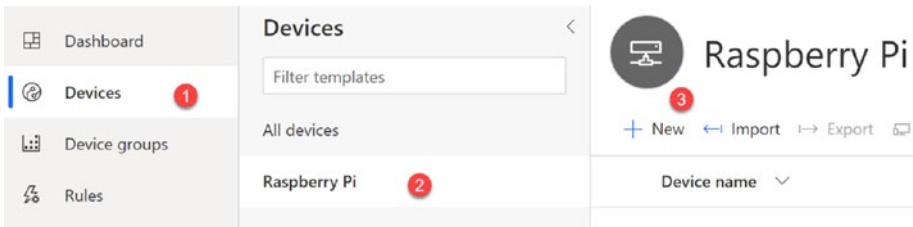




Figure 10-17. Create a device


You will be given a form to enter the details of your device. Make sure that the template type is the same as the one you created recently. The device name is just a unique name; you can be selective here. The device ID is a unique identifier that is used to connect to the device. As we have a device that will send the telemetry data, choose No for the simulated device question. Click the Create button, as shown in Figure 10-18.

Create a new device ×

To create a new device, select a template type, a name, and a unique ID. [Learn more](#) 

Template type *

Device name * 

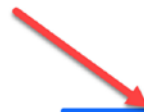
Device ID * 

Simulate this device?

A simulated device generates telemetry that enables you to test the behavior of your application before you connect a real device.

No

* Required



Create

Cancel

Figure 10-18. Device properties

When it is done, a new device will be created with a Registered status.

Getting the Device Connection Keys

We already created a device, so let's get the connection keys. Click the Connect button on the top-right menu. That will open a device connection pop-up. Make a note of these values and remember to keep the connect method set to Shared Access Signature (SAS), as shown in Figure 10-19.

Device connection



ID scope ⓘ

[Redacted ID scope]



Device ID ⓘ

[Redacted Device ID]



Select the connect method for this device instance. You can update later.

Connect method

Shared access signature (SAS) ▾

SAS security tokens are an attestation mechanism for devices to connect to IoT Central. The SAS keys from the default enrollment group are shown below. Use them to register your device with IoT Central. [Click to learn more.](#)

Primary key ⓘ

TTxziL [Redacted Primary key]



Secondary key ⓘ

nMd31U [Redacted Secondary key]



Close

Figure 10-19. Device connection properties

Creating a Device Application That Sends Telemetry to IoT Central

We are going to create a device application that sends the telemetry data to IoT Central. Let's create a new folder called `raspberrypi.net.core.central` and run the following command inside the folder. It will create a .NET Core console application with all of its required packages:

```
dotnet new console --langVersion=latest && dotnet add package Iot.Device.Bindings && dotnet add package Microsoft.Azure.Devices.Client && dotnet add package Microsoft.Azure.Devices.Provisioning.Transport.Mqtt && dotnet add package Microsoft.Azure.Devices.Shared && dotnet add package Newtonsoft.Json && dotnet add package Microsoft.Azure.Devices.Provisioning.Transport.Http
```

When the project is created, open it with VSCode and create an `.env` file with the following variables.

```
IDSCOPE="<Replace with your IoT Central ID Scope>"
CENTRAL_DEVICE_ID="<Replace with your IoT Central Device ID>"
PRIMARY_KEY="<Replace with your IoT Central Primary key>"
```

Don't forget to update the values based on the connection properties that you received. When you are done, you can start writing the program. Open the `Program.cs` file and add the following using statements:

```
using System;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Iot.Device.CpuTemperature;
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.Devices.Provisioning.Client;
```

```
using Microsoft.Azure.Devices.Provisioning.Client.Transport;
using Microsoft.Azure.Devices.Shared;
using Newtonsoft.Json;
```

Now let's get the required values from the .env file and add some other variables.

```
private static string idScope = Environment.
    GetEnvironmentVariable("ID_SCOPE");
private static string centralDeviceId = Environment.
    GetEnvironmentVariable("CENTRAL_DEVICE_ID");
private static string primaryKey = Environment.
    GetEnvironmentVariable("PRIMARY_KEY");
private const string endPoint = "global.azure-devices-
    provisioning.net";
private static CpuTemperature _temperature = new
    CpuTemperature();
private static int _messageId = 0;
```

Before we start writing the Main program, let's write a program that can register the device:

```
private static async Task<DeviceRegistrationResult>
    RegisterDeviceAsync(SecurityProviderSymmetricKey
        security)
    {
        using var transportHandler = new ProvisioningTransp
            ortHandlerMqtt(TransportFallbackType.TcpOnly);
        var provDeviceClient = ProvisioningDeviceClient.
            Create(endPoint, idScope, security,
                transportHandler);
        return await provDeviceClient.RegisterAsync();
    }
```

This method will register the current device using the Device Provisioning Service and assign it to an IoT Hub. As discussed, Azure IoT Central uses IoT Hub in the background. We also need a method that can send telemetry data to IoT Central. Let's write that method:

```
private static async Task SendMessage(DeviceClient
deviceClient, double temperature)
{
    var dataToSend = new Telemetry() { MessageId = ++_
messageId, Temperature = temperature };
    var stringToSend = JsonConvert.
SerializeObject(dataToSend);
    var messageToSend = new Message(Encoding.UTF8.
GetBytes(stringToSend));
    await deviceClient.SendEventAsync(messageToSend).
ConfigureAwait(false);
}
```

This method may look very familiar to you, as it looks the same as the method we created in the initial chapters. We use a property class to serialize the data. Keep in mind that the name of this property class should be `Telemetry`, or else you will see some reporting errors in your IoT Central dashboards because of the model name difference, so this data will fall under unmodeled data.

```
class Telemetry
{
    [JsonPropertyAttribute(PropertyName = "Temperature")]
    public double Temperature { get; set; } = 0;
    [JsonPropertyAttribute(PropertyName = "MessageId")]
    public int MessageId { get; set; } = 0;
}
```


Let's write the Main method now, as we created all the supporting methods.

```
static async Task Main(string[] args)
{
    try
    {
        using var security = new SecurityProvider
            SymmetricKey(centralDeviceId, primaryKey, null);
        var deviceRegistrationResult = await
            RegisterDeviceAsync(security);
        if (deviceRegistrationResult.Status !=
            ProvisioningRegistrationStatusType.Assigned)
            return;
        var auth = new DeviceAuthenticationWithRegistry
            SymmetricKey(deviceRegistration
                Result.DeviceId,
                (security as SecurityProviderSymmetricKey).
                GetPrimaryKey());
        using var _deviceClient = DeviceClient.
            Create(deviceRegistrationResult.AssignedHub,
                auth, TransportType.Mqtt);
        while (true)
        {
            if (_temperature.IsAvailable)
            {
                await SendMessage(_deviceClient,
                    _temperature.Temperature.Celsius);
            }
            Thread.Sleep(3000);
        }
    }
}
```

```

        catch (System.Exception ex)
        {
            Console.WriteLine($"Hm, that's an error: {ex}");
        }
    }
}

```

As you can see in the code, we get the `AssignedHub` from the `Device Registration Result`. Once we get that, we send the telemetry to the Hub, as simple as that. Now, all we have to do is run our application in the Raspberry Pi. To do that, we need to update the `launch.json` and `task.json` files inside the `.vscode` file.

Launch.json file:

```

{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug Publish, Launch, and Attach Debugger",
            "type": "coreclr",
            "request": "launch",
            "envFile": "${workspaceFolder}/.env",
            "preLaunchTask": "DebugPublish",
            "program": "~/${workspaceFolderBasename}/${workspaceFolderBasename}",
            "cwd": "~/${workspaceFolderBasename}",
            "stopAtEntry": false,
            "console": "internalConsole",
            "pipeTransport": {
                "pipeCwd": "${workspaceRoot}",
                "pipeProgram": "/usr/bin/ssh",
                "pipeArgs": [
                    "pi@192.168.0.80"
                ],
            },
        },
    ],
}

```

```

        "debuggerPath": "~/vsdbg/vsdbg"
    }
},
{
    "name": "Release Publish, Launch, and Attach
Debugger",
    "type": "coreclr",
    "request": "launch",
    "preLaunchTask": "ReleasePublish",
    "program": "~/${workspaceFolderBasename}/${workspace
FolderBasename}",
    "cwd": "~/${workspaceFolderBasename}",
    "stopAtEntry": false,
    "console": "internalConsole",
    "pipeTransport": {
        "pipeCwd": "${workspaceRoot}",
        "pipeProgram": "/usr/bin/ssh",
        "pipeArgs": [
            "pi@192.168.0.80"
        ],
        "debuggerPath": "~/vsdbg/vsdbg"
    }
}
]
}

```

As you can see, we use two tasks in the `launch.json` file. Let's add them to the `task.json` file.

```

{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "DebugPublish",
      "command": "sh",
      "type": "shell",
      "problemMatcher": "$msCompile",
      "args": [
        "-c",
        "\dotnet publish -r linux-arm -c Debug -o ./bin/
        linux-arm/publish ./${workspaceFolderBasename}.
        csproj && rsync -rvuz ./bin/linux-arm/publish/ pi
        @192.168.0.80:~/${workspaceFolderBasename}\"",
      ]
    },
    {
      "label": "ReleasePublish",
      "command": "sh",
      "type": "shell",
      "problemMatcher": "$msCompile",
      "args": [
        "-c",
        "\dotnet publish -r linux-arm -c
        Release -o ./bin/linux-arm/publish
        ./${workspaceFolderBasename}.csproj && rsync -rvuz
        ./bin/linux-arm/publish/ pi@192.168.0.80:~/${wo
        rkspaceFolderBasename}\"",
      ]
    }
  ]
}

```

Open your solution in VSCode. Press F1 and type and select Remote-WSL: Reopen Folder in WSL. Now all you have to do is press F5 and wait to see the magic. If you don't see any errors in the console, you are good to go. Congrats. Now go to your IoT Central application and find the telemetry data there. Click the device that you created. You should now see that the device status is "provisioned," shown as in Figure 10-20.

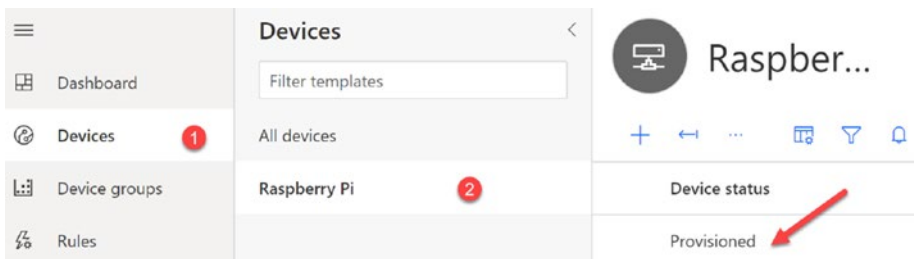


Figure 10-20. Device is provisioned

Click the device name from the grid and go to the View tab. You can see the telemetry data in a graph there, as shown in Figure 10-21.

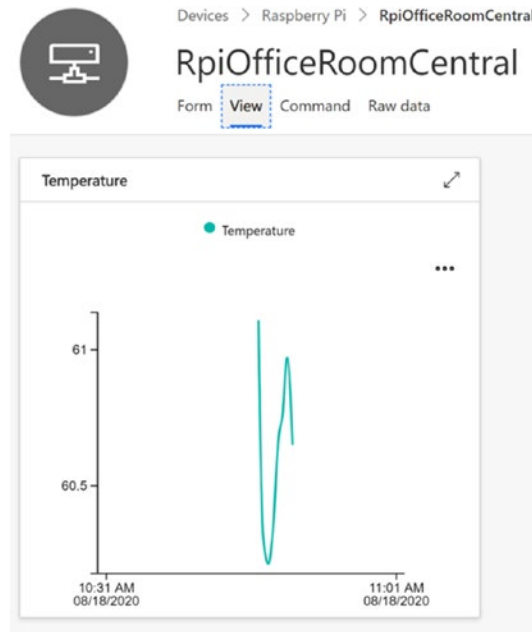


Figure 10-21. Telemetry data in a graph

You should also be able to see the raw data if you click the Raw Data tab.

Test Property and Command

Remember that we created a writable property and a command when we created the device template? Now, we will work on that. The first thing is to create a handler for our writable property, Room. Keep in mind that this is just a dummy scenario, and you can think of any real-time scenarios and implement them the same way. Add the following code to the Main method.

```
_deviceClient.SetDesiredPropertyUpdateCallbackAsync(HandleSettingChanged, null).GetAwaiter().GetResult();
    Console.WriteLine("Done");
```

As you may have already guessed, we need to create the handler now.

```
static async Task HandleSettingChanged(TwinCollection
desiredProperties, object userContext)
{
    var setting = "Room";
    if (desiredProperties.Contains(setting))
    {
        var roomChange = reportedProperties[setting] =
desiredProperties[setting];
    }
    await _deviceClient.UpdateReportedPropertiesAsync(r
eportedProperties);
}
```

We have also made a few other changes.

- Added a private variable for DeviceClient
- Used the newly created device client variable
- Updated the SendMessage method

In the end, this is how your Program.cs file should look:

```
using System;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Iot.Device.CpuTemperature;
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.Devices.Provisioning.Client;
using Microsoft.Azure.Devices.Provisioning.Client.Transport;
using Microsoft.Azure.Devices.Shared;
using Newtonsoft.Json;
```

```
namespace raspberrypi.net.core.central
{
    public class Program
    {
        private static string idScope = Environment.
        GetEnvironmentVariable("ID_SCOPE");
        private static string centralDeviceId = Environment.
        GetEnvironmentVariable("CENTRAL_DEVICE_ID");
        private static string primaryKey = Environment.
        GetEnvironmentVariable("PRIMARY_KEY");
        private const string endPoint = "global.azure-devices-
        provisioning.net";
        private static CpuTemperature _temperature = new
        CpuTemperature();
        private static TwinCollection reportedProperties = new
        TwinCollection();
        private static int _messageId = 0;
        private static DeviceClient _deviceClient;
        static async Task Main(string[] args)
        {
            try
            {
                using var security = new SecurityProviderSymmet
                ricKey(centralDeviceId, primaryKey, null);
                var deviceRegistrationResult = await
                RegisterDeviceAsync(security);
                if (deviceRegistrationResult.Status !=
                ProvisioningRegistrationStatusType.Assigned)
                    return;
                var auth = new DeviceAuthenticationWithRegistry
                SymmetricKey(deviceRegistrationResult.DeviceId,
```



```

        (security as SecurityProviderSymmetricKey).
        GetPrimaryKey());
        _deviceClient = DeviceClient.
        Create(deviceRegistrationResult.AssignedHub,
        auth, TransportType.Mqtt);
        _deviceClient.SetDesiredPropertyUpdateCallback
        Async(HandleSettingChanged, null).GetAwaiter().
        GetResult();
        Console.WriteLine("Done");

        await SendMessage(_temperature.Temperature.
        Celsius);
    }
    catch (System.Exception ex)
    {
        Console.WriteLine($"Hm, that's an error: {ex}");
    }
}

private static async Task SendMessage(double temperature)
{
    while (true)
    {
        if (_temperature.IsAvailable)
        {
            var dataToSend = new Telemetry() {
                MessageId = ++_messageId, Temperature =
                temperature };
            var stringToSend = JsonConvert.
                SerializeObject(dataToSend);
            var messageToSend = new Message(Encoding.
                UTF8.GetBytes(stringToSend));

```

```

        await _deviceClient.SendEventAsync
            (messageToSend).ConfigureAwait(false);
    }
    Thread.Sleep(3000);
}
}

private static async Task<DeviceRegistrationResult> RegisterDeviceAsync(
    SecurityProviderSymmetricKey security)
{
    using var transportHandler = new Provisioning
        TransportHandlerMqtt(TransportFallbackType.TcpOnly);
    var provDeviceClient = ProvisioningDeviceClient.
        Create(endPoint, idScope, security, transportHandler);
    return await provDeviceClient.RegisterAsync();
}

static async Task HandleSettingChanged(TwinCollection
    desiredProperties, object userContext)
{
    var setting = "Room";
    if (desiredProperties.Contains(setting))
    {
        var roomChange = reportedProperties[setting] =
            desiredProperties[setting];
    }
    await _deviceClient.UpdateReportedPropertiesAsync(
        reportedProperties);
}
}
}

```

```

class Telemetry
{
    [JsonPropertyAttribute(PropertyName = "Temperature")]
    public double Temperature { get; set; } = 0;
    [JsonPropertyAttribute(PropertyName = "MessageId")]
    public int MessageId { get; set; } = 0;
}
}

```

Now run your application and go to the IoT Central device section. On the device page, go to the Form tab and type any value in the given textbox. Then click the Save button. If you have enabled a debugger in the handler method, you should now see the values, as shown in Figure 10-22.

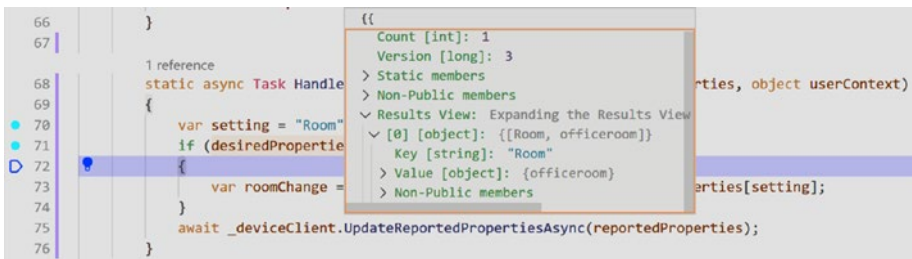
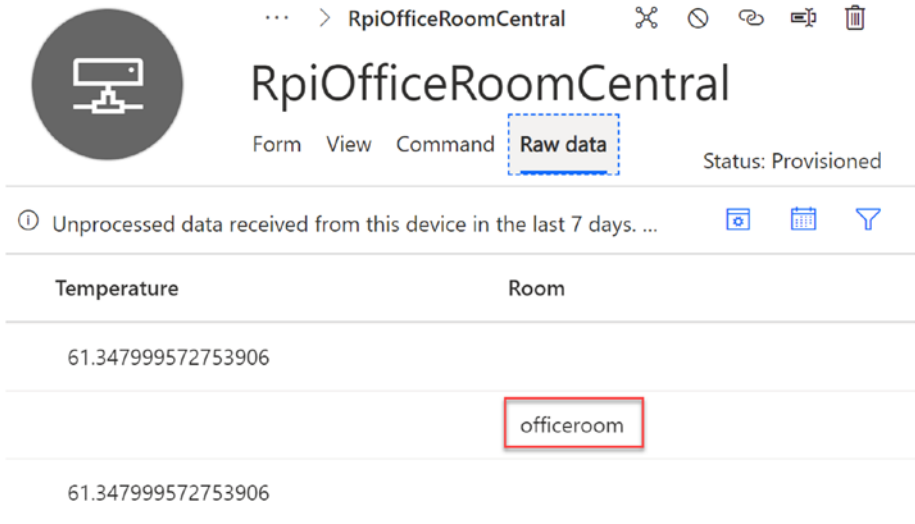


Figure 10-22. Writable property value

Go back to the Raw Data tab on your device page; there should be an entry for the property, as shown in Figure 10-23.



... > RpiOfficeRoomCentral

RpiOfficeRoomCentral

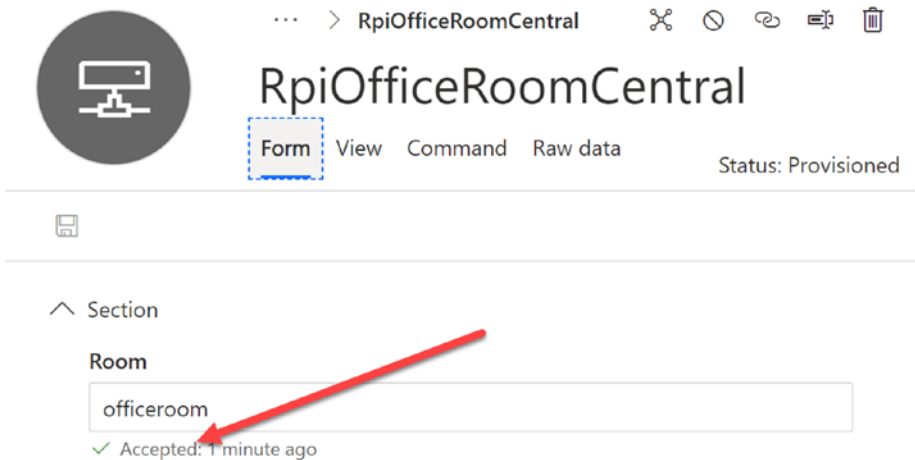
Form View Command **Raw data** Status: Provisioned

Unprocessed data received from this device in the last 7 days. ...

Temperature	Room
61.347999572753906	officerroom
61.347999572753906	

Figure 10-23. Writable property record

You should also see a message on the Form tab saying that the property value was accepted, as shown in Figure 10-24.



... > RpiOfficeRoomCentral

RpiOfficeRoomCentral

Form View Command Raw data Status: Provisioned

Section

Room

officerroom

✓ Accepted: 1 minute ago

Figure 10-24. Form tab message

Let's implement the `TakeThePicture` command now. As before, this is just a dummy command to show you the flow. We need a direct method handler to do that. Add the following code in the `Main` method, just before the method to send the telemetry data.

```
_deviceClient.SetMethodHandlerAsync("TakeThePicture",
CommandTakeThePicture, null).Wait();
```

And a handler method.

```
private static Task<MethodResponse> CommandTakeThePicture(
    MethodRequest methodRequest, object userContext)
{
    // Get the data from the payload
    var payload = Encoding.UTF8.GetString(
        methodRequest.Data);
    Console.WriteLine(payload);
    // Code to take the picture
    // Save in the given format
    // Return the image URL
    // Imagine that your device is setup with a camera
    // Acknowledge the direct method call
    string result = "{\"result\": \"Executed : \" +
        methodRequest.Name + "\"}";
    return Task.FromResult(new MethodResponse(Encoding.
        UTF8.GetBytes(result), 200));
}
```

Now go back to the `Command` tab on the device page and enter a value `PNG` in the textbox. Click the `Run` button. You can also put a debugger on the direct method so that you can debug the values. See [Figure 10-25](#).

```

46     private static Task<MethodResponse> CommandTakeThePicture(MethodRequest methodRequest, object userContext)
47     {
48         // ← "\png\" from the payload
49         var payload = Encoding.UTF8.GetString(methodRequest.Data);
50         Console.WriteLine(payload);
51         // Code to take the picture
52         // Save in the given format
53         // Return the image URL
54         // Imagine that your device is setup with a camera
55         // Acknowledge the direct method call
56         string result = "{\result\": \"Executed : \" + methodRequest.Name + \"\"}";
57         return Task.FromResult(new MethodResponse(Encoding.UTF8.GetBytes(result), 200));
58     }

```

Figure 10-25. *The Debug command*

You can also see the command's run history in the portal. Go to the Command tab and click the Command History link, as shown in Figure 10-26.

Interface / Take the picture

Image type

Run

To see response, please check the [command history](#).

History - Take the picture

RESPONSE	1 minute ago
SENT	1 minute ago
SENDING	1 minute ago

png

sibeesh.venu@gmail.com

Figure 10-26. *The Command history link*

Wow, you just handled your device remotely. Wasn't that cool?

Summary

I enjoyed writing this chapter, as it involves a lot of hands-on work. I hope you felt the same. In this chapter, you learned about the following topics:

- What is Azure IoT Central?
- What are the differences between Azure IoT Hub and Azure IoT Central?
- How to create an Azure IoT Central application?
- How to create a device in Azure IoT Central?
- How to create a .NET Core application to send telemetry data to Azure IoT Central?
- How to remotely handle a device from Azure IoT Central?

Summary

Thanks a lot for being with me this far. I appreciate that. I strongly believe that you could learn something from this book and you will be able to build your own IoT solution with Microsoft Azure. Albert Einstein once said, "Learning is experience. Everything else is just information." So, now is the time to go build your IoT applications. I wish you all the very best. You can always share your feedback about this book via email at sibeesh.venu@gmail.com.

Index

A, B

Azure Cloud Shell

- advanced settings, 65, 66
- command, 67
- deletion, 67
- login, 66
- select directory, 64, 65
- storage account, 65

Azure Container Registry

- admin access, 158
- capabilities/differences, 156
- creation, 156, 157

Azure IoT Central

- Add Capability, 197
- Add command, 198
- Add property, 197
- Add Tile, 199, 200
- capability model, 195
- connection keys, 209, 210
- creating device, 208, 209
- custom interface, 196
- custom template, 195, 196
- definition, 191
- device application (*see* Device application, Azure IoT Central)
- device templates, 194

- Editing Device/Cloud Data, 201
- export device template, 204, 205
- Free plan, 193
- industries, 192, 193
- interface, 203, 204
- vs.* IoT Hub, 192
- JSON file, 205–207
- publish device template, 203
- select properties, 201, 202
- template type, 195
- Visualizing Device, 199

Azure IoT Hub, 192

- Azure Portal (*see* Azure Portal, IoT Hub)
- Cloud Shell (*see* Azure Cloud Shell)
- custom event message
 - properties, 85, 86
- definition, 63
- Raspberry Pi (*see* Raspberry Pi)
- registering
 - create device, 74, 75
 - IoT Devices menu, 74

Azure IoT tools, 95

- first page, 81, 82
- installation, 81
- IoT Hub

INDEX

Azure IoT tools (*cont.*)

device monitoring, [84, 85](#)

list, [82](#)

selection, [83](#)

options, [84](#)

Azure Portal, IoT Hub

bi-directional

communication, [69](#)

creation, [67](#)

Device-to-Cloud-Partitions, [71](#)

messages, [71](#)

networking, [68, 69](#)

resources/resource groups, [72](#)

review/create, [73](#)

select region, [68](#)

size/scale, [69, 71](#)

tags, [72](#)

tier capabilities, [69, 70](#)

Azure Security Center, [71](#)

C

callback function, [113, 117](#)

Cloud to Device Communication

backend application, [121](#)

demo application, [125, 127](#)

device-specific endpoint, [120](#)

direct methods, [96–101, 103](#)

IOT Hub, [96](#)

receiving feedback, [124](#)

scenarios, [95](#)

SendCloudToDevice

MessageAsync, [121, 122](#)

sending feedback, [122, 124](#)

twin's desired

properties, [104–107](#)

CompleteAsync() function, [124](#)

Connection, setup

hostname, [37](#)

IPV6 address, [37](#)

.NET debugger, [39](#)

SSH certificate, [38](#)

tasks, [36](#)

WSL1, [36](#)

D

Desired property

codeblock, [108](#)

device application, [108](#)

Main Method, [112, 113](#)

OnDesiredProperty

ChangedAsync method

debugger, [118–120](#)

Program.cs class, [114–117](#)

Registry Manager, [111](#)

service connect/registry read

permissions, [109, 110](#)

telemetryConfig, [113](#)

Device application, Azure IoT

Central

.env file, [211](#)

launch.json file, [215, 216](#)

Main method, [214, 215](#)

packages, [211](#)

Program.cs file, [211](#)

provisioned, [218](#)

register device, [212](#)

- task.json file, 215–217
- telemetry data, 213, 218, 219
- unmodeled data, 213
- update values, 211
- variables, 212

Device Provisioning

- Service, 70, 192, 213

E, F, G, H

Reported property, 107, 108, 113, 120

I, J, K

IoT Edge

- creating device, 132, 134
- definition, 129
- deploying module, 142, 144–150
- Linux systems, installing,
 - 134–136, 138, 140–142
- runtime, 130, 131
- viewing sent images, 151, 152

IoT Edge modules

- armv7l, 175
- Azure CLI, 174
- Azure container registry
 - repositories, 180
- Build/Push, 174
- change version number, 182, 183
- command, 173
- connectionModuleId, 170
- default architecture, 167, 168
- default platform, 168
- deployment.arm32v7.json, 177

- deployment.debug.template.
 - json, 160–167
- deployment, device
 - backoff, 187
 - Container registry
 - credentials, 185, 186
 - creation, 184, 185
 - modules running, 186
 - output, 185
- deployment.template.json
 - file, 168
- Dockerfile.arm32v7 file, 175, 176
- Docket Desktop, 154
- \$edgeHub, 168
- .env file, 160, 167, 177–179
- .gitignore file, 167
- image repository, 159
- Init method, 171
- install Git, 154
- module.json file, 159, 160
- nested virtualization, 153
- new docker image, 183, 184
- operations, 175
- PipeMessage method, 172, 173
- possible source
 - properties, 169, 170
- raspberrypi.edge, 158
- route syntax, 169
- sendtelemetry
 - repository, 181, 182
- SetInputMessageHandler
 - Async, 171
- SimulatedTemperature
 - Sensor, 169

INDEX

IoT Edge modules (*cont.*)

- sink property values, 171

- viewing device

 - messages, 187, 188

 - list, 189

 - output, 188, 189

 - sendtelemetry module

 - logs, 190

- VSCode, 154, 155

IoT Edge Security Daemon, 135

L, M

Linux distribution, 34–36

N

.NET Core, 29, 30, 40–42

- C# extension, 46

- chdir, 44

- code options, 46

- debug window console, 60

- dummy application, VSCode, 43

- folder structure, 45

- launch.json, 47

- ms-dotnettools.charp

 - extension, 58, 59

- name attribute, 52, 53

- obj., 45

- Program.cs file, 47

- rewriting application, 51

- rsync attribute, 53–56

- type attribute, 53

- VSCode variables, 56–58

- WSL, 48–50

O, P, Q

OnDesiredPropertyChangedAsync

- function, 117

R

Raspberry Pi

- accessories, 5–10

- arithmetic/logical operations, 1

- B model, 5

- change password warning

 - window, 26

- checking connection, 23

- command, 76

- command tool, 24, 25

- configuration window, 25

- connection string, 80

- .csproj file, 76

- development/deployment, 19

- DeviceClient, 79

- DeviceData class, 76–78

- history, 2, 3

- IoT Devices list, 79

- IoT Devices properties, 79, 80

- package, 76

- password changed, 27

- Publish/Debug task, 80

- send data, 80, 81

- SSH, 19–21

- version 4, 3

- Wi-Fi configuration, 22

Raspberry Pi imager, 13–16

Raspbian

- downloading image/writing,
 - 16, 17
 - NOOBS, 17
 - operating system, 10
 - ReceiveAsync() function, 124

S

- SendCloudToDevice
 - MessageAsync(), 121
- SendEventAsync method, 79, 173
- SentToIoTHub method, 79
- SetInputMessageHandlerAsync
 - method, 171, 172
- Shared Access Signature (SAS), 209

T, U

- TakeThePicture command
 - debug command, 226, 227
 - handler method, 226
 - history link, 227
 - Main method, 226

V

- Visual Studio Code
 - (VSCode), 30, 153, 184

W, X, Y, Z

- Windows 10 IoT, 11, 12
- Windows Subsystem for
 - Linux (WSL), 48
 - installation, 32, 33
 - key points, 31
 - vs.* WSL2, 31, 32
- Windows Terminal
 - command-line tools, 87
 - configuration, 89, 90
 - emojis/icons, 89
 - font weight
 - support, 91
 - installation, 88
 - multiple tabs, 88
 - open folders, 91
 - open profile, 91
 - rename tab, 93
 - tab color, 92
- Writable property
 - changes, 220
 - create handler, 220
 - Main method, 219
 - Program.cs file, 220–224
 - record, 224, 225
 - tab message, 225
 - value, 224