**Chapter 1. Develop Azure Infrastructure as a service compute solution**

Today, cloud computing is a consolidated reality that any company or professional should consider when developing or maintaining new or existing products. When you are planning for developing or deploying an application, you can choose between two main models of cloud services, Infrastructure as a Service (IaaS) or Platform as a Service (PaaS), and each model has its pros and cons. If you decide to use the IaaS model, you have more granular control over the infrastructure that will support your application.

However, once the deployment in the production environment has finished, you need to maintain it. This maintenance means that you also need to allocate the budget for the support of the infrastructure, and you must have trained staff for conducting this maintenance.

Thanks to cloud technologies, you can drastically reduce these infrastructure planning and deployment requirements by deploying your software on a managed service known as Platform as a Service (PaaS). Doing so means you only need to worry about your code and how it interacts with other services in Azure. PaaS products such as Azure App Service or Azure Functions releases you from worrying about highly available or fault-tolerant configurations because the service provided by Azure already manages these things.

This chapter reviews how to work with the options that Azure makes available to you for developing your solutions based on the IaaS model. The chapter also covers the PaaS solutions that Azure provides, which allow you to focus on your code and forget about the underlying infrastructure.

**IMPORTANT HAVE YOU READ PAGE XVII?**

It contains valuable information regarding the skills you need to pass the exam.

**Skills covered in this chapter:**

* [Skill 1.1: Implement solutions that use virtual machines (VM)](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev1sec1)
* [Skill 1.2: Create Azure App Service web apps](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev1sec2)
* [Skill 1.3: Implement Azure Functions](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev1sec3)

**Skill 1.1: Implement solutions that use virtual machines (VM)**

One of the main characteristics of the IaaS model is the higher level of control that it offers when deploying the infrastructure needed for your application. Typically, you need to work with this model because you need more control over the different elements of your application. Using IaaS, you deploy your virtual machines, where you will implement all the components required for your solution.

Azure provides you with all the underlying hardware and configuration needed for your virtual machine (VM) to run correctly. However, you still need to manage all administrative tasks related to the VM’s operating system, such as installing operating system upgrades or security patches. Microsoft manages the configuration required for providing the fault tolerance for the physical hardware that supports your VM. But if you need your application or software solution to be highly available, you have to manage the configuration of the VMs that host your application.

**This skill covers how to**

* [Provision VMs](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec1)
* [Configure VMs for remote access](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec2)
* [Create ARM templates](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec3)
* [Create container images for solutions by using Docker](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec4)
* [Publish an image to the Azure Container Registry](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec5)
* [Run containers by using Azure Container Instance](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec6)

**Provision VMs**

Deploying a VM in Azure is a straightforward process, but you still need to think about some key points if you want to achieve the best balance between the costs and your requirements. Perhaps the most obvious decision is which operating system you should use. The good news is that Azure fully supports Windows, Windows Server, and the principal distributions of Linux.

***NOTE* SUPPORTED OPERATING SYSTEMS**

You can review the full list of supported operating systems to be used in Azure VMs at the following URLs:

* Windows: [*https://support.microsoft.com/en-us/help/2721672/microsoft-server-software-support-for-microsoft-azure-virtual-machines*](https://support.microsoft.com/en-us/help/2721672/microsoft-server-software-support-for-microsoft-azure-virtual-machines)
* Linux: [*https://docs.microsoft.com/en-us/azure/virtual-machines/linux/endorsed-distros*](https://docs.microsoft.com/en-us/azure/virtual-machines/linux/endorsed-distros)

All these Windows and Linux OSes are preinstalled and available to you in the Azure Marketplace as VM Images. Apart from these default VM images, you will also find other images in the marketplace from other vendors containing preconfigured solutions that may better match your needs.

Once you have chosen your operating system, you need to decide other essential aspects of the VM:

* **Name** Enter the name of the VM. Names may be up to 15 characters long.
* **Location** Select the geographical region where you deploy your VM. Azure has several data centers distributed across the globe that are grouped in geographical regions. Choosing the wrong region or location may have adverse effects.
* **Size** Designate the number of resources that you will assign to your virtual machines. These resources include the amount of memory, processing power, number of virtual network interface cards (NICs) that you can attach to your VM, and total storage capacity that will be available for your VM.
* **Limits** Every subscription has default quota limits. These limits can affect you when deploying new virtual machines. By default, each subscription has a limit of 20 VMs per region. However, you can increase this limit by contacting Azure’s support service.
* **Extensions** Extensions enable you to automate some tasks or configuration once the deployment of your VM completes successfully. Some of the most common extensions are
  + Run custom scripts
  + Deploy and manage configurations
  + Collect diagnostic data
* **Related resources** When you deploy a virtual machine, you need to think about the amount and type of storage, such as whether this VM will be connected to the internet and need a public IP or which kind of traffic is allowed to go to or from the VM. Some of these related resources, described in the following list, are mandatory for deploying a VM.
  + **Resource group** Every virtual machine needs to be contained in a resource group. You can create a new resource group or reuse an existing one.
  + **Storage account** The virtual disks needed by the VM are .vhd files stored as page blobs in a storage account. Depending on the performance requirements of your VM, you can use standard or premium Storage accounts. If you configure managed disks when deploying a VM, Azure handles the Storage account automatically and it won’t appear in the VM configuration.
  + **Virtual network** To be able to communicate with the rest of the world, your new VM needs to be connected to a virtual network.
  + **Network interface** As in the physical world, your VM needs a network interface to connect to the virtual network for sending and receiving information.

Once you have gathered all the information that you need to deploy your VM, you are ready for deployment. You have several ways of doing this task:

* Using the Azure portal
* Using PowerShell
* Using Azure CLI
* Programmatically using REST API or C#

In general, when you want to deploy a new VM, you need to follow these steps:

1. Create a resource group for the VM. You can also use an existing resource group for this VM.
2. Create a virtual network. If you are using the Azure portal, you can do this while you are creating the VM. For PowerShell and Azure CLI, you need to specify the virtual network. However, if a virtual network doesn’t already exist, one is created automatically.
3. Create a virtual NIC. If you are using Azure portal, PowerShell, or Azure CLI, you don’t need to do this because the deployment process automatically creates the NIC for you.
4. Create the virtual machine.

The following example shows how to create a simple .NET Core console application for creating a VM:

1. Open Visual Studio Code. You need to have installed the Omnishare extension.
2. Create a folder for your project.
3. In the terminal window, change the working directory to the folder that you created in the previous step and type the following command.

dotnet new console

1. Install the nuget package Microsoft.Azure.Management.Fluent.

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg004a)

dotnet add package Microsoft.Azure.Management.Fluent

1. Create an empty file called **azureauth.properties**. Add the content shown in [Listing 1-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex01) to the file. Replace the variables with the values from your Azure subscription.
2. Replace the content of Program.cs file with the content in [Listing 1-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex02). [Listing 1-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex02) shows how to create a virtual machine with managed disks in your Azure subscription.

**Listing 1-1** azureauth.properties

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-1a)

subscription=<subscription-id>

client=<client-id>

key=<client-secret>

tenant=<tenant-id>

managementURI=https://management.core.windows.net/

baseURL=https://management.azure.com/

authURL=https://login.windows.net/

graphURL=https://graph.windows.net/

**Listing 1-2** Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-2a)

//dotnet core 2.2

using System;

using Microsoft.Azure.Management.Compute.Fluent;

using Microsoft.Azure.Management.Compute.Fluent.Models;

using Microsoft.Azure.Management.Fluent;

using Microsoft.Azure.Management.ResourceManager.Fluent;

using Microsoft.Azure.Management.ResourceManager.Fluent.Core;

namespace ch1\_1\_1

{

class Program

{

static void Main(string[] args)

{

//Create the management client. This will be used for all the operations

//that we will perform in Azure.

var credentials = SdkContext.AzureCredentialsFactory

.FromFile("./azureauth.properties");

var azure = Azure.Configure()

.WithLogLevel(HttpLoggingDelegatingHandler.Level.Basic)

.Authenticate(credentials)

.WithDefaultSubscription();

//First of all, we need to create a resource group where we will add all

//the resources

// needed for the virtual machine

var groupName = "az204-ResourceGroup";

var vmName = "az204VMTesting";

var location = Region.USWest2;

var vNetName = "az204VNET";

var vNetAddress = "172.16.0.0/16";

var subnetName = "az204Subnet";

var subnetAddress = "172.16.0.0/24";

var nicName = "az204NIC";

var adminUser = "azureadminuser";

var adminPassword = "Pa$$w0rd!2019";

Console.WriteLine($"Creating resource group {groupName} ...");

var resourceGroup = azure.ResourceGroups.Define(**groupName**)

.WithRegion(**location**)

.Create();

//Every virtual machine needs to be connected to a virtual network.

Console.WriteLine($"Creating virtual network {vNetName} ...");

var network = azure.Networks.Define(**vNetName**)

.WithRegion(**location**)

.WithExistingResourceGroup(**groupName**)

.WithAddressSpace(**vNetAddress**)

.WithSubnet(**subnetName**, **subnetAddress**)

.Create();

//Any virtual machine need a network interface for connecting to the

//virtual network

Console.WriteLine($"Creating network interface {nicName} ...");

var nic = azure.NetworkInterfaces.Define(**nicName**)

.WithRegion(**location**)

.WithExistingResourceGroup(**groupName**)

.WithExistingPrimaryNetwork(**network**)

.WithSubnet(**subnetName**)

.WithPrimaryPrivateIPAddressDynamic()

.Create();

//Create the virtual machine

Console.WriteLine($"Creating virtual machine {vmName} ...");

azure.VirtualMachines.Define(**vmName**)

.WithRegion(**location**)

.WithExistingResourceGroup(**groupName**)

.WithExistingPrimaryNetworkInterface(**nic**)

.WithLatestWindowsImage("MicrosoftWindowsServer", "WindowsServer",

"2012-R2-Datacenter")

.WithAdminUsername(**adminUser**)

.WithAdminPassword(**adminPassword**)

.WithComputerName(**vmName**)

.WithSize(VirtualMachineSizeTypes.StandardDS2V2)

.Create();

{

}

}

}

}

***NOTE* APPLICATION REQUIREMENTS**

To run all the examples through this book, you need to have an Azure subscription. If you don’t have an Azure subscription, you can create a free subscription for testing the code in this book.

Also, you need to create an Azure AD application and a security principal in your Azure subscription. You need to configure these elements to grant, create, and modify privileges to your application. Follow the instructions in this procedure for creating the Azure AD application and the security principal. See [*https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-create-service-principal-portal*](https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-create-service-principal-portal)*.*

As you can see in [Listing 1-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex02), you need to create each of the related and required resources separately and then provide all the required dependencies to the Azure management client that creates the VM.

Before you proceed to deploy a new VM, you also need to take into account other considerations that would affect the deployment. For example, if your application or software solution must be highly available, you would typically use a load balancer. If your virtual machines use a load balancer, you need to put your VMs that host the application into an availability set. Using an availability set ensures that any virtual machine in the same availability set is not placed on the same hardware. Placing the virtual machines in different equipment ensures that the VMs are not restarted at the same time because of software upgrades on the servers running the VM. A virtual machine may only be added to an availability set during the creation of the VM. If you forget to add the VM to an availability set, you need to delete the VM and start from the beginning.

***NEED MORE REVIEW?* MANAGE THE AVAILABILITY OF YOUR VIRTUAL MACHINES**

You can find more information about how to manage the availability of your virtual machines by reviewing the article at [*https://docs.microsoft.com/en-us/azure/virtual-machines/windows/manage-availability*](https://docs.microsoft.com/en-us/azure/virtual-machines/windows/manage-availability).

**Images *EXAM TIP***

Creating a VM is a straightforward process, but you still need to plan the deployment. You need to consider whether you need to make the application highly available hosted in the virtual machine, or if you need to scale-up and down the number of VMs associated to a load balancer. In such cases, you need to remember to create the availability set before you create the VMs.

**Configure VMs for remote access**

The preceding section reviewed how to create a new VM programmatically. When you created the VM from the previous example, you were not able to access your new VM. The reason for this is that you didn’t configure a public IP address that you can use for remotely accessing  
the VM.

As you can imagine, just adding a public IP to your VM may lead to excessive exposure to the internet. You could control that exposure by using the firewalls provided by the operating system. But maintaining all firewalls for a number VMs can be a time-consuming task. Azure provides you with Network Security Groups as a mechanism for filtering the traffic between your Azure resources and different networks, including the internet. If you need to configure the remote access for your VM, you need to add a security rule to a network security group associated with your VM.

By default, any VM that you deploy using an Azure virtual machine image has the corresponding remote access protocol enabled—that is, Remote Desktop Protocol or RDP for Windows VMs and Secure Shell or SSH for Linux VMs.

Following from the example in [Listing 1-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex02), you need to add three additional items to the code to have remote access to your VM:

* **A public IP** You can configure a static or dynamic IP. Static IP has costs associated, so this example uses a dynamic IP.
* **A network security group** You need this for managing the security rules that allow or deny access to the VM.
* **A security rule** You need to create a security rule for allowing access to the VM using the appropriate remote protocol. For this example, you need to allow the traffic to TCP/3389 port. This is the port for the Remote Desktop Protocol.

[Listing 1-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex03) shows the modifications in bold that you need to make in your code for configuring the remote access to your VM during the deployment.

**Listing 1-3** Modified Program.cs

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-3a)

//dotnet core 2.2

using System;

using Microsoft.Azure.Management.Compute.Fluent;

using Microsoft.Azure.Management.Compute.Fluent.Models;

using Microsoft.Azure.Management.Network.Fluent;

using Microsoft.Azure.Management.Fluent;

using Microsoft.Azure.Management.ResourceManager.Fluent;

using Microsoft.Azure.Management.ResourceManager.Fluent.Core;

using Microsoft.Azure.Management.Network.Fluent.Models;

namespace ch1\_1\_2

{

class Program

{

static void Main(string[] args)

{

//Create the management client. This will be used for all the operations

//that we will perform in Azure.

var credentials = SdkContext.AzureCredentialsFactory

.FromFile("./azureauth.properties");

var azure = Azure.Configure()

.WithLogLevel(HttpLoggingDelegatingHandler.Level.Basic)

.Authenticate(credentials)

.WithDefaultSubscription();

//First of all, we need to create a resource group where we will add all

//the resources

// needed for the virtual machine

var groupName = "az204-ResourceGroup";

var vmName = "az204VMTesting";

var location = Region.USWest2;

var vNetName = "az204VNET";

var vNetAddress = "172.16.0.0/16";

var subnetName = "az204Subnet";

var subnetAddress = "172.16.0.0/24";

var nicName = "az204NIC";

var adminUser = "azureadminuser";

var adminPassword = "Pa$$w0rd!2019";

**var publicIPName = "az204publicIP";**

**var nsgName = "az204VNET-NSG";**

Console.WriteLine($"Creating resource group {groupName} ...");

var resourceGroup = azure.ResourceGroups.Define(groupName)

.WithRegion(location)

.Create();

//Every virtual machine needs to be connected to a virtual network.

Console.WriteLine($"Creating virtual network {vNetName} ...");

var network = azure.Networks.Define(vNetName)

.WithRegion(location)

.WithExistingResourceGroup(groupName)

.WithAddressSpace(vNetAddress)

.WithSubnet(subnetName, subnetAddress)

.Create();

**//You need a public IP to be able to connect to the VM from the Internet**

**Console.WriteLine($"Creating public IP {publicIPName} ...");**

**var publicIP = azure.PublicIPAddresses.Define(publicIPName)**

**.WithRegion(location)**

**.WithExistingResourceGroup(groupName)**

**.Create();**

**//You need a network security group for controlling the access to the VM**

**Console.WriteLine($"Creating Network Security Group {nsgName} ...");**

**var nsg = azure.NetworkSecurityGroups.Define(nsgName)**

**.WithRegion(location)**

**.WithExistingResourceGroup(groupName)**

**.Create();**

**//You need a security rule for allowing the access to the VM from the**

**//Internet**

**Console.WriteLine($"Creating a Security Rule for allowing the remote**

**access");**

**nsg.Update()**

**.DefineRule("Allow-RDP")**

**.AllowInbound()**

**.FromAnyAddress()**

**.FromAnyPort()**

**.ToAnyAddress()**

**.ToPort(3389)**

**.WithProtocol(SecurityRuleProtocol.Tcp)**

**.WithPriority(100)**

**.WithDescription("Allow-RDP")**

**.Attach()**

**.Apply();**

//Any virtual machine needs a network interface for connecting to the

//virtual network

Console.WriteLine($"Creating network interface {nicName} ...");

var nic = azure.NetworkInterfaces.Define(nicName)

.WithRegion(location)

.WithExistingResourceGroup(groupName)

.WithExistingPrimaryNetwork(network)

.WithSubnet(subnetName)

.WithPrimaryPrivateIPAddressDynamic()

**.WithExistingPrimaryPublicIPAddress(publicIP)**

**.WithExistingNetworkSecurityGroup(nsg)**

.Create();

//Create the virtual machine

Console.WriteLine($"Creating virtual machine {vmName} ...");

azure.VirtualMachines.Define(vmName)

.WithRegion(location)

.WithExistingResourceGroup(groupName)

.WithExistingPrimaryNetworkInterface(nic)

.WithLatestWindowsImage("MicrosoftWindowsServer", "WindowsServer",

"2012-R2-Datacenter")

.WithAdminUsername(adminUser)

.WithAdminPassword(adminPassword)

.WithComputerName(vmName)

.WithSize(VirtualMachineSizeTypes.StandardDS2V2)

.Create();

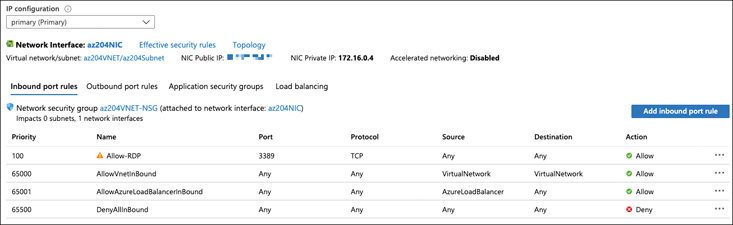
}

}

}

The code that you used for creating and enabling the remote access to the VM is a good way to understand the relationship between the different components needed for deploying a VM. You need to understand these relationships if you need to deploy or reconfigure a VM using PowerShell or Azure CLI. After you have created the VM using the modified code in  
[Listing 1-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex03), use the following procedure for verifying that everything is working correctly:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the Search Resources text box on the middle-top side of the portal, type **az204VMTesting**.
3. In the result list, click the name of the virtual machine.
4. On the az204VMTesting virtual machine page, click Networking in the Settings section.
5. In the Network Security Group list of security rules, shown in [Figure 1-1](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig01), ensure that there is a rule named Allow-RDP.



**Figure 1-1** Virtual machine network security group

1. On the az204VMTesting page, click Connect in the Settings section in the navigation list on the left side of the page.
2. On the Connect page, ensure that the RDP tab is selected.
3. Ensure that the Public IP address option is selected in the IP Address drop-down menu.
4. Click the Download RDP File button.
5. Once the RDP file has been downloaded, double-click the RDP file for opening the remote session with your VM. You need to provide the password configured in the code in [Listing 1-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex03).

When configuring remote access in a VM, you need to consider whether that VM needs to be accessible from the internet or if you can restrict the access. As a general rule of thumb, you should avoid configuring remote access from the internet for production VMs. For these VMs, you should consider deploying a virtual private network and remote access the VM using the private IP instead of a public IP. This way, you can use the public IP to publish only the service hosted in the VM, like an IIS service, while getting remote access to the VM using the private IP.

***NEED MORE REVIEW?* NETWORK SECURITY GROUPS**

You can find more information about how to manage network security groups using the Azure portal, Azure PowerShell, or Azure CLI by reviewing the following articles:

* [*https://docs.microsoft.com/en-us/azure/virtual-network/security-overview*](https://docs.microsoft.com/en-us/azure/virtual-network/security-overview)
* [*https://docs.microsoft.com/en-us/azure/virtual-network/tutorial-restrict-network-access-to-resources*](https://docs.microsoft.com/en-us/azure/virtual-network/tutorial-restrict-network-access-to-resources)

**Images *EXAM TIP***

You need to carefully consider when to configure a VM for remote access from the internet. In general, you should not configure remote access over public IPs on production VMs. For those cases, you should deploy a virtual private network and connect to your VM using its private IP.

**Create ARM templates**

One of the most significant advantages of using Azure IaaS is the level of automation that you can achieve when deploying new services, resources, or infrastructure. One of the main reasons you can do this is because Microsoft provides you the Azure Resource Manager (ARM), which is the deployment and management service in Azure. The ARM service is in charge of creating, updating, and deleting the different services you can deploy in your subscription. You can interact with all actions offered by the ARM service using the same API. Because of this same API, no matter which mechanism you use—portal, PowerShell, Azure CLI, Rest API, or client SDKs— you get a consistent behavior and result when interacting with ARM.

When you work with the Azure Resource Manager, there are some concepts and terms that you need to understand clearly:

* **Resource** These are the items you can manage in Azure.
* **Resource group** This is a container that you use for holding resources. You can use any grouping criteria for your resources, but you need to remember that any single resource needs to be contained in a resource group. You can also use resource groups for managing different levels of management access to different groups of users.
* **Resource provider** A resource provider is a service that offers different kinds of Azure resources, and they manage the resource’s lifecycle. For example, the service in charge of providing virtual machine resources is Microsoft.Compute provider. You can also use Microsoft.Storage provider for storage accounts or Microsoft.Network for all networking resources.
* **Resource Manager template** This is the file that you need to provide to the ARM API when you want to deploy one or more resources to a resource group or subscription. This file is written in JavaScript Object Notation (JSON).

The main advantage of using ARM templates is that you have the definition of all the resources that you want to deploy in a consistent structure. This allows you to reuse the same template for deploying the same group of resources in different subscriptions, resource groups, or regions. Some common scenarios in which you can take advantage of the ARM templates are disaster recovery plan implementations, high-availability configurations, or automatic provisioning scenarios (such as continuous deployment scenarios). In the following code snippet, you can see the most basic structure for an ARM template.

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg013a)

{

"$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.

json#",

"contentVersion": "",

"parameters": { },

"variables": { },

"functions": [ ],

"resources": [ ],

"outputs": { }

}

Insofar as the ARM template structure is concerned, only the $schema, contentVersion, and resources sections are required to be present in a valid template. Following is a brief description of each section in a template:

* **$schema** This required section sets the JSON schema that describes the version of the template you will use in the file. You can choose between two different schemas depending on the deployment type:
  + **Resource group deployments** You should use [*https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#*](https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json).
  + **Subscription deployments** You should use [*https://schema.management.azure.com/schemas/2018-05-01/subscriptionDeploymentTemplate.json#*](https://schema.management.azure.com/schemas/2018-05-01/subscriptionDeploymentTemplate.json).
* **contentVersion** In this required section, you set a value you can use for providing your internal version number to the template, such as 1.0.0. This version number is only meaningful for you; Azure does not use it. Typically, you change the version number when you make significant changes to the template.
* **parameters** This is an optional section that you can use to set the values provided to the Resource Manager when you perform a deployment. You can use customizable template parameters for different deployments without changing the content of the template.
* **variables** This optional section contains the values that you will reuse across the entire template. You use variables for improving the usability and readability of the template.
* **functions** You can use this optional section for defining your functions to be used in the template.
* **resources** This is a required section that contains all the resources that will be deployed or updated by the template.
* **outputs** This optional section defines the values that the Resource Manager should return once the deployment has finished.

[Listing 1-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex04) shows the ARM template that you need to use for deploying new VMs with the same configuration. You may modify the values of the parameters according to your needs.

**Listing 1-4** ARM template for deploying a VM

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-4a)

{

"$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.

json#",

"contentVersion": "1.0.0.0",

"parameters": {

"virtualNetworks\_az204VNET\_name": {

"defaultValue": "az204demoVNET",

"type": "string"

},

"networkInterfaces\_az204NIC\_name": {

"defaultValue": "az204demoNIC",

"type": "string"

},

"virtualMachines\_az204VMTesting\_name": {

"defaultValue": "az204demoVM",

"type": "string"

},

"subnets\_az204Subnet\_name": {

"defaultValue": "az204demoSubnet",

"type": "string"

},

"virtualMachines\_az204VMTesting\_id": {

"defaultValue": "[concat(parameters('virtualMachines\_az204VMTesting\_name'),

'\_OSDisk1\_1')]",

"type": "string"

},

"virtualMachines\_adminUser": {

"defaultValue": "azureadminuser",

"type": "string"

},

"virtualMachines\_adminpassword": {

"defaultValue": "Pa$$w0rd",

"type": "securestring"

}

},

"variables": {

"osDiskName": "\_OSDisk1\_1\_39c654d89d88405e968db84b722002d1"

},

"resources": [

{

"type": "Microsoft.Compute/virtualMachines",

"name": "[parameters('virtualMachines\_az204VMTesting\_name')]",

"apiVersion": "2018-06-01",

"location": "westus2",

"tags": {},

"scale": null,

"properties": {

"hardwareProfile": {

"vmSize": "Standard\_DS2\_v2"

},

"storageProfile": {

"imageReference": {

"publisher": "MicrosoftWindowsServer",

"offer": "WindowsServer",

"sku": "2012-R2-Datacenter",

"version": "latest"

},

"osDisk": {

"osType": "Windows",

"name": "[concat(parameters('virtualMachines\_az204VMTesting\_

name'), variables('osDiskName'))]",

"createOption": "FromImage",

"caching": "ReadWrite"

},

"dataDisks": []

},

"osProfile": {

"computerName": "[parameters(

'virtualMachines\_az204VMTesting\_name')]",

"adminUsername": "azureadminuser",

"adminPassword": "Pa$$w0rd",

"windowsConfiguration": {

"provisionVMAgent": true,

"enableAutomaticUpdates": true

},

"secrets": [],

"allowExtensionOperations": true

},

"networkProfile": {

"networkInterfaces": [

{

"id": "[resourceId('Microsoft.Network/networkInterfaces', parameters

('networkInterfaces\_az204NIC\_name'))]",

"properties": {

"primary": true

}

}

]

}

},

"dependsOn": [

"[resourceId('Microsoft.Network/networkInterfaces', parameters(

'networkInterfaces\_az204NIC\_name'))]"

]

},

{

"type": "Microsoft.Network/networkInterfaces",

"name": "[parameters('networkInterfaces\_az204NIC\_name')]",

"apiVersion": "2018-10-01",

"location": "westus2",

"tags": {},

"scale": null,

"properties": {

"ipConfigurations": [

{

"name": "primary",

"properties": {

"privateIPAllocationMethod": "Dynamic",

"subnet": {

"id": "[resourceId('Microsoft.Network/virtualNetworks/subnets',

parameters('virtualNetworks\_az204VNET\_name'),

parameters('subnets\_az204Subnet\_name'))]"

},

"primary": true,

"privateIPAddressVersion": "IPv4"

}

}

],

"dnsSettings": {

"dnsServers": [],

"appliedDnsServers": []

},

"enableAcceleratedNetworking": false,

"enableIPForwarding": false,

"primary": true,

"tapConfigurations": []

},

"dependsOn": [

"[resourceId('Microsoft.Network/virtualNetworks/subnets', parameters('virtualNetworks\_

az204VNET\_name'), parameters('subnets\_az204Subnet\_name'))]"

]

},

{

"type": "Microsoft.Network/virtualNetworks",

"name": "[parameters('virtualNetworks\_az204VNET\_name')]",

"apiVersion": "2018-10-01",

"location": "westus2",

"tags": {},

"scale": null,

"properties": {

"resourceGuid": "145e7bfc-8b00-48cf-8fa1-082448a30bae",

"addressSpace": {

"addressPrefixes": [

"172.16.0.0/16"

]

},

"dhcpOptions": {

"dnsServers": []

},

"subnets": [

{

"name": "[parameters('subnets\_az204Subnet\_name')]",

"properties": {

"addressPrefix": "172.16.0.0/24"

}

}

],

"virtualNetworkPeerings": [],

"enableDdosProtection": false,

"enableVmProtection": false

},

"dependsOn": []

},

{

"type": "Microsoft.Network/virtualNetworks/subnets",

"name": "[concat(parameters('virtualNetworks\_az204VNET\_name'), '/',

parameters('subnets\_az204Subnet\_name'))]",

"apiVersion": "2018-10-01",

"scale": null,

"properties": {

"addressPrefix": "172.16.0.0/24"

},

"dependsOn": [

"[resourceId('Microsoft.Network/virtualNetworks',

parameters('virtualNetworks\_az204VNET\_name'))]"

]

}

]

}

This example has some interesting features. You have defined the parameters and variables that you will use throughout the template. If you look at any parameter definition, you can see that it has three elements—paramenterName, defaultValue, and type. The type element is almost self-explanatory; it sets the kind of value that this parameter will contain. The allowed types are string, securestring, int, bool, object, secureObject, and array. The parameterNam*e* is also quite straightforward and is any valid JavaScript identifier that represents the name of the parameter. However, why use a defaultValue element instead of a value element? You use defaultValue because when you define a parameter in the template file, the only required components are parameterName and type. The parameter’s value is provided during the deployment process. If you don’t provide a value for a parameter that you defined in the template, then the defaultValue will be used instead. You should bear in mind that this element is optional.

You can provide values to the parameters that you define for your template by using the command line or creating a file with the values that you want to provide to each parameter. The following example shows the content of a parameter file for the template shown previously in [Listing 1-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex04):

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg018a)

{

"$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.

json#",

"contentVersion": "1.0.0.0",

"parameters": {

"virtualNetworks\_az204VNET\_name": {

"value": "az204demoVNET"

},

"networkInterfaces\_az204NIC\_name": {

"value": "az204demoNIC"

},

"virtualMachines\_az204VMTesting\_name": {

"value": "az204demoVM"

},

"subnets\_az204Subnet\_name": {

"value": "az204demoSubnet"

},

"virtualMachines\_az204VMTesting\_id": {

"value": "[concat(parameters(

'virtualMachines\_az204VMTesting\_name'),

'\_OSDisk1\_1\_39c654d89d88405e968db84b722002d1')]"

},

"virtualMachines\_adminUser": {

"value": "azureadminuser"

},

"virtualMachines\_adminpassword": {

"value": "Pa$$w0rd"

}

}

}

When you are defining the value for a parameter, you can also use functions to construct dynamic values. If you take a look at the virtualMachines\_az204VMTesting\_id parameter, you can see that its value is set to a function. In this case, the function returns a string that is the result of adding the string \_OSDisk1\_1\_39c654d89d88405e968db84b722002d1 to the value of the parameter virtualMachines\_az204VMTesting\_name.

There are many predefined functions that you can use in your template. You can even define your custom functions for those complicated pieces of code that repeat in your template. When working with custom functions, beware of some limitations:

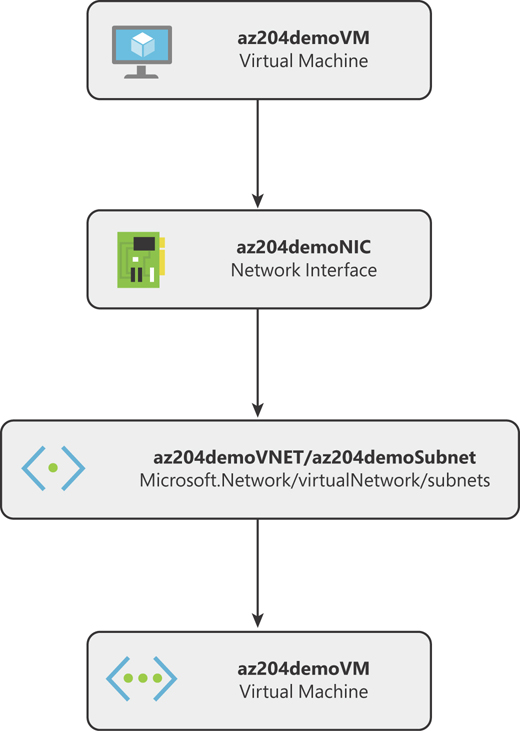
* Custom functions cannot access template variables, although you can pass them as a parameter of your function.
* Your custom function cannot access the template’s parameters; instead, they have access only to the parameters that you define in your function.
* Parameters on your custom function cannot have default values.
* Your custom function cannot call other custom functions; you only can call predefined functions.
* You cannot use the reference() predefined function.

***NOTE* TEMPLATE REFERENCE**

When you are working with ARM templates, it’s useful to consult the template reference for each type of resource you are configuring. You can review the complete template reference at [*https://docs.microsoft.com/en-us/azure/templates/*](https://docs.microsoft.com/en-us/azure/templates/). You can also review the complete reference of predefined functions at [*https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-template-functions*](https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-template-functions).

When I initially talked about the resources that you need for deploying a VM, I explained that there are some resources that you need for the VM to run correctly. For example, you need at least one virtual disk for storing the operating system. You also need a virtual network for connecting the VM with the world, and you need a virtual network interface card for connecting the VM to the virtual network. All those dependencies are defined in an ARM template by using the element dependsOn on each resource type. This element accepts a list of resource names, separated by commas, that define the resources that need to be deployed before the resource can be deployed. As a best practice to avoid ambiguity, you should reference any resource that you put on the dependsOn element by using its provider namespace and type. You can do this by using the resourceId() predefined function.

If you review the example, the virtual network virtualNetworks\_az204VNET\_name needs to be deployed before subnets\_az204Subnet\_name can be deployed (see [Figure 1-2](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig02)). The dependsOn element is required because the resources defined in the template are not deployed in the same order that appears in the template.



**Figure 1-2** Resource dependencies

***NOTE* CHILD ELEMENTS VERSUS DEPENDENCIES**

Some resources can contain other resources as child elements. This parent-child relationship is not the same as a dependency relationship. The parent-child relationship does not guarantee that the parent will be deployed before its children. You need to use the dependsOn element to ensure the correct deployment order**.**

Once you are happy with your template, you can deploy it to your Azure subscription by using PowerShell, Azure CLI, Azure Cloud Shell, or REST API. Another exciting feature is that you can store your template JSON files in a remote location. This remote location needs to be publicly available. If your template contains information that shouldn’t be public, you can provide that information as an inline parameter during the deployment. If you prefer your template not to be open to the world, you can also store your template in a Storage account and protect it by using a SAS token.

The following command shows how to deploy the example template using the template file az204-template.json and the properties file az204-parameters.json.

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg021a)

#!/bin/bash

#Azure CLI template deployment

az group create --name AZ204-ResourceGroup --location "West US"

az group deployment create \

--name AZ204DemoDeployment \

--resource-group AZ204-ResourceGroup \

--template-file az204-template.json \

--parameters @az204-parameters.json

The previous command creates the resource group called AZ204-ResoureGroup in the West US region. Then it creates a new deployment called AZ204DemoDeployment that will generate the resources defined in the az204-template.json template using the values provided in the parameters file named az204-parameters.json. Note the use of the @ symbol in front of the parameters file. This @ symbol is required by the az group deployment create command.

**Images *EXAM TIP***

ARM templates are powerful tools that enable you to create custom functions for automating some repeating actions. When you create your custom function, remember the limitations when calling predefined or other custom functions. You should also bear in mind the visibility of the template variables and parameters when working with custom functions.

**Create container images for solutions by using Docker**

With the evolution of technology and the emergence of the cloud, you need to meet other challenges presented by this technical evolution. One of these requirements is the ability to deploy pieces of software in a reliable and quick manner. Virtualization technologies were one of the keys for making this kind of reliable and quick deployment possible.

However, in the context of operating system virtualization using virtual machines, one of the main drawbacks is the fact that you have a complete set of binaries, libraries, and resources that are duplicated between virtual machines. This is where containerization provides a different approach to deploying pieces of software across multiple servers reliably and quickly.

A container is a piece of software that packages your code and all its dependencies in a single package that can be run directly by the computer environment. When a container is executed, it uses a read-only copy of the shared libraries of the operating system that your code needs to run. This reduces the required amount of resources that a container needs to run your code when compared to running the same code on a virtual machine. Container technology was initially born on Linux environments, but it also has been ported to the Microsoft Windows environment. There are several implementations of container technology in the Linux ecosystem, but Docker Containers are the most widely used.

When you move the container technology to an enterprise environment, scaling dynamically and automatically is a problem, just as it is with virtual machines. There are several available solutions in the market, such as Docker Swarm, DC/OS, or Kubernetes. All these solutions are orchestration solutions that automatically scale and deploy your containers in the available resources.

Azure provides several services that allow you to deploy your application in a container. It doesn’t matter if you decide to use Azure Kubernetes Services, Service Fabric, Azure Web Apps for Containers, Azure Container Registry, or Azure Container Instances; all these services use the same container technology implementation, Docker.

Before you can deploy your application to any of these services, you need to put your application into a container by creating an image of your container. A container image is a package that contains everything you need—code, libraries, environment variables, and configuration files—to run your application. Once you have your container images, you can create instances of the image for running the code, each of which is a container. If you need to make modifications to one of your containers, you need to modify the image definition and redeploy the container. In general, any change that you make to a container is not persisted across reboots. If you need to ensure that some information in your container is not deleted when a container reboots, you need to use external mount points, known as volumes.

When you create your container image, you must define your application’s requirements, which are placed in a file called Dockerfile. This Dockerfile contains the definition and requirements needed for creating your container image. Use the following high-level procedure for creating an image:

1. **Create a directory for the new image.** This directory contains your Docker file, your code, and any other dependency that you need to include in the image, and that is not available in a separate image.
2. **Create the Dockerfile.** This file contains the definition of your image. [Listing 1-5](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex05) shows an example of a functional Dockerfile.
3. **Open a command line.** You use this command line to run the Docker commands.
4. **Create the container image.** Use the command docker build to create the image. When you create an image, you should add a tag to identify the image and the version. If you 23don’t set a version number, docker automatically assigns the default value *latest*. You need to provide the path of the folder that contains the Dockerfile. This command has the following structure:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg023a)

docker build --tag=<tag\_name>[:<version>] <dockerfile\_dir>

1. **List the newly created image.** Once Docker finishes downloading all the dependencies for your image, you can ensure that your image has been created by executing this command:

docker image ls

**Listing 1-5** Dockerfile example

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-5a)

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define an environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

***NEED MORE REVIEW?* BEST PRACTICES FOR WRITING DOCKERFILES**

When you are writing your Dockerfile, you should bear in mind some best practices detailed at [*https://docs.docker.com/develop/develop-images/dockerfile\_best-practices/*](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/).

For complex applications, creating an image for each component of the application can become a complicated task. For scenarios in which you need to define and run multiple containers, you can use Docker Compose. You can also think of Docker Compose as the definition of your images for a production environment. If your application is comprised of several images, you can define the relationship between those images and how they are exposed to the external world. 24Using Docker Compose you can also set the limits of the resources assigned to each container when it executes and define what happens if one container associated with a service fails.

A service in the Docker world is each of the pieces that are part of your application. A service has a one-to-one relationship with an image. It’s important to remember that a service can have multiple instances of the same image, which means you can have various containers. The docker-composer.yaml file contains the definitions of the relationships and requirements needed for running your application.

***NEED MORE REVIEW?* FULLY FUNCTIONAL EXAMPLE**

You can run a fully functional example in your local environment by reviewing the instructions published by Microsoft *at*[*https://docs.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app*](https://docs.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app).

**Images *EXAM TIP***

The modifications that you make to a container while it’s running do not persist if you reboot the container. If you need to make changes to the content of a container, you need to modify the image container and then redeploy the container. If you need your container to save information that needs to be persisted across reboots, you need to use volumes.

**Publish an image to the Azure Container Registry**

The main purpose of creating an image is to make your code highly portable and independent from the server that executes your code. To achieve this objective, your image needs to be accessible by all the servers that can execute your image. Therefore, you need to store your image in a centralized storage service.

Azure Container Registry (ACR) is Microsoft’s implementation of a Docker registry service, based on the Docker Registry 2.0 definition. Using this managed Docker registry service, you can privately store your images for later distribution to container services, such as Azure Managed Kubernetes Service. You can also use ACR for building your images on the fly and automating the building of the image based on the commits of your source code.

Before you can upload an image to your private container registry, you need to tag the image. To do this, you need to include the name of your private container registry in the tag. You will use the name structure <acr\_name>.azurecr.io/[repository\_name][:version]. The following list breaks down each part of the tag:

* **acr\_name** This is the name that you gave to your registry.
* **repository\_name** This is an optional name for a repository in your registry. ACR allows you to create multilevel repositories inside the registry. If you want to use a custom repository, just put its name in the tag.
* **version** This is the version that you use for the image.

Use the following procedure for pushing your image to your ACR registry. These steps assume that you already created an Azure Container Registry and installed the latest Azure CLI:

1. Log in to your Azure subscription.

az login

1. Log in to your registry using this command:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg025-1a)

az acr login –--name <acr\_name>

1. Tag the image that you want to upload to the registry using this command:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg025-2a)

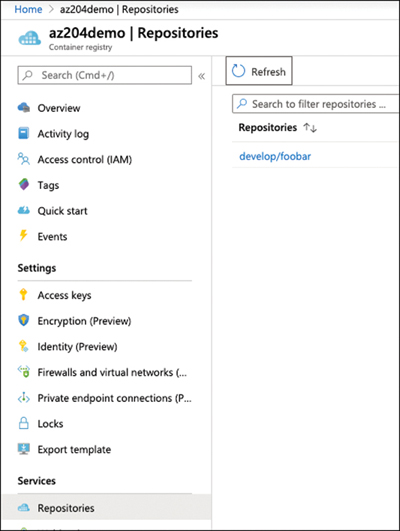
docker tag foobar <acr\_name>.azurecr.io/<repository\_name>/<image\_name>

1. Push the image to the registry using this command:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg025-3a)

docker push <acr\_name>.azurecr.io/<repository\_name>/<image\_name>

When Docker finishes pushing your image to the registry, you can browse the repositories in your registry, as shown in [Figure 1-3](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig03), to verify that it has been successfully uploaded.



**Figure 1-3** Browse container repository

The next section reviews how to run the container from the image you have already pushed to the registry.

**Images *EXAM TIP***

A container registry is useful not only for storing your container images but also for automating the deployment of containers into the Azure Container services. Any continuous delivery service, like Azure Pipelines, would need a container registry for deploying the container images.

**Run containers by using Azure Container Instance**

Once you have created your image and made it available to Azure services by pushing it to your container registry, it is time to run the container in any of the services that Azure offers to you. Follow this high-level procedure:

1. Create as many images as your application needs to run correctly.
2. Upload or push your application images to a container registry.
3. Deploy the application.

When you want to create an image in the Azure Container Instance (ACI) service from your Azure Container Registry (ACR), you need to authenticate before you can pull the image from your ACR. For the purpose of demonstration, the following procedure uses Admin account authentication to show how to create and run a container in ACI:

1. Sign in to the Azure cloud shell ([*https://shell.azure.com*](https://shell.azure.com/)).
2. In the Shell Selector, select Bash.
3. Open the online editor by clicking the curly brace icon to the right of the Shell Selector.
4. Use the script in [Listing 1-6](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex06) to create a service principal password and to create a container from your images in the registry.

**Listing 1-6** Creating a service principal password

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-6a)

#!/bin/bash

#Some variable definition useful for the script

ACR\_NAME=az204demo

SP\_NAME=az204demo\_sp

IMAGE\_TAG=az204demo.azurecr.io/develop/foobar:latest

RESOURCE\_GROUP=AKSdemo-RG

APP\_NAME=foobar

APP\_DNS\_NAME=prueba

#Get the registry ID. You will need this ID for creating the authorization to the

#service principal

ACR\_ID=$(az acr show --name $ACR\_NAME --query id --output tsv)

#Get the ACR login server

ACR\_SERVER=$(az acr show --name $ACR\_NAME --query loginServer --output tsv)

#Get the service principal password. We will grant pull only privileges to the service

#principal

echo "Generating Service Principal password"

SP\_PASS=$(az ad sp create-for-rbac --name http://$SP\_NAME --scopes $ACR\_ID

--role acrpull --query password --output tsv)

#Get the App ID associated to the service principal

SP\_ID=$(az ad sp show --id http://$SP\_NAME --query appId --output tsv)

echo "Service principal ID: $SP\_ID"

echo "Service principal password: $SP\_PASS"

#Create the container in the Container Instance service

az container create --resource-group $RESOURCE\_GROUP --name $APP\_NAME --image

$IMAGE\_TAG --cpu 1 --memory 1 --registry-login-server $ACR\_SERVER --registry-username

$SP\_ID --registry-password $SP\_PASS --dns-name-label $APP\_DNS\_NAME --ports 80

1. In the top-right corner of the online editor, below the user information, click the ellipsis icon, and then click Save. Provide a name for the script.
2. In the Azure Cloud Shell, execute the script by typing the following command in the bash shell:

sh <your\_script\_name>

Once you have executed this procedure, you can access your container by looking for your container’s name in the Azure portal. You can also access the application that you put in this container by entering the URL of the container into a browser. The URL for this container will be in the form of <APP\_DNS\_NAME>.<region>.azurecontainer.io, based on the value of the variable APP\_DNS\_NAME that you provided in the previous script.

**Images *EXAM TIP***

You can use several authentication mechanisms, such as an individual login with Azure AD, an admin account, or a service principal. Authentication with Azure AD is a good approach for your development and testing environment. Using the admin account is disabled by default and is discouraged for production environments because you need to put the admin account password in your code. For production environments, the recommended way to pull images is using service principals for authentication with the ACR.

**Skill 1.2: Create Azure App Service web apps**

Azure App Service is a Platform as a Service (PaaS) solution that Microsoft offers to assist with developing your applications, mobile app back end, or REST APIs without worrying about the underlying infrastructure.

You use most of the more popular programming languages—.NET, .NET Core, Java, Ruby, Node.js, PHP, or Python—on top of your preferred platform (Linux or Windows). Azure App Service provides you with enterprise-level infrastructure capabilities, such as load balancing, security, autoscaling, and automated management. You can also include Azure App Service in your continuous deployment lifecycle thanks to the integration with GitHub, Docker Hub, and Azure DevOps.

**This skill covers how to**

* [Create an Azure App Service web app](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec7)
* [Enable diagnostics logging](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec8)
* [Deploy code to a web app](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec9)
* [Configure web app settings including SSL, API, and connection strings](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec10)
* [Implement autoscaling rules, including scheduled autoscaling and scaling by operational or system metrics](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec11)

**Create an Azure App Service web app**

Azure App Service is a PaaS service based on HTTP that allows you to deploy your web or mobile back-end applications or REST APIs to the cloud. Using Azure App Services enables you to develop your application in any of the most popular languages of the moment, like .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Azure App Services also offers you the flexibility of working with any of your favorite platforms: Windows, Linux, or Linux-based containers. The advantage of using Azure App Services is not limited only to the different options for developing. It also integrates quite well with different continuous integration and deployment platforms.

When you plan to create an Azure App Service, there are some concepts about how your application performs that you need to understand. Every App Service needs resources to execute your code. Virtual machines are the base of these resources. Although Azure automatically provides the low-level configuration for running these virtual machines, you still need to provide some high-level information. An App Service plan manages the group of virtual machines that host your web application.

You can think of an App Service plan like a server farm that runs in a cloud environment. This also means that you are not limited to running a single App Service in an App Service plan. You can share the same computing resources between several App Services that you deploy on the same App Service plan.

When you create a new App Service plan, you need to provide the following information:

* **Region** This is the region where you deploy the App Service plan. Any App Service in this App Service plan is placed in the same region as the App Service plan.
* **Number of instances** This is the number of VMs added to your App Service plan. Bear in mind that the maximum number of instances that you can configure for your App Service plan depends on the pricing tier that you select. You can scale the number of instances manually or automatically.
* **Size of the instances** You configure the size of the VM used in the App Service plan.
* **Operating system platform** This controls whether your web application runs on Linux or Windows VMs. Depending on the operating system, you have access to different pricing tiers. Beware that once you have selected the operating system platform, you cannot change the OS for the App Service without re-creating the App Service.
* **Pricing tier** This sets the features and capabilities available for your App Service plan and how much you pay for the plan. For Windows VMs, two basic pricing tiers use shared VMs—F1 and D1. This shared tier is not available for Linux VMs. When you use the basic pricing tiers, your code runs alongside other Azure customers’ code.

When you run an App Service in an App Service plan, all instances configured in the plan execute the code corresponding to your app. This means that if you have five virtual machines, any app you deploy into the App Service runs on each of the five VMs. Other operations related to the App Service, such as additional deployment slots, diagnostic logs, backups, or WebJobs, also are executed using the resources of each virtual machine in the App Service plan.

Azure App Service also provides you with the ability to integrate the authentication and authorization of your web application, REST API, a mobile app back end, or even Azure Functions. You can use different well-known authentication providers, like Azure, Microsoft, Google, Facebook, and Twitter, for authenticating users in your application. You can also use other authentication and authorization mechanisms on your apps. However, by using this security module, you can provide a reasonable level of security to your application with minimal or even no required code changes.

There are situations when your application may require access to resources on your on-premises infrastructure, and App Service provides you with two different approaches:

* **VNet integration** This option is available only for Standard, Premium, or PremiumV2 pricing tiers. This integration allows your web app to access resources in your virtual network. If you create a site-to-site VPN with your on-premises infrastructure, you can access your private resources from your web app.
* **Hybrid connections** This option depends on the Azure Service Bus Relay and creates a network connection between the App Service and an application endpoint. This means that hybrid connections enable the traffic between specific TCP host and port combinations.

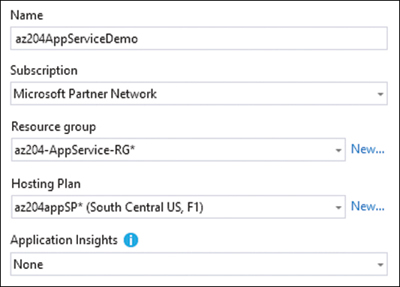
The following procedure shows how to create an App Service plan and upload a simple web application based on .NET Core using Visual Studio 2019. Ensure that you have installed the ASP.NET and web development workload, and you have installed the latest updates.

1. Open Visual Studio 2019 on your computer.
2. In the Visual Studio 2019 Home window, in the column named Get Started, click the Continue Without Code link at the bottom of the column.
3. Click the Tools menu and choose Get Tools And Features. Verify that the ASP.NET And Web Development In The Web & Cloud section is checked.
4. In the Visual Studio 2019 window, click File > New > Project to open the New Project window.
5. In the Create a New Project window, select C# in the drop-down menu below the Search For Templates text box at the top right of the window.
6. In the All Project Types drop-down menu, select Web.
7. In the list of templates on the right side of the window, select ASP.NET Core Web Application.
8. In the Configure Your New Project window, complete the following steps:
   * Select a name for the project.
   * Enter a path for the location of the solution.
   * In the Solution drop-down menu, select Create A New Solution.
   * Enter a name for the solution.
9. Click the Create button in the bottom-right corner of the Configure Your New Project window. This opens the Create A New ASP.NET Core Web Application window.
10. In the Create A New ASP.NET Core Web Application window, ensure that the following values are selected in the two drop-down menus at the top of the window:
    * .NET Core
    * ASP.NET Core 3.1
11. Select Web Application from the Project Templates area in the center of the window.
12. Uncheck the option Configure For HTTPS on the bottom-right side of the window.
13. Click the Create button in the bottom-right corner of the Create A New ASP.NET Core Web Application window.

At this point, you have created a simple ASP.NET Core web application. You can run this application in your local environment to ensure that the application is running correctly before you publish the application to Azure.

Now you need to create the Resource Group and App Service plan that hosts the App Service in Azure:

1. In your Visual Studio 2019 window, ensure that you have opened the solution of the web application that you want to publish to Azure.
2. On the right side of the Visual Studio window, in the Solution Explorer window, right-click the project’s name.
3. In the contextual menu, click Publish. This opens the Pick A Publish Target window.
4. In the Pick A Publish Target window, make sure that App Service is selected from the list of Available Targets on the left side of the window.
5. In the Azure App Service section, in the right side of the window, ensure that Create New Option is selected.
6. In the bottom-right corner of the window, click the Create Profile button, which opens the Create App Service window.
7. In the Create App Service window, add a new Azure account. This account needs to have enough privileges in the subscription for creating new resource groups, app services, and an App Service plan.
8. Once you have added a valid account, you can configure the settings for publishing your web application, as shown in [Figure 1-4](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig04).



**Figure 1-4** Creating an app service

1. In the App Name text box, enter a name for the App Service. By default, this name matches the name that you gave to your project.
2. In the Subscription drop-down menu, select the subscription in which you want to create the App Service.
3. In the Resource Group drop-down menu, select the resource group in which you want to create the App Service and the App Service plan. If you need to create a new resource group, you can do so by clicking the New link on the right side of the drop-down menu.
4. To the right of the Hosting Plan drop-down menu, click the New link to open the Configure Hosting Plan window.
5. In the Configure Hosting Plan window, type a name for the App Service plan in the App Service Plan text box.
6. Select a region from the Location drop-down menu.
7. Select a virtual machine size from the Size drop-down menu.
8. Click the OK button in the bottom-right corner of the window. This closes the Configure Hosting Plan window.
9. At the bottom-right corner of the Create App Service window, click the Create button. This starts the creation of the needed resources and the upload of the code to the App Service.
10. Once the publishing process has finished, Visual Studio opens your default web browser with the URL of the newly deployed App Service. This URL will have the structure https://<*your\_app\_service\_name*>.azurewebsites.net.

Depending on the pricing tier that you selected, some features are enabled, such as configuring custom domains or configuring SSL connections for your web applications. For production deployment, you should use Standard or Premium pricing tiers. As your feature needs change, you can choose different pricing tiers. You can start by using the free tier, F1, in the early stages of your deployment and then increase to an S1 or P1 tier if you need to make backups of your web application or need to use deployment slots.

Even if the premium pricing tiers do not fit your computer requirements, you can still deploy a dedicated and isolated environment, called Isolated pricing tier. This tier provides you with dedicated VMs running on top of dedicated virtual networks where you can achieve the maximum level of scale-out capabilities. Bear in mind that you cannot use the shared tier D1 to deploy a Linux App Service plan.

**Images *EXAM TIP***

Because Azure App Service does not support the same features for Linux and Windows, you cannot mix Windows and Linux apps in the same resource group in the same region. For more information about the limitations of Linux App Services, review the following article: [*https://docs.microsoft.com/en-us/azure/app-service/containers/app-service-linux-intro*](https://docs.microsoft.com/en-us/azure/app-service/containers/app-service-linux-intro).

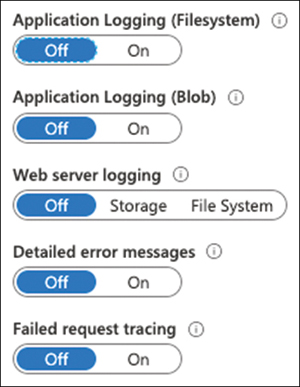
**Enable diagnostics logging**

Troubleshooting and diagnosing the behavior of an application is a fundamental operation in the lifecycle of every application. This is especially true if you are developing your application. Azure App Service provides you with some mechanisms for enabling diagnostics logging at different levels that can affect your application:

* **Web server diagnostics** These are message logs generated from the web server itself. You can enable three different types of logs:
  + **Detailed error logging** This log contains detailed information for any request that results in an HTTP status code 400 or greater. When an error 400 happens, a new HTML file is generated, containing all the information about the error. A separate HTML file is generated for each error. These files are stored in the file system of the instance in which the web app is running. A maximum of 50 error files can be stored. When this limit is reached, the oldest 26 files are automatically deleted from the file system.
  + **Failed request tracing** This log contains detailed information about failed requests to the server. This information contains a trace of the IIS components that were involved in processing the request. It also contains the time taken by each IIS component. These logs are stored in the file system. The system creates a new folder for each new error, applying the same retention policies as for detailed error logging.
  + **Web server logging** This log registers the HTTP transaction information for the requests made to the web server. The information is stored using the W3C extended log file format. You can configure custom retention policies to these log files. By default, these diagnostic logs are never deleted, but they are restricted by the space they can use in the file system. The default space quota is 35 MB.
* **Application diagnostics** You can send a log message directly from your code to the log system. You use the standard logging system of the language that you use in your app for sending messages to the application diagnostics logs. This is different from Application Insights because application diagnostics are just logged information that you register from 33your application. If you want your application to send logs to Application Insights, you need to add the Application Insights SDK to your application.
* **Deployment diagnostics** This log is automatically enabled for you, and it gathers all information related to the deployment of your application. Typically, you use this log for troubleshooting failures during the deployment process, especially if you are using custom deployment scripts.

You can enable the different diagnostics logs, shown in [Figure 1-5](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig05), using the Azure portal. When you enable application logging, you can select the level of error log that will be registered on the files. These error levels are

* **Disabled** No errors are registered.
* **Error** Critical and Error categories are registered.
* **Warning** Registers Warning, Error, and Critical categories.
* **Information** Registers Info, Warning, Error, and Critical log categories.
* **Verbose** Registers all log categories (Trace, Debug, Info, Warning, Error, and Critical).



**Figure 1-5** Enabling diagnostics logging

When you configure application logging, you can set the location for storing the log files. You can choose between saving the logs in the file system or using Blob Storage. Storing application logs in the file system is intended for debugging purposes. If you enable this option, it will be automatically disabled after 12 hours. If you need to enable the application logging for a longer period, you need to save the log files in Blob Storage. When you configure application logging for storing the log files in Blob Storage, you can also provide a retention period in days. When log files become older than the value that you configure for the retention period, the files are automatically deleted. By default, there is no retention period set. You can configure the web server logging in the same way that you configure the storage for your application logging.

If you configure application or web server logging for storing the log files in the file system, the system creates the following structure for the log files:

* **/LogFiles/Application/** This folder contains the logs files from the application logging.
* **/LogFiles/W3SVC#########/** This folder contains the files from the failed request traces. The folder contains an XSL file and several XML files. The XML files contain the actual tracing information, whereas the XSL file provides the formatting and filtering functionality for the content stored in the XML files.
* **/LogFiles/DetailedErrors/** This folder contains the \*.htm files related to the detailed error logs.
* **/LogFiles/http/RawLogs/** This folder contains the web server logs in W3C extended log format.
* **/LogFiles/Git** This folder contains the log generated during the deployment of the application. You can also find deployment files in the folder D:\home\site\deployments.

You need this folder structure when you want to download the log files. You can use two different mechanisms for downloading the log files: FTP/S or Azure CLI. The following command shows how to download log files to the current working directory:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg034a)

az webapp log download --resource-group <Resource group name> --name <App name>

The logs for the application *<App name>* are automatically compressed into a file named webapp\_logs.zip. Then, this file is downloaded in the same directory where you executed the command. You can use the optional parameter --log-file for downloading the log files to a different path in a different zip file.

There are situations in which you may need to view the logs for your application in near real time. For these situations, App Service provides you with log streams. Using streaming, you can see the log messages as they are being saved to the log files. Any text file stored in the D:\home\LogFiles\ folder is also displayed on the log stream. You can view log streams by using the embedded viewer in the Azure portal, on the Log Stream item under the monitoring section in your App Service. You can also use the following Azure CLI command for viewing your application or web server logs in streaming:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg034-1a)

az webapp log tail --resource-group <Resouce group name> --name <App name>

***NEED MORE REVIEW?* INTEGRATE LOGS WITH AZURE MONITOR**

You can also send the diagnostics information from your Windows or Linux App Services to Azure Monitor. At the time of this writing, this feature is in preview. You can get more information about how to integrate your Azure App Service logs with Azure Monitor by reviewing the following article: [*https://azure.github.io/AppService/2019/11/01/App-Service-Integration-with-Azure-Monitor.html*](https://azure.github.io/AppService/2019/11/01/App-Service-Integration-with-Azure-Monitor.html).

**Images *EXAM TIP***

When you are planning to configure the application logging, you should consider that not all the programming languages’ codes can write the log information in Blob Storage. You can use Blob Storage only with .NET application logs. If you use Java, PHP, Node.js, or Python, you need to use the application log file system option.

**Deploy code to a web app**

As part of the typical development lifecycle of your application, there is a point where you need to deploy your code to an Azure App Service. The “[Create an Azure App Service web app](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec7)” section earlier in this chapter reviews how to deploy the code directly from Visual Studio 2019. This section explains how to deploy your code using other alternatives more suitable to continuous deployment or continuous integration workflows.

When you are developing your web application, you need to test your code on both your local environment and in development or testing environments that are similar to the production environment. Starting with the Standard pricing tier, Azure App Service provides you with the deployment slots. These slots are deployments of your web application that reside in the same App Service of your web application. A deployment slot has its configuration and host name. You can use these additional deployment slots for testing your code before moving to the production slot. The main benefit of using these deployment slots is that you can swap these slots without any downtime. You can even configure an automated swap of the slots by using Auto Swap.

When you plan for deploying your web application into an App Service, Azure offers you several options:

* **ZIP or WAR files** When you want to deploy your application, you can package all your files into a ZIP or WAR package. Using the Kudu service, you can deploy your code to the App Service.
* **FTP** You can copy your application files directly to the App Service using the FTP/S endpoint configured by default in the App Service.
* **Cloud synchronization** Powered by the Kudu deployment engine, this method allows you to have your code in a OneDrive or Dropbox folder, and it syncs that folder with the App Service.
* **Continuous deployment** Azure can integrate with GitHub, BitBucket, or Azure Repos repositories for deploying the most recent updates of your application to the App Service. Depending on the service, you can use the Kudu build server, or Azure Pipelines for implementing a continuous delivery process. You can also configure the integration manually with other cloud repositories like GitLab.
* **Your local Git repository** You can configure your App Service as a remote repository for your local Git repository and push your code to Azure. Then the Kudu build server automatically compiles your code for you and deploys to the App Service.
* **ARM template** You can use Visual Studio and an ARM template for deploying your code into an App Service.

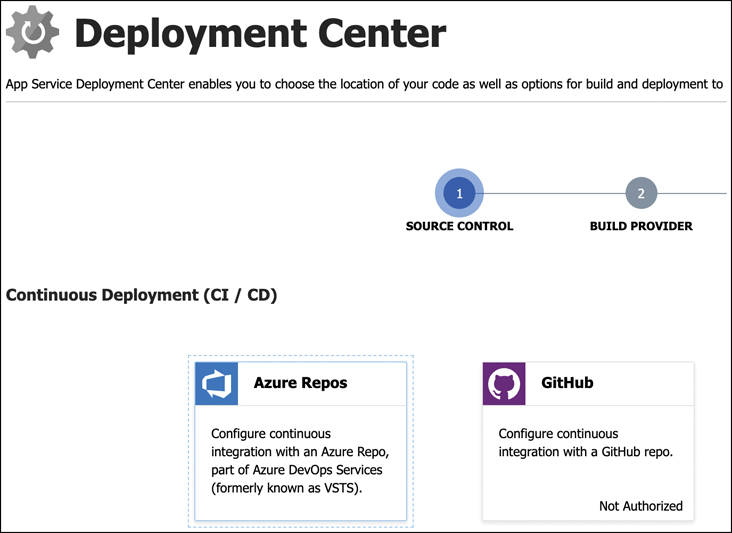
***NOTE* KUDU**

Kudu is the platform that is in charge of the Git deployments in Azure App Service. You can find more detailed information on its GitHub site at [*https://github.com/projectkudu/kudu/wiki*](https://github.com/projectkudu/kudu/wiki)*.*

The following example shows how to deploy your code to a web app using Azure Pipelines. For this example, you need to have your code deployed in an Azure Repos repository. If you don’t already have your code in an Azure Repos repository, you can use the following article for creating a new repo: [*https://docs.microsoft.com/en-us/azure/devops/repos/git/creatingrepo*](https://docs.microsoft.com/en-us/azure/devops/repos/git/creatingrepo).

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the search text box at the top of the Azure portal, type the name of your App Service.
3. In the result list below the search text box, click your App Service.
4. On your App Service blade, on the menu on the left side of the page, under the Deployment section, click Deployment Center.
5. On the Deployment Center, shown in [Figure 1-6](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig06), in the source control step, click Azure Repos.
6. At the bottom of the page, click Continue.
7. In the Build Provider step, click Azure Pipelines (Preview) and click Continue at the bottom of the page.
8. In the Configure step, on the Code section, select your organization in the Azure DevOps Organization drop-down menu.
9. In the Project drop-down menu, select the project with the repository that contains the code that you want to deploy to the Azure App Service.
10. In the Repository drop-down menu, select the repository that contains your code.
11. In the Branch drop-down menu, select the branch that you want to deploy.
12. In the Build section, in the Web Application Framework drop-down menu, select the appropriate framework for your code.
13. Click the Continue button at the bottom of the page.
14. In the Summary step, review the details for the configuration.
15. At the bottom of the page, click Finish.

At this point, you have configured an Azure Pipeline in your Azure Repo that automatically deploys your code to the Azure App Service. When you make a commit to the branch that you selected in the previous example, the Azure Pipeline automatically uses the code in the last commit. Once you have configured the continuous deployment for your Azure App Service, you can review the status of the different deployments in the Deployment Center of your Azure App Service.



**Figure 1-6** Enabling diagnostics logging

***NEED MORE REVIEW?* APP SERVICE DEPLOYMENT EXAMPLES**

You can review samples of the different types of deployments by following the cases published in the following articles:

* Deploy ZIP or WAR files: [*https://docs.microsoft.com/en-us/azure/app-service/deploy-zip*](https://docs.microsoft.com/en-us/azure/app-service/deploy-zip)
* Deploy via cloud sync: [*https://docs.microsoft.com/en-us/azure/app-service/deploy-content-sync*](https://docs.microsoft.com/en-us/azure/app-service/deploy-content-sync)
* Deploy from local Git: [*https://docs.microsoft.com/en-us/azure/app-service/deploy-local-git*](https://docs.microsoft.com/en-us/azure/app-service/deploy-local-git)

When you deploy your code to an Azure App Service, you can do it in different deployment slots. A deployment slot is a live app that is different from the main app. Each deployment slot has its own host name and group of settings. You usually use the various slots as a staging environment for testing purposes. You can switch between the different slots without losing requests. Deployment slots are available only to Standard, Premium, and Isolated App Services tiers.

***NEED MORE REVIEW?* DEPLOYMENT SLOTS**

You can learn more about how to work with deployment slots by reviewing the following article: [*https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots*](https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots)*.*

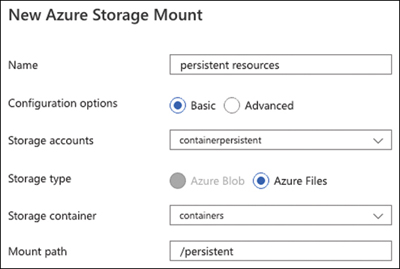
**Images *EXAM TIP***

You can use different mechanisms for deploying your code into an Azure App Service. If you decide to use continuous deployment systems for deploying your code, like Azure Repos, or GitHub, remember that you need to authorize your continuous deployment system before you can perform any deployment.

**Configure web app settings including SSL, API, and connection strings**

After you have created your App Service application, you can manage the various parameters that may affect your application. You can access these settings in the Configuration menu on the Settings section in the App Service blade. The available parameters are grouped by the following four main categories of settings:

* **Application Settings** You can configure the environment variables that are passed to your code. Using these settings is equivalent to setting the same variables in the <appSettings> section in the Web.config or appsettings.json files in an ASP.NET or ASP.NET Core project. If you set a variable in this section that matches a variable in Web.config or appsettings.json files, the value of the variables in the configuration files will be replaced with the value in your Azure Web App settings. These settings are always encrypted at rest, that is, when they are stored.
* **Connection Strings** You use this section for configuring the connection strings for the database that your code needs to use. This is similar to using the <connectionString> section in the Web.config or appsettings.json files in ASP.NET or ASP.NET Core projects.
* **General Settings** These settings are related to the environment and platform in which your app runs. You can control the following items:
  + **Stack Settings** You configure the stack and the version used for running your application.
    - **Stack** You can choose between .NET Core, .NET, Java, PHP, and Python.
    - **Version** This is the version for the stack that you chose in the previous setting.
  + **Platform Settings** This section controls the different settings related to the platform that runs your code:
    - **Platform** This setting controls whether your application runs on a 32- or 64-bit platform.
    - **Managed Pipeline Version** Configures the IIS pipeline mode. You should set this to *classic* if you need to run a legacy application that requires an older version of IIS.
    - **FTP State** Configures the possibility of using FTP or FTPS to deploy your web app to the Azure App Service. By default, both FTP and FTPS protocols are enabled.
    - **HTTP Version** This enables the HTTPS/2 protocol.
    - **Web Sockets** If your application uses SignalR or socket.io, you need to enable web sockets.
    - **Always On** Enabling this setting means your app is always loaded. By default, the application is unloaded if it is idle for some amount of time. You can configure this idle timeout in the host.json project file. The default value for App Service is 30 minutes.
    - **ARR Affinity** Enabling this setting ensures that client requests are routed to the same instance for the life of the session. This setting is useful for stateful applications but can negatively affect stateless applications.
  + **Debugging** Enable remote debugging options so you can directly connect from your IDE to the Azure App Service for debugging your ASP.NET, ASP.NET Core, or Node.js apps. This option automatically turns off after 48 hours.
  + **Incoming Client Certificates** If you require mutual SSL authentication for your application, you need to enable this option.
* **Default Documents** This setting configures which web page is displayed at the root URL of your app. You can set a list of different default documents, where the first valid match is shown at the root URL of your website.
* **Path Mappings** The settings in this section depend on the type of operating system that you choose for your Azure App Service:
  + **Windows Apps (Uncontainerized)** These settings are similar to the ones that you can find in IIS:
    - **Handler Mappings** You can configure custom script processors for different file extensions.
    - **Virtual Applications And Directories** This setting allows you to add additional virtual directories or applications to your App Service.
  + **Containerized Apps** You can configure the mount points that are attached to the containers during the execution. You can attach up to five Azure files or blob mount points per app. [Figure 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig07) shows the dialog box for configuring an Azure File mount point.



**Figure 1-7** Enabling diagnostics logging

Once you have created an app setting or connection string variable, you can access these values from your code by using environment variables. The following code snippet shows how to access an app setting named testing-var1 and connection string named testing-connsql1 from a PHP page:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg040a)

<?php

$testing\_var1 = getenv('APPSETTING\_testing-var1')

$connection\_string = getenv('SQLAZURECONNSTR\_testing-connsql1')

?>

As you can see in the previous code snippet, you need to prepend the string APPSETTING\_ to your app setting variable’s name. In the case of connection strings, the string that you need to prepend to your connection string’s name depends on the type that you configure in the connection string in the Azure portal:

* **SQL Databases** SQLAZURECONNSTR\_
* **SQL Server** SQLCONNSTR\_
* **MySQL** MYSQLCONNSTR\_
* **PostgreSQL** POSTGRESQLCONNSTR\_
* **Custom** CUSTOMCONNSTR\_

For ASP.NET applications, you can also access app settings and connection strings by using the traditional ConfigurationManager. If you decide to use the ConfigurationManager, you don’t need to prepend any string to the name of your app setting or connection string. The following code snippet shows how to access your app settings or connection string from ASP.NET code:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg040-2a)

System.Configuration.ConfigurationManager.AppSettings["testing-var1"]

System.Configuration.ConfigurationManager.ConnectionStrings["testing-connsql1"]

When you are configuring an Azure web app for a production environment, you usually need to secure the connections with the web app. You also need to make your Azure web app available through your own domain instead of the default azurewebsites.net domain. You can do it by configuring the Custom Domain and SSL settings.

Use the following procedure for configuring SSL settings for an existing web app. Remember that SSL settings are available only for B1 or higher pricing tiers:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the Search Resources text box, type the name of your Azure web app.
3. In the result list, click the name of your Azure web app.
4. On your Azure web app page, on the navigation list on the left side of the page, click Custom Domain in the Settings section.
5. Click the Add Custom Domain button in the middle of the Custom Domains page.
6. On the Add Custom Domain blade on the right side of the page, in the Custom Domain text box, type a name for your domain.
7. Click the validate button and follow the instructions for validating your domain.
8. Once you have validated your custom domain, click the Add Custom Domain button in the Add Custom Domain blade.
9. Click TLS/SSL Settings in the Settings section on the left side of your Azure Web App page.
10. In the TLS/SSL Settings page, in the TLS/SSL Bindings sections, click Add TLS/SSL Binding. Note that you need the appropriate pfx certificate for configuring this binding. You can import an existing certificate or buy a new one.
11. In the TLS/SSL Binding blade, in the Custom Domain drop-down menu, select the custom domain that you added in step 8.
12. In the Private Certificate Thumbprint drop-down menu, select a valid certificate for your custom domain.
13. In the TLS/SSL Type drop-down menu, select SNI SSL.
14. At the bottom of the TLS/SSL Binding blade, click the Add Binding button.

***NOTE* AZURE STORAGE IN APP SERVICE**

At the time of this writing, using Azure Storage in App Services is a feature that is in preview and should not be used in production environments.

***NEED MORE REVIEW?* CONFIGURE APP SETTINGS**

You can review more details about how to configure the different settings in your Azure web app by reviewing the article at [*https://docs.microsoft.com/en-us/azure/app-service/configure-common*](https://docs.microsoft.com/en-us/azure/app-service/configure-common).

**Images *EXAM TIP***

Remember that the settings that you configure in the Application Settings section overwrite the values that you configure in the <appSettings> or <connectionStrings> in your Web.config or appsettings.json files.

**Implement autoscaling rules, including scheduled autoscaling, and scaling by operational or system metrics**

One of the biggest challenges that you face when you deploy your application in a production environment is to ensure that you provide enough resources, so your application has the expected performance. Determining the number of resources you should allocate is the big question when it comes to configuring the resources for your app. If you allocate too many resources, your application will perform well during usage peaks, but you are potentially wasting resources. If you allocate fewer resources, you are saving resources, but your app may not perform well during usage peaks. Another issue with the application performance is that it’s challenging to anticipate when a heavy 42usage peak may happen. This statement is especially true for applications that have unpredictable usage patterns.

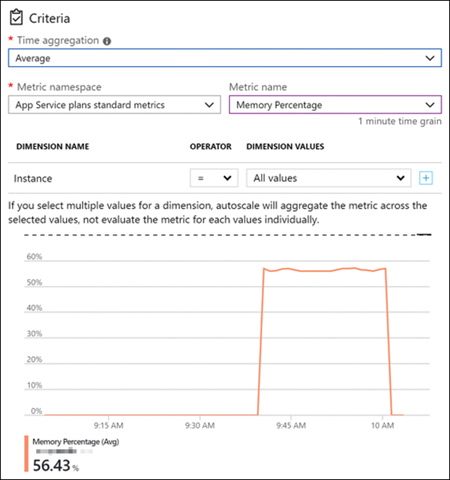
Fortunately, Azure provides a mechanism for addressing this issue. You can dynamically assign more resources to your application when you need them. Autoscaling is the action of automatically adding or removing resources to an Azure service and providing needed computing power for your application in each situation. An application can scale in two different ways:

* **Vertically** You add more computing power by adding more memory, CPU resources, and IOPS to the application. At the end of the day, your application runs on a virtual machine. It doesn’t matter if you use an IaaS virtual machine, Azure App Service, or Azure Service Fabric, you are using virtual machines under the hood. Vertically scaling an application means moving from a smaller VM to a larger VM and adding more memory, CPU, and IOPS. Vertically scaling requires stopping the system while the VM is resizing. This type of scaling is also known as “scaling up and down.”
* **Horizontally** You can also scale your application by creating or removing instances of your application. Each instance of your application is executed in a virtual machine that is part of a virtual machine scale set. The corresponding Azure service automatically manages for you the virtual machines in the scale set. All these instances of your application work together to provide the same service. The advantage of scaling horizontally is that the availability of your application is not affected because there is no need for rebooting all the instances of your application that provide the service. This type of scaling is also known as “scaling out and in.”

When you work with autoscaling, we refer to horizontal scaling because vertical scaling requires the service interruption while the Azure Resource Manager is changing the size of the virtual machine. For that reason, vertical scaling is not suitable for autoscaling.

You configure autoscaling based on some criteria that your application should meet for providing the right performance level. You configure these criteria in Azure by using autoscaling rules. A rule defines which metric should use Azure Monitor for performing the autoscaling. When that metric reaches the configured condition, Azure automatically performs the action configured for that rule. The typical action that you may think the rule should do is adding or removing a VM to the scale set, but it also can perform other actions like sending an email or making an HTTP request to a webhook. You can configure three different types of rules when working the autoscaling rules:

* **Time-based** The Azure Monitor executes the autoscaling rule based on a schedule. For example, if your application requires more resources during the first week of the month, you can add more instances and reduce the number of resources for the rest of the month.
* **Metric-based** You configure the threshold for standard metrics, such as the usage of the CPU, the length of the HTTP queue, or the percentage of memory usage, as shown in [Figure 1-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig08).
* **Custom-based** You can create your metrics in your application, expose them using Application Insight, and use them for autoscaling rules.



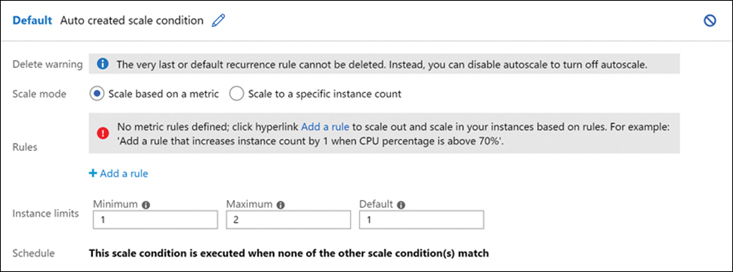
**Figure 1-8** Configuring a metric-based autoscale rule

You can only use the built-in autoscaling mechanism with a limited group of Azure resource types:

* **Azure virtual machines** You can apply autoscaling by using virtual machine scale sets. All the VMs in a scale set are treated as a group. By using autoscaling, you can add virtual machines to the scale set or remove virtual machines from it.
* **Azure Service Fabric** When you create an Azure Service Fabric cluster, you define different node types. A different virtual machine scale set supports each node type that you define in an Azure Service Fabric cluster. You can apply the same type of autoscaling rules that you use in a standard virtual machine scale set.
* **Azure App Service** This service has built-in autoscaling capabilities that you can use for adding or removing instances to the Azure App Service. The autoscale rules apply to all apps inside the Azure App Service.
* **Azure Cloud Services** This service has built-in autoscaling capabilities that you can use for adding or removing resources to the roles in the Azure Cloud Service.

When you work with the autoscale feature in one of the supported Azure Service, you define a profile condition. A profile condition defines the rule that you configure for adding or removing resources. You can also define the default, minimum, and maximum allowed instances for this profile. When you define a minimum and maximum, your service cannot decrease or grow beyond the limits you define in the profile. You can also configure the profile for scaling based on a schedule or 44based on the values of built-in or custom metrics. You can use the following procedure for adding a metric-based autoscale rule to an Azure App Service. This rule adds an instance to the Azure App Service plan when the average percentage of CPU usage is over 80 percent more than 10 minutes:

1. Open the Azure portal ([*https://portal.azure.com*](https://portal.azure.com/)).
2. In the search text box at the top of the Azure portal, type the name of your Azure App Service.
3. Click the name of your Azure App Service in the results list.
4. On your Azure App Service blade, on the navigation menu on the left side of the blade, click the Scale-Out (App Service Plan) option in the Settings section.
5. On the Scale-Out (App Service Plan) blade, on the Configure tab, click the Custom Autoscale button. Autoscale rules are available only for the App Service plans that are Standard size or bigger.
6. On the Scale-Out (App Service Plan) blade, on the Configure tab, in the Default Auto Created Scale Condition window shown in [Figure 1-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig09), click the Add A Rule link.



**Figure 1-9** Configuring a metric-based autoscale rule

1. On the Scale rule panel, in the Criteria section, ensure that the CPU Percentage value is selected in the Metric Name drop-down menu.
2. Ensure that the Greater Than value is selected from the Operator drop-down menu.
3. Type the value **80** in the Metric Threshold To Trigger Scale Action text box.
4. In the Action section, ensure that the Instance count value is set to **1**.
5. Click the Add button at the bottom of the panel.
6. On the Scale-Out (App Service Plan) blade, in the Default Profile condition, set the Maximum Instance Limit to **3**.
7. Click the Save button in the top-left corner of the blade.

***NOTE* SCALE-OUT/SCALE-IN**

The previous procedure shows how to add an instance to the App Service plan (it is scaling out the App Service plan) but doesn’t remove the additional instance once the CPU percentage falls below the configured threshold. You should add a Scale-In rule for removing the additional instances once they are not needed. You configure a Scale-In rule in the same way you did it if for the Scale-Out rule. Just set the Operation drop-down menu to the Decrease Count To value.

You can use different common autoscale patterns, based on the settings that I have reviewed so far:

* **Scale based on CPU** You scale your service (Azure App Service, VM Scale Set, or Cloud Service) based on CPU. You need to configure Scale-Out and Scale-In rules for adding and removing instances to the service. In this pattern, you also set a minimum and a maximum number of instances.
* **Scale differently on weekdays versus weekends** You use this pattern when you expect to have the primary usage of your application occur on weekdays. You configure the default profile condition with a fixed number of instances. Then you configure another profile condition for reducing the number for instances during weekends.
* **Scale differently during holidays** You use the Scale based on CPU pattern. Still, you add a profile condition for adding additional instances during holidays or days that are important to your business.
* **Scale based on custom metrics** You use this pattern with a web application comprised of three layers: front end, back end, and API tiers. The front end of an API tier communicates with the back-end tier. You define your custom metrics in the web application and expose them to the Azure Monitor by using Application Insights. You can then use these custom metrics for adding more resources to any of the three layers.

**Images *EXAM TIP***

Autoscaling allows you to assign resources to your application in an efficient way. Autoscale rules for adding more instances to your application do not remove those instances when the rule condition is not satisfied. As a best practice, if you create a scale-out rule for adding instances to a service, you should create the opposite scale-in rule for removing the instance. This ensures that the resources are assigned efficiently to your application.

***NEED MORE REVIEW?* AUTOSCALE BEST PRACTICES**

You can find more information about best practices when configuring autoscale rules by reviewing the article at [*https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-best-practices*](https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-best-practices)*.*

***NEED MORE REVIEW?* APPLICATION DESIGN CONSIDERATIONS**

Simply adding more resources to your application doesn’t guarantee that your application is going to perform well. Your application needs to be aware of the new resources to take advantage of them. You can review some application design considerations reviewing the article at [*https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling#related-patterns-and-guidance*](https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling#related-patterns-and-guidance)*.*

**Skill 1.3: Implement Azure Functions**

Based on Azure App Service, Azure Functions allow you to run pieces of code that solve particular problems inside the whole application. You use these functions in the same way that you may use a class or a function inside your code. That is, your function gets some input, executes the piece of code, and provides an output.

The big difference between Azure Functions and other app services models is that with Azure Functions (using the Consumption pricing tier), you are charged per second only when your code is running. If you use App Service, you are charged hourly when the App Service Plan is running—even if there is no code executing. Because Azure Functions is based on App Service, you can also decide to run your Azure Function in your App Service Plan if you already have other app services executing.

**This skill covers how to**

* [Implement input and output bindings for a function](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec12)
* [Implement function triggers by using data operations, timers, and webhooks](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec13)
* [Implement Azure Durable Functions](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01lev2sec14)

**Implement input and output bindings for a function**

When you are writing a function in your code, that function may require data as input information for doing the job that you are writing. The function can also produce some output information as the result of the operations performed inside the function. When you work with Azure Functions, you may also need these input and output flows of data.

Binding uses Azure Functions for connecting your function with the external world without hard-coding the connection to the external resources. An Azure Function can have a mix of input and output bindings, or it can have no binding at all. Bindings pass data to the function as parameters.

Although triggers and bindings are closely related, you should not confuse them. Triggers are the events that cause the function to start its execution; bindings are like the connection to the data needed for the function. You can see the difference in this example:

One publisher service sends an event (to an Event Grid that reads a new image that has been uploaded to Blob Storage) to an Azure Storage account. Your function needs to read this image, process it, and place some information in a Cosmos DB document. When the image has been processed, your function also sends a notification to the user interface using SignalR.

In this example, you can find one trigger, one input binding, and two output bindings:

* **Trigger** The Event Grid should be configured as the trigger for the Azure Function.
* **Input binding** Your function needs to read the image that has been uploaded to the Blob Storage. In this case, you need to use Blob Storage as an input binding.
* **Output bindings** Your function needs to write a Cosmos DB document with the results of processing the image. You need to use the Cosmos DB output binding. Your function also needs to send a notification to the user interface using the SignalR output binding.

Depending on the language that you use for programming your Azure Function, the way you declare a binding changes:

* **C#** You declare bindings and triggers by decorating methods and parameters.
* **Other** Update the function.json configuration file.

When defining a binding for non-C# language functions, you need to define your binding using the following minimum required attributes:

* **type** This string represents the binding type. For example, you would use eventHub when using an output binding for Event Hub.
* **direction** The only allowed values are in for input bindings and out for output bindings. Some bindings also support the special direction inout.
* **name** The function uses this attribute for binding the data in the function. For example, in JavaScript, the key in a key-value list is an attribute.

Depending on the specific binding that you are configuring, there could be some additional attributes that should be defined.

***NOTE* SUPPORTED BINDINGS**

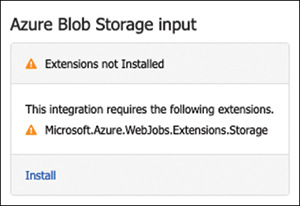
For a complete list of supported bindings, please refer to the article at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-bindings*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-bindings)*.*

Before you can use a binding in your code, you need to register it. If you are using C# for your functions, you can do this by installing the appropriate NuGet package. For other languages, you need to install the package with the extension code using the func command-line utility. The following example installs the Service Bus extension in your local environment for non-C# projects:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg047a)

func extensions install –package Microsoft.Azure.WebJobs.ServiceBus

If you are developing your Azure Function using the Azure portal, you can add the bindings in the Integrate section of your function. When you add a binding that is not installed in your environment, you will see the warning message shown in [Figure 1-10](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig10). You can install the extension by clicking the Install link.



**Figure 1-10** Installing a binding extension

***NEED MORE REVIEW?* MANUALLY INSTALL BINDING EXTENSIONS FROM THE AZURE PORTAL**

When you develop your Azure Function using the Azure portal, you can use the standard editor or the advanced editor. When you use the advanced editor, you can directly edit the function.json configuration file. If you add new bindings using the advanced editor, you need to manually install any new binding extensions that you added to the function.json. You can review the following article for manually installing binding extensions from the Azure portal at [*https://docs.microsoft.com/en-us/azure/azure-functions/install-update-binding-extensions-manual*](https://docs.microsoft.com/en-us/azure/azure-functions/install-update-binding-extensions-manual).

If you decide to program your Azure Function using C#, you make the configuration of the bindings by using decorators for function and parameters. The function.json file is automatically constructed based on the information that you provide in your code. [Listing 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex07) shows how to configure input and output bindings using parameter decorators.

**Listing 1-7** Configuring input and output bindings

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-7a)

// C# ASP.NET Core

using System;

using System.IO;

using Microsoft.Azure.WebJobs;

using Microsoft.Extensions.Logging;

using Microsoft.Azure.WebJobs.Extensions.SignalRService;

using Microsoft.Azure.WebJobs.Extensions.EventGrid;

using Microsoft.Azure.EventGrid.Models;

using System.Threading.Tasks;

namespace Company.Functions

{

public static class BlobTriggerCSharp

{

[FunctionName("BlobTriggerCSharp")]

public static Task Run(

[EventGridTrigger]EventGridEvent eventGridEvent,

[Blob("{data.url}", FileAccess.Read, Connection = "ImagesBlobStorage")] Stream

imageBlob,

[CosmosDB(

databaseName: "GIS",

collectionName: "Processed\_images",

ConnectionStringSetting = "CosmosDBConnection")] out dynamic document,

[SignalR(HubName = "notifications")]IAsyncCollector<SignalRMessage> signalRMessages,

ILogger log)

{

document = new { Description = eventGridEvent.Topic,

id = Guid.NewGuid() };

log.LogInformation($"C# Blob trigger function Processed event\n Topic: {eventGridEvent.

Topic} \n Subject: {eventGridEvent.Subject} ");

return signalRMessages.AddAsync(

new SignalRMessage

{

Target = "newMessage",

Arguments = new [] { eventGridEvent.Subject }

});

}

}

}

Let’s review the portions of [Listing 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex07) that are related to the binding configuration. In this example, you configured one input binding and two output bindings. The parameter imageBlob is configured as an input binding. You have decorated the parameter with the attribute Blob, which takes the following parameters:

* **Path** The value {data.url} configures the path of the blobs that are passed to the function. In this case, you are using a binding expression that resolves to the full path of the blob in the Blob Storage.
* **Blob access mode** In this example, you access the blob in read-only mode.
* **Connection** This sets the connection string to the storage account where the blobs are stored. This parameter sets the app setting name that contains the actual connection string.

You have also configured two output bindings, though you have configured them differently. The first output binding is configured using the keyword out in the parameter definition. Just as you did with the input parameter, you configured the output parameter document by using a parameter attribute. In this case, you used the CosmosDB attribute. You use the following parameters for configuring this output binding:

* **databaseName** Sets the database in which you save the document that you create during the execution of the function.
* **collectionName** Sets the collection in which you save the generated document.
* **ConnectionStringSetting** Sets the name of the app setting variable that contains the actual connection string for the database. You should not put the actual connection string here.

Setting a value for this output binding is as simple as assigning a value to the parameter document. You can also configure output bindings by using the return statement of the function. In the example, you configure the second output binding this way.

The function parameter signalRMessages is your second output binding. As you can see in [Listing 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex07), you didn’t add the out keyword to this parameter because you can return multiple output values. When you need to return multiple output values, you need to use ICollector or IAsyncCollector types with the output binding parameter, as you did with signalRMessages. Inside the function, you add needed values to the signalRMessages collection and use this collection as the return value of the function. You used the SignalR parameter attribute for configuring this output binding. In this case, you only used one parameter for configuring the output binding.

* **HubName** This is the name of the SignalR hub where you send your messages.
* **ConnectionStringSetting** In this case, you didn’t use this parameter, so it uses its default value AzureSignalRConnectionString. As you saw in the other bindings, this parameter sets the name of the app setting variable that contains the actual connection string SignalR.

When you are configuring bindings or triggers, there are situations when you need to map the trigger or binding to a dynamically generated path or element. In these situations, you can use binding expressions. You define a binding expression by wrapping your expression in curly braces. You can see an example of a binding expression shown previously in [Listing 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex07). The path that you configure for the input binding contains the binding expression {data.url}, which resolves to the full path of the blob in the Blob Storage. In this case, EventGridTrigger sends a JSON payload to the input binding that contains the data.url attribute.

***NEED MORE REVIEW?* BINDING EXPRESSION PATTERNS**

You can learn about more binding expression patterns by reviewing this article about Azure Functions binding expression patterns in Microsoft Docs at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-expressions-patterns).

The way you configure the bindings for your code depends on the language that you used for your Azure Function. In the previous example, you review how to configure input and output bindings using C# and parameter decorations. If you use any of the other supported languages in your Azure Function, the way you configure input and output bindings changes.

The first step when configuring bindings in non-C# languages is to modify the function.json configuration file. [Listing 1-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex08) shows the equivalent function.json for the binding configuration made in [Listing 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex07). Once you have configured your bindings, you can write your code to access the bindings that you configured. [Listing 1-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex09) shows an example written in JavaScript for using bindings in your code.

**Listing 1-8** Configuring input and output bindings in function.json

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-8a)

{

"disabled": false,

"bindings": [

{

"name": "eventGridEvent",

"type": "eventGridTrigger",

"direction": "in"

},

{

"name": "imageBlob",

"type": "blob",

"connection": "ImagesBlobStorage",

"direction": "in",

"path": "{data.url}"

},

{

"name": "document",

"type": "cosmosDB",

"direction": "out",

"databaseName": "GIS",

"collectionName": "Processed\_images",

"connectionStringSetting": "CosmosDBConnection",

"createIfNotExists": true

},

{

"name": "signalRMessages",

"type": "signalR",

"direction": "out",

"hubName": "notifications"

}

]

}

**Listing 1-9** Using bindings in JavaScript

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-9a)

// NodeJS. Index.js

const uuid = require('uuid/v4');

module.exports = async function (context, eventGridEvent) {

context.log('JavaScript Event Grid trigger function processed a request.');

context.log("Subject: " + eventGridEvent.subject);

context.log("Time: " + eventGridEvent.eventTime);

context.log("Data: " + JSON.stringify(eventGridEvent.data));

context.bindings.document = JSON.stringify({

id: uuid(),

Description: eventGridEvent.topic

});

context.bindings.signalRMessages = [{

"target": "newMessage",

"arguments": [ eventGridEvent.subject ]

}];

context.done();

};

[Listings 1-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex08) and [1-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex09) represent the equivalent code in JavaScript to the code in the C# code shown in [Listing 1-7](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex07). Most important is that name attributes in the binding definitions shown in [Listing 1-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex08) correspond to the properties of the context object shown in [Listing 1-9](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex09). For example, you created a Cosmos DB output binding and assigned the value document to the name attribute in the binding definition in [Listing 1-8](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex08). In your JavaScript code, you access this output binding by using context.bindings.document.

**Images *EXAM TIP***

Remember that you need to install the extensions on your local environment before you can use bindings or triggers. You can use the func command-line command from the Azure Function CLI tools.

**Implement function triggers by using data operations, timers, and webhooks**

When you create an Azure Function, that function is executed based on events that happen in the external world. Some examples include

* Executing a function periodically
* Executing a function when some other process uploads a file to Blob Storage or sends a message to a queue storage
* Executing a function when an email arrives in Outlook

Triggers programmatically manage all these events.

You can configure function triggers in the same way that you configure input or output bindings, but you need to pay attention to some additional details when dealing with triggers. You configure a trigger for listening to specific events. When an event happens, the trigger object can send data and information to the function.

You can configure three different types of triggers:

* **data operation** The trigger is started based on new data that is created, updated, or added to the system. Supported systems are Cosmos DB, Event Grid, Event Hub, Blob Storage, Queue Storage, and Service Bus.
* **timers** You use this kind of trigger when you need to run your function based on a schedule.
* **webhooks** You use HTTP or webhooks triggers when you need to run your function based on an HTTP Request.

Triggers send data to the function with information about the event that caused the trigger to start. This information depends on the type of trigger. [Listing 1-10](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex10) shows how to configure a data operation trigger for Cosmos DB.

**Listing 1-10** Configuring a Cosmos DB trigger

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-10a)

// C# ASP.NET Core

using System.Collections.Generic;

using Microsoft.Azure.Documents;

using Microsoft.Azure.WebJobs;

using Microsoft.Azure.WebJobs.Host;

using Microsoft.Extensions.Logging;

namespace Company.Function

{

public static class CosmosDBTriggerCSharp

{

[FunctionName("CosmosDBTriggerCSharp")]

public static void Run([CosmosDBTrigger(

databaseName: "databaseName",

collectionName: "collectionName",

ConnectionStringSetting = "AzureWebJobsStorage",

LeaseCollectionName = "leases",

CreateLeaseCollectionIfNotExists = true)]IReadOnlyList<Document> input, ILogger log)

{

if (input != null && input.Count > 0)

{

log.LogInformation("Documents modified " + input.Count);

log.LogInformation("First document Id " + input[0].Id);

log.LogInformation("Modified document: " + input[0]);

}

}

}

}

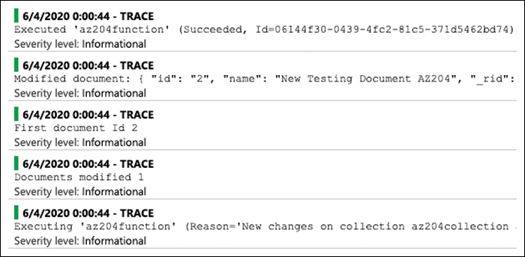
***IMPORTANT* WORKING WITH LEASES COLLECTION**

At the time of this writing, Cosmos DB trigger does not support working with a partitioned lease collection. Microsoft is removing the ability to create a nonpartitioned collection using Azure portal. You can still create your nonpartitioned collections using SDKs. Cosmos DB trigger requires a second collection to store leases over partitions. Both collections—leases and the collection that you want to monitor—need to exist before your code runs. To ensure that the lease collection is correctly created as a nonpartitioned collection, don’t create the collection using the Azure portal, and set the trigger parameter CreateLeaseCollectionIfNotExists to true.

Just as with bindings, you need to install the corresponding NuGet package with the appropriate extension for working with triggers. In this case, you need to install the package Microsoft.Azure.WebJobs.Extensions.CosmosDB. You used the CosmosDBTrigger parameter attribute for configuring the trigger with the following parameters:

* **databaseName** This is the name of the database that contains the collection this trigger should monitor.
* **collectionName** This is the name of the collection that this trigger should monitor. This collection needs to exist before your function runs.
* **ConnectionStringSetting** This is the name of the app setting variable that contains the connection string to the Cosmos DB database. If you want to debug your function in your local environment, you should configure this variable in the file local.settings.json file and assign the value of the connection string to your development CosmosDB database. This local.settings.json file is used by Azure Functions Core Tools to store app settings, connection strings, and settings locally and won’t be automatically uploaded to Azure when you publish your Azure Function.
* **LeaseCollectionName** This is the name of the collection used for storing leases over partitions. By default, this collection is stored in the same database as the collectionName. If you need to store this collection in a separate database, use the parameter leaseDatabaseName or leaseConnectionStringSetting if you need to store the database in a separate Cosmos DB account.
* **CreateLeaseCollectionIfNotExists** This creates the lease collection set by the LeaseCollectionName parameter if it does not exist in the database. Lease collection should be a nonpartitioned collection and needs to exist before your function runs.

The Cosmos DB trigger monitors for new or updated documents in the database that you configure in the parameters of the trigger. Once the trigger detects a change, it passes detected changes to the function using an IReadOnlyList<Document>. Once you have the information provided by the trigger in the input list, you can process the information inside your function. If you have enabled Application Insight integration, you should be able to see the log messages from your function, as shown in [Figure 1-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig11).



**Figure 1-11** View Azure Function logs in Application Insight

***NOTE* VERSION 1.0 VERSUS VERSION 2.0 VERSUS VERSION 3.0**

When you work with Azure Functions, you can choose between versions 1.0, 2.0, and 3.0. The main difference between the versions 1.0 and the other versions is that you can only develop and host Azure Functions 1.0 on Azure portal or Windows computers. Functions 2.0 and 3.0 can be developed and hosted on all platforms supported by .NET Core. The Azure Function you use affects the extension packages that you need to install when configuring triggers and bindings. Review the overview of Azure Functions runtime versions at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions).

When you work with timer and webhooks triggers, the main difference between them and a data operations trigger is that you do not need to install the extension package that supports the trigger explicitly.

Timer triggers execute your function based on a schedule. This schedule is configured using a CRON expression that is interpreted by the NCronTab library. A CRON expression is a string compound of six different fields with this structure:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg055a)

{second} {minute} {hour} {day} {month} {day-of-week}

Each field can have numeric values that are meaningful for the field:

* **second** Represents the seconds in a minute. You can assign values from 0 to 59.
* **minute** Represents the minutes in an hour. You can assign values from 0 to 59.
* **hour** Represents the hours in a day. You can assign values from 0 to 23.
* **day** Represents the days in a month. You can assign values from 1 to 31.
* **month** Represents the months in a year. You can assign values from 1 to 12. You can also use names in English, such as January, or you can use abbreviations of the name in English, such as Jan. Names are case-insensitive.
* **day-of-week** Represents the days of the week. You can assign values from 0 to 6, where 0 is Sunday. You can also use names in English, such as Monday, or you can use abbreviations of the name in English, such as Mon. Names are case-insensitive.

All fields need to be present in a CRON expression. If you don’t want to provide a value to a field, you can use the asterisk character \*. This means that the expression uses all available values for that field. For example, the CRON expression \* \* \* \* \* \* means that the trigger is executed every second, in every minute, in every hour, in every day, and every month of the year. You can also use some operators with the allowed values in fields:

* **Range of values** Use the dash operator (–) for representing all the values available between two limits. For example, the expression 0 10–12 \* \* \* \* means that the function is executed at hh:10:00, hh:11:00, and hh:12:00 where hh means every hour. That is, it is executed three times every hour.
* **Set of values** Use the comma operator (,) for representing a set of values. For example, the expression 0 0 11,12,13 \* \* \* means that the function will be executed three times a day, every day, once at 11:00:00, a second time at 12:00:00, and finally at 13:00:00.
* **Interval of values** Use the forward slash operator (/) for representing an interval of values. The function is executed when the value of the field is divisible by the value that you put on the right side of the operator. For example, the expression \*/5 \* \* \* \* \* will execute the function every five seconds.

[Listings 1-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex11) and [1-12](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex12) show how to configure a timer trigger and how to use the trigger with JavaScript code.

**Listing 1-11** Configuring a timer trigger in function.json

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-11a)

{

"disabled": false,

"bindings": [

{

"name": "myTimer",

"type": "timerTrigger",

"direction": "in",

"schedule": "0 \*/5 \* \* \* \*",

"useMonitor": true,

"runOnStartup": true

}

]

}

**Listing 1-12** Using a timer trigger with JavaScript

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-12a)

//NodeJS. Index.js file

module.exports = async function (context, myTimer) {

var timeStamp = new Date().toISOString();

if(myTimer.isPastDue)

{

context.log('JavaScript is running late!');

}

context.log('JavaScript timer trigger Last execution: ', myTimer.ScheduleStatus.

Last);

context.log('JavaScript timer trigger Next execution: ', myTimer.ScheduleStatus.

Next);

};

Just as you did when you configured bindings in the previous section, when you configure a trigger for non-C# languages, you need to add them to the function.json configuration file. You configure your triggers in the bindings section. [Listing 1-11](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex11) shows the appropriate properties for configuring a timer trigger:

* **name** This is the name of the variable that you use on your JavaScript code for accessing the information from the trigger.
* **type** This is the type of trigger that you are configuring. In this example, the value for the timer trigger is timerTrigger.
* **direction** This is always included in a trigger.
* **schedule** This is the CRON expression used for configuring the execution scheduling of your function. You can also use a TimeSpan expression.
* **useMonitor** This property monitors the schedule even if the function app instance is restarted. The default value for this property is true for every schedule with a recurrence greater than one minute. Monitoring the schedule occurrences will ensure that the schedule is maintained correctly.
* **runOnStartup** This indicates that the function should be invoked as soon as the runtime starts. The function will be executed after the function app wakes up after going idle because of inactivity or if the function app restarts because of changes in the function. Setting this parameter to true is not recommended on production environments because it can lead to unpredictable execution times of your function.

***NOTE* TROUBLESHOOTING FUNCTIONS ON YOUR LOCAL ENVIRONMENT**

While you are developing your Azure Functions, you need to troubleshoot your code in your local environment. If you are using non-HTML triggers, you need to provide a valid value for the AzureWebJobsStorage attribute in the local.settings.json file .

You use TimeSpan expressions to specify the time interval between the invocations of the function. If the function execution takes longer than the specified interval, then the function is invoked immediately after the previous invocation finishes. TimeSpan expressions are strings with the format hh:mm:ss where hh represents hours, mm represents minutes, and ss represents seconds. Hours in a TimeSpan expression need to be less than 24. The TimeSpan expression 24:00:00 means the function is going to be executed every day. 02:00:00 means the function will be invoked every two hours. You can use TimeSpan expressions only on Azure Functions that are executed on App Service Plans. That is, you cannot use TimeSpan expressions when you are using the Consumption pricing tier.

You use HTTP triggers for running your Azure Function when an external process makes an HTTP request. This HTTP request can be a regular request using any of the available HTTP methods or a webhook. A web callback or webhook is an HTTP request made by third-party systems, or external web applications, or as a result of an event generated in the external system. For example, if you are using GitHub as your code repository, GitHub can send a webhook to your Azure Function each time a new pull request is opened.

When you create an Azure Function using HTTP triggers, the runtime automatically publishes an endpoint with the following structure:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg058a)

http://<*your\_function\_app*>.azurewebsites.net/api/<*your\_function\_name*>

This is the URL or endpoint that you need to use when calling to your function using a regular HTTP request or when you configure an external webhook for invoking your function. You can customize the route of this endpoint by using the appropriate configuration properties. This means that you can also implement serverless APIs using HTTP triggers. You can even protect the access to your function’s endpoints by requesting authorization for any request made to your API using the App Service Authentication/Authorization. [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13) shows how to configure an HTTP trigger with a custom endpoint.

**Listing 1-13** Configuring an HTTP trigger

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-13a)

// C# ASP.NET Core

using System.Security.Claims;

using System;

using System.IO;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Azure.WebJobs;

using Microsoft.Azure.WebJobs.Extensions.Http;

using Microsoft.AspNetCore.Http;

using Microsoft.Extensions.Logging;

using Newtonsoft.Json;

namespace Company.Function

{

public static class HttpTriggerCSharp

{

[FunctionName("HttpTriggerCSharp")]

public static async Task<IActionResult> Run(

[HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = "devices/

{id:int?}")] HttpRequest req,

int? id,

ILogger log)

{

log.LogInformation("C# HTTP trigger function processed a request.");

//We access to the parameter in the address by adding a function parameter //with the same name

log.LogInformation($"Requesting information for device {id}");

//If you enable Authentication / Authorization at Function App level,

//information

//about the authenticated user is automatically provided in the

//HttpContext

ClaimsPrincipal identities = req.HttpContext.User;

string username = identities.Identity?.Name;

log.LogInformation($"Request made by user {username}");

string name = req.Query["name"];

string requestBody = await new StreamReader(req.Body).ReadToEndAsync();

dynamic data = JsonConvert.DeserializeObject(requestBody);

name = name ?? data?.name;

//We customize the output binding

return name != null

? (ActionResult)new JsonResult(new { message = $"Hello, {name}",

username = username, device = id})

: new BadRequestObjectResult("Please pass a name on the query string or

in the request body");

}

}

}

The example in [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13) shows the following points when working with HTTP triggers:

* How to work with authentication.
* How to work with the authorization level.
* How to customize the function endpoint, using route parameters.
* How to customize the output binding.

HTTP triggers are automatically provided to you out-of-the-box with the function runtime. There is no need to install a specific NuGet package for working with this extension. You use the HTTPTrigger parameter attribute for configuring the HTTP trigger. This trigger accepts the following parameters:

* **AuthLevel** This parameter configures the authorization key that you should use for accessing the function. Allowed values are
  + **anonymous** No key is required.
  + **function** This is the default value. You need to provide a function-specific key.
  + **admin** You need to provide the master key.
* **Methods** You can configure the HTTP methods that your function accepts. By default, the function runtime accepts all HTTP methods. [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13) reduces these accepted HTTP methods to GET and POST. Don’t use this parameter if you set the WebHookType parameter.
* **Route** You can customize the route of the endpoint used for the function to listen to a new request. The default route is https://<*your\_function\_app*>.azurewebsites.net/api/<*your\_function\_name*>.
* **WebHookType** This parameter is available only for version 1.x runtime functions. You should not use the Methods and WebHookType parameters together. This parameter sets the webhook type for a specific provider. Allowed values are
  + **genericJson** This parameter is used for nonspecific providers.
  + **github** This parameter is used for interacting with GitHub webhooks.
  + **slack** This parameter is used for interacting with Slack webhooks.

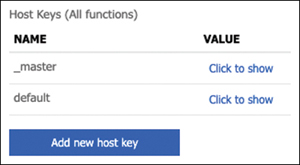
When you declare the variable type that your function uses as the input from the trigger, you can use HttpRequest or a custom type. If you use a custom type, the runtime tries to parse the request body as a JSON object for getting needed information for setting your custom type properties. If you decide to use HttpRequest for the type of the trigger input parameter, you get full access to the request object.

Every Azure Function App that you deploy automatically exposes a group of admin endpoints that you can use for accessing programmatically some aspects of your app, such as the status of the host. These endpoints look like

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg060a)

https://<*your\_function\_app\_name*>.azurewebsites.net/admin/host/status

By default, these endpoints are protected by an access code or authentication key that you can manage from your Function App in the Azure portal, as shown in [Figure 1-12](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig12).



**Figure 1-12** Managing host keys for a Function App

When you use the HTTP trigger, any endpoint that you publish is also protected by the same mechanism, although the keys that you use for protecting those endpoints are different. You can configure two types of authorization keys:

* **host** These keys are shared by all functions deployed in the Function App. This type of key allows access to any function in the host.
* **function** These keys only protect the function where they are defined.

When you define a new key, you assign a name to the key. If you have two keys of a different type—host and function—with the same name, the function key takes precedence. There are also two default keys—one per type of key—that you can also use for accessing your endpoints. These default keys take precedence over any other key that you created. If you need access to the admin endpoints that I mentioned earlier, you need to use a particular host key called \_master. You also need to use this administrative key when you set the admin value to the AuthLevel trigger configuration parameter. You can provide the appropriate key when you make a request to your API by using the code parameter or using the x-function-key HTTP header.

Protecting your endpoints using the authorization keys is not a recommended practice for production environments. You should only use authorization keys on testing or development environments for controlling the access to your API. For a production environment, you should use one of the following approaches:

* **Enable Function App Authorization/Authentication** This integrates your API with Azure Active Directory or other third-party identity providers to authenticate clients.
* **Use Azure API Management (APIM)** This secures the incoming request to your API, such as filtering by IP address or using authentication based on certificates.
* **Deploy your function in an App Service Environment (ASE)** ASEs provides dedicated hosting environments that allow you to configure a single front-end gateway that can authenticate all incoming requests.

If you decide to use any of the previous security methods, you need to ensure that you configure the AuthLevel as anonymous. You can see this configuration in [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13) in this line:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg061-1a)

HttpTrigger(AuthorizationLevel.Anonymous…

When you enable the App Service Authentication/Authorization, you can access the information about the authentication users by reading special HTTP headers set by the App Service. These special headers cannot be set by external resources; they can be set only by the App Service. For ASP.NET projects, the framework automatically fills a ClaimsPrincipal object with the authentication information. You can use ClaimsPrincipal as an additional parameter of your function signature or from the code—using the request context—as shown previously in [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13).

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg061-2a)

ClaimsPrincipal identities = req.HttpContext.User;

string username = identities.Identity?.Name;

As described in this section, Azure Functions runtime exposes your function by default using the following URL schema:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg062-1a)

https://<your\_function\_app\_name>.azurewebsites.net/api/<your\_function\_name>

You can customize the endpoint by using the route HTTPTrigger parameter. In [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13), you set the route parameter to devices/{id:int?}. This means that your endpoint looks like this:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg062-2a)

https://<*your\_function\_app\_name*>.azurewebsites.net/api/devices/{id:int?}

When you customize the route for your function, you can also add parameters to the route, which are accessible to your code by adding them as parameters of your function’s signature. You can use any Web API Route Constraint (see [*https://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2#constraints*](https://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2#constraints)) that you may use when defining a route using Web API 2.

By default, when you make a request to a function that uses an HTTP trigger, the response is an empty body with these status codes:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg062-3a)

HTTP 200 OK in case of Function 1.x runtime

HTTP 204 No Content in case of Function 2.x runtime

If you need to customize the response of your function, you need to configure an output binding. You can use any of the two types of output bindings, using the return statement or a function parameter. [Listing 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex13) shows how to configure the output binding for returning a JSON object with some information.

It is important to remember the limits associated with the function when you plan to deploy your function in a production environment. These limits are

* **Maximum request length** The HTTP request should not be larger than 100 MB.
* **Maximum URL length** Your custom URL is limited to 4096 bytes.
* **Execution timeout** Your function should return a value in less than 230 seconds. Your function can take more time to execute, but if it doesn’t return anything before that time, the gateway will time out with an HTTP 502 error. If your function needs to take more time to execute, you should use an async pattern and return a ping endpoint to allow the caller to ask for the status of the execution of your function.

***NEED MORE REVIEW?* HOST PROPERTIES**

You can also make some adjustments to the host where your function is running by using the host.json file. Visit the following article for reviewing all the properties available in the host.json file at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook#trigger---hostjson-properties*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook#trigger---hostjson-properties).

You can also review which are the limits associated with the different framework versions and hosting plans by reviewing the article at [*https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale*](https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale).

**Images *EXAM TIP***

In earlier versions of Azure Functions, the authentication information was available only to ASP.NET projects using the ClaimsPrincipal class. Now you can access that information by reading the special HTTP Headers set by the App Service. For a complete list of authentication headers refer to the article [*https://docs.microsoft.com/en-us/azure/app-service/app-service-authentication-how-to#access-user-claims*](https://docs.microsoft.com/en-us/azure/app-service/app-service-authentication-how-to#access-user-claims).

**Implement Azure Durable Functions**

One crucial characteristic of Azure functions is that they are stateless. This characteristic means that function runtime does not maintain the state of the objects that you create during the execution of the function if the host process or the VM where the function is running is recycled or rebooted.

Azure Durable Functions are an extension of the Azure Functions that provide stateful workflow capabilities in a serverless environment. These stateful workflow capabilities allow you to

* **Chain function calls together** This chaining means that a function can call other functions, which maintains the status between calls. These calls can be synchronous or asynchronous.
* **Define workflow by code** You don’t need to create JSON workflow definitions or use external tools.
* **Ensure that the status of the workflow is always consistent** When a function or activity on a workflow needs to wait for other functions or activities, the workflow engine automatically creates checkpoints for saving the status of the activity.

The main advantage of using Azure Durable Functions is that it eases the implementation of complex stateful coordination requirements in serverless scenarios. Although Durable Azure Functions is an extension of Azure Functions, at the time of this writing, it doesn’t support all languages supported by Azure Functions. The following languages are supported:

* **C#** Both precompiled class libraries and C# script are supported.
* **F#** Precompiled class libraries and F# script are supported. F# script is available only for Azure Functions runtime 1.x.
* **JavaScript** Supported only for Azure Functions runtime version 2.x runtime. Version 1.7.0 or later or Azure Durable Functions is required.

Durable Functions are billed using the same rules that apply to Azure Functions. That is, you are charged only for the time that your functions are running.

Working with Durable Functions means that you need to deal with different kinds of functions. Each type of function plays a different role in the execution of the workflow. These roles are

* **Activity** These are the functions that do the real work. An activity is a job that you need your workflow to do. For example, you may need your code to send a document to a content reviewer before other activity can publish the document, or you need to create a shipment order to send products to a client.
* **Orchestrator** Any workflow executes activity functions in a particular order. Orchestrator functions define the actions that a workflow executes. These actions can be activity functions, timers, or waiting for external events or suborchestrations. Each instance of an orchestrator function has an instance identifier. You can generate this identifier manually or leave the Durable Function framework to generate it dynamically.
* **Client** This is the entry point of a workflow. Triggers such as HTTP, queue, or event triggers create instances of a client function. Client functions create instances of orchestrator functions by sending an orchestrator trigger.

In the same way that Azure Functions uses triggers and bindings for sending and receiving information from functions, you need to use triggers and bindings for setting the communication between the different types of durable functions. Durable functions add two new triggers to control the execution of orchestration and activity functions:

* **Orchestration trigger** These allow you to work with orchestration functions by creating new instances of the function or resuming instances that are waiting for a task. The most important characteristic of these triggers is that they are single-threaded. When you use orchestration triggers, you need to ensure that your code does not perform async calls—other than waiting for durable function tasks—or I/O operations. This ensures that the orchestration function is focused on calling activity functions in the correct order and waiting for the correct events or functions.
* **Activity trigger** This is the type of trigger that you need to use when writing your activity functions. These triggers allow communications between orchestration functions and activity functions. They are multithreaded and don’t have any restrictions related to threading or I/O operations.

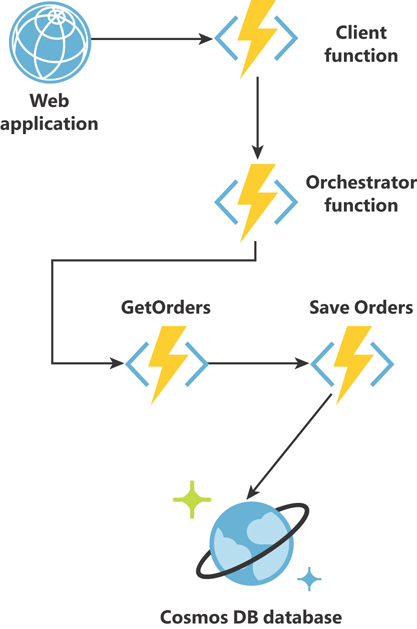
The following example shows how the different types of functions and triggers work together for processing and saving a hypothetical order generated from an external application and saved to a Cosmos DB database. Although the example is quite simple, it shows how the different functions interact. [Figure 1-13](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01fig13) shows a diagram of the workflow implemented on the functions shown in [Listings 1-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex14) to [1-21](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex21). For running this example, you need to meet the following requirements:

* An Azure subscription.
* An Azure Storage Account. The orchestration function needs an Azure Storage Account for saving the status of each durable function instance during the execution of the workflow.
* An Azure Cosmos DB database.
* Install the following dependencies using this command:

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg064a)

func extensions install -p <*package\_name*> -v <*package\_version*>

* Cosmos DB:
  + Package name: Microsoft.Azure.WebJobs.Extensions.CosmosDB
  + Version: 3.0.3
* Durable Functions extension:
  + Package name: Microsoft.Azure.WebJobs.Extensions.DurableTask
  + Version: 1.8.0



**Figure 1-13** Durable function workflow

You can run this example using your favorite Integrated Development Environment (IDE). Visual Studio and Visual Studio Code offer several tools that make working with Azure projects more comfortable. Use the following steps for configuring your Visual Studio Code and creating the durable functions:

1. Open your Visual Studio Code.
2. Click the Extensions icon on the left side of the window.
3. On the Extensions panel, on the Search Extensions In Marketplace text box, type **Azure Functions**.
4. In the result list, on the Azure Functions extension, click the Install button. Depending on your Visual Studio Code version, you may need to restart Visual Studio Code.
5. Click the Azure icon on the left side of the Visual Studio Code window.
6. In the Functions section, click Sign In To Azure For Log Into Azure.
7. In the Functions section, click the lightning bolt icon, which creates a new Azure Function.
8. In the Create New Project dialog box, select JavaScript.
9. In the Select A Template For Your Project’s First Function dialog box, select HTTP Trigger.
10. For the Provide A Function Name option, type **HTTPTriggerDurable**. This creates the first function that you need for this example.
11. Select Anonymous for the Authorization Level.
12. Select Open In Current Window to open the project that you just created.

Repeat steps 5 to 12 for all the Durable Functions that you need for this example. It is important to save all the functions you need in the same folder.

[Listings 1-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex14) and [1-15](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex15) show the JavaScript code and the JSON configuration file that you need to create the client function that calls the orchestration function.

**Listing 1-14** Azure Durable Functions client function code

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-14a)

// NodeJS. HTTPTriggerDurable/index.js

const df = require("durable-functions");

module.exports = async function (context, req) {

context.log('JavaScript Durable Functions example');

const client = df.getClient(context);

const instanceId = await client.startNew(req.params.functionName, undefined,

req.body);

context.log('Started orchestration with ID = '${instanceId}'.');

return client.createCheckStatusResponse(context.bindingData.req, instanceId);

};

**Listing 1-15** Durable Functions—Client function JSON configuration file

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-15a)

{

"disabled": false,

"bindings": [

{

"authLevel": "anonymous",

"type": "httpTrigger",

"direction": "in",

"name": "req",

"route": "orchestrators/{functionName}",

"methods": [

"get",

"post"

]

},

{

"type": "http",

"direction": "out",

"name": "$return"

},

{

"name": "context",

"type": "orchestrationClient",

"direction": "in"

}

]

}

[Listings 1-16](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex16) and [1-17](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex17) show the JavaScript code and the JSON configuration file that you need to create the Orchestration function that invokes, in the correct order, all the other activity functions. This function also returns to the client’s function the results of the execution of the different activity functions.

**Listing 1-16** Azure Durable Functions Orchestrator function code

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-16a)

// NodeJS. OrchestratorFunction/index.js

const df = require("durable-functions");

module.exports = df.orchestrator(function\*(context) {

context.log("Starting workflow: chain example");

const order = yield context.df.callActivity("GetOrder");

const savedOrder = yield context.df.callActivity("SaveOrder", order);

return savedOrder;

});

**Listing 1-17** Durable Functions—Orchestrator function JSON configuration file

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-17a)

{

"disabled": false,

"bindings": [

{

"type": "orchestrationTrigger",

"direction": "in",

"name": "context"

}

]

}

[Listings 1-18](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex18) and [1-19](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex19) show the JavaScript code and the JSON configuration file that you need to create the activity function Get Order. In this example, this function is in charge of constructing the information that is used in the Save Order function. In a more complex scenario, this function could get information from the user’s shopping cart from an e-commerce system or any other potential source.

**Listing 1-18** Azure Durable Functions activity function code

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-18a)

// NodeJS. GetOrder/index.js

module.exports = async function (context) {

//Create a mock order for testing

var order = {

"id" : Math.floor(Math.random() \* 1000),

"name" : "Customer",

"date" : new Date().toJSON()

}

context.log(order);

return order;

};

**Listing 1-19** Azure Durable Functions activity function JSON configuration file

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-19a)

{

"disabled": false,

"bindings": [

{

"type": "activityTrigger",

"direction": "in",

"name": "name"

}

]

}

[Listings 1-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex20) and [1-21](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex21) show the JavaScript code and the JSON configuration file that you need to create the activity function that saves the order in a Cosmos DB database. In a much more complex scenario, you could use this function to insert the order into your ERP system or send it to another activity function that could do further analysis or processing.

**Listing 1-20** Azure Durable Functions activity function code

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-20a)

// NodeJS. SaveOrder/index.js

module.exports = async function (context) {

//Saves the order object received from other activities to a CosmosDB document

context.bindings.orderDocument = JSON.stringify({

"id": '${context.bindings.order.id}',

"customerName": context.bindings.order.name,

"orderDate": context.bindings.order.date,

"cosmosDate": new Date().toJSON()

});

context.done();

};

**Listing 1-21** Azure Durable Functions activity function JSON configuration file

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#lis1-21a)

{

"disabled": false,

"bindings": [

{

"type": "activityTrigger",

"direction": "in",

"name": "order"

}

,

{

"name": "orderDocument",

"type": "cosmosDB",

"databaseName": "ERP\_Database",

"collectionName": "Orders",

"createIfNotExists": true,

"connectionStringSetting": "CosmosDBStorage",

"direction": "out"

}

]

}

The entry point in any workflow implemented using Durable Functions is always a client function. This function uses the orchestration client for calling the orchestrator function. [Listing 1-15](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex15) shows how to configure the output binding.

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg069a)

{

"name": "context",

"type": "orchestrationClient",

"direction": "in"

}

When you are using JavaScript for programming your client function, the orchestrator client output binding is not directly exposed using the value of the name attribute set in the function.json configuration file. In this case, you need to extract the actual client from the context variable using the getClient() function declared in the durable-functions package, as shown in [Listing 1-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex14).

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg069-2a)

const client = df.getClient(context);

Once you have the correct reference to the orchestrator client output binding, you can use the method startNew() for creating a new instance of the orchestrator function. The parameters for this method are

* **Name of the orchestrator function** In the example, you get this name from the HTTP request, using the URL parameter functionName, as previously shown in [Listings 1-14](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex14) and [1-15](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex15).
* **InstanceId** Sets the Id assigned to the new instance of the orchestration function. If you don’t provide a value to this parameter, then the method creates a random Id. In general, you should use the autogenerated random Id.
* **Input** This is where you place any data that your orchestration function may need. You need to use JSON-serializable data for this parameter.

Once you have created the instance of the orchestration function and saved the *Id* associated with the instance, the client function returns a data structure with several useful HTTP endpoints. You can use these endpoints to review the status of the execution of the workflow, or terminate the workflow, or send external events to the workflow during the execution. Following is an example of the workflow management endpoints for the execution of the example in a local computer environment:

***NOTE* CONSOLE OUTPUT**

For the shake of the space, some lines have been trimmed. Your output should show longer lines, including the instance id and other codes.

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg070-1a)

{

"id": "789e7eb945a04ab78e74e9216870af28",

"statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/

instances…",

"sendEventPostUri": "http://localhost:7071/runtime/webhooks/durabletask/

instances…",

"terminatePostUri": "http://localhost:7071/runtime/webhooks/durabletask/

instances…",

"rewindPostUri": "http://localhost:7071/runtime/webhooks/durabletask/

instances…",

"purgeHistoryDeleteUri": "http://localhost:7071/runtime/webhooks/durabletask/

instances."

}

This example uses an Azure Function based on an HTTP trigger, but your client function is not limited to use this trigger. You can use any of the triggers available in the Azure Function framework.

Once you have created the instance of the orchestrator function, this function calls the activity functions by the order defined in the code, as previously shown in [Listing 1-16](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex16).

[Click here to view code image](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01_images.xhtml#pg070-2a)

const order = yield context.df.callActivity("GetOrder");

const savedOrder = yield context.df.callActivity("SaveOrder", order);

The orchestrator function uses an orchestration trigger for getting the information that the client function sends when it creates the instance. The orchestration trigger creates the instances of the different activity functions by using the callActivity() method of the durable-functions package. This method takes two parameters:

* **Name of the activity function**
* **Input** You put here any JSON-serializable data that you want to send to the activity function.

In the example, you execute the activity function GetOrder, previously shown in [Listing 1-18](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex18), for getting the order object that you use as the input parameter for the next activity function SaveOrder, previously shown in [Listing 1-20](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex20), for saving the information in the Cosmos DB database configured in [Listing 1-21](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01ex21).

You can test this example on your local computer by running the functions that reviewed in this section, in the same way that you test any other Azure Function. Once you have your function running, you can test it by using curl or postman. You should make a GET or POST HTTP request to this URL: *http://localhost:7071/api/orchestrators/OrchestratorFunction*.

Notice that the parameter functionName of the URL matches with the name of the orchestrator function. Your client function allows you to call different orchestration functions just by providing the correct orchestration function name.

You can use different patterns when you are programming the orchestration function. These patterns show how the orchestration and activity functions interact with each other:

* **Chaining** The activity functions are executed in a specific order, where the output of one activity function is the input of the next one. This is the pattern that you used in your example.
* **Fan out/fan in** Your orchestration function executes multiple activity functions in parallel. The result of these parallel activity functions is processed and aggregated by a final aggregation activity function.
* **Async HTTP APIs** This pattern coordinates the state of long-running operations with external clients.
* **Monitor** This pattern allows you to create recurrent tasks using flexible time intervals.
* **Human Interaction** Use this pattern when you need to run activity functions based on events that a person can trigger. An example of this type of pattern is the document approval workflow, where publishing a document depends on the approval of a person.

***NEED MORE REVIEW?* DURABLE FUNCTION PATTERNS**

You can get more information about Durable Function patterns by reviewing the article Patterns and Concepts in Microsoft Docs at [*https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-concepts*](https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-concepts).

**Images *EXAM TIP***

When working with Azure Durable Functions, remember that you can pass information between the different functions in the workflow by using the binding mechanism.

**Chapter summary**

* Azure provides computing services for deploying your own virtualized infrastructure directly in the cloud. You can also deploy hybrid architectures to connect your on-premises infrastructure with your IaaS resources.
* Azure Resource Manager is the service in Azure that manages the different resources that you can deploy in the cloud. You can define the resources and their dependencies by using a JSON-based file called an ARM template.
* A container image is a package of software in which you store your code and any library or dependencies for running your application in a highly portable environment.
* When you create a new instance of a container image, each of these instances is named a “container.”
* You can store your container images in a centralized store called a registry.
* Azure Container Registry is a managed registry, which is based on the open-source specification of Docker Registry 2.0.
* You can run your containers in several Azure services, such as Azure Managed Kubernetes Service, Azure Container Instance, Azure Batch, Azure App Service, or Azure Container Service.
* Azure provides you with needed services for deploying serverless solutions, allowing you to center on the code and forget about the infrastructure.
* Azure App Services is the base of the serverless offering. On top of App Services, you can deploy web apps, mobile back-end apps, REST APIs, or Azure Functions and Azure Durable Functions.
* When you work with App Services, you are charged only by the time that your code is running.
* App Services runs on top of App Services Plans.
* An App Service Plan provides the resources and virtual machines needed for running your App Services code.
* You can run more than one App Service on top of a single App Service Plan.
* When troubleshooting your App Service application, you can use several types for diagnostics logging: webserver logging and diagnostics, detailed error, failed requests, application diagnostics, and deployment diagnostics.
* Diagnostics logging is stored locally on the VM, where the instance of your application is running.
* Horizontal scaling or in-out scaling is the process of adding or removing instances of an application.
* Vertical scaling or up-down scaling is the process of adding or removing resources to the same virtual machine that hosts your application.
* Scale In/Out doesn’t have an effect on the availability of the application.
* Vertical scaling affects the availability of the application because the application needs to be deployed in a virtual machine with the new resources assignment.
* You can add and remove resources to your applications by using autoscale rules.
* You can apply autoscale only to some Azure Resource types.
* Autoscale depends on Azure virtual machine scale sets.
* Your application needs to be aware of the changes in the resources assignment.
* Azure Functions is the evolution of WebJobs.
* Azure Functions use triggers and bindings for creating instances of Azure functions and sending or receiving data to or from external services, like Queue storage or Event Hub.
* There are three versions of Azure Functions. Version 1.0 only supports .NET Framework and Windows environments. Version 2.0 and later support .NET Core and Windows and Linux environments.
* When you work with triggers and bindings, you need to install the appropriate NuGet package for function extension that contains that trigger or binding.
* Azure Function runtime already includes extensions for Timers and HTTP triggers. You don’t need to install specific packages for using these trigger bindings.
* Triggers that create function instances can be based on data operations, timers, or webhooks.
* Azure Durable Functions is the evolution of Azure Functions that allow you to create workflows where the state of the instances is preserved in case of VM restart or function host process respawn.
* Orchestration functions define the activity and the order of execution of the functions that do the job.
* Activity functions contain the code that makes the action that you need for a step in the workflow, like sending an email, saving a document, or inserting information in a database.
* Client functions create the instance of the orchestration function using an orchestration client.
* Azure Function Apps provides the resources needed for running Azure Functions and Durable Functions.

**Thought experiment**

In this Thought Experiment, you can demonstrate your skills and knowledge about the topics covered in this chapter. You can find the answers to this Thought Experiment in the next section.

You are developing an application for making the integration between several systems. One of the systems is a legacy application that generates some reports in a specific file format. Those file reports are uploaded to an Azure Storage account. Your application reads the information from these file reports and inserts the information in different destination systems. Answer the following questions related to the described scenario:

[**1.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01que1a) Before your application can insert information on the destination, the information needs to be approved. This approval workflow needs to start when a new report file is added to the Azure Storage Account. Which Azure service best fits your needs?

[**2.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01que2a) Your application is suffering from performance issues. The performance issues only happen during some days in the month. You need to ensure that your application doesn’t suffer performance issues during the usage peaks. How can you achieve this?

**Thought experiment answers**

This section contains the solutions to the Thought Experiment.

[**1.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01que1) As the information needs to be approved before it can be inserted in the target systems, you should use Azure Durable Functions. By implementing a Human Interaction pattern, you can wait for the information to be validated before inserting it into the correct destination system. You can also use Azure Blob Storage triggers for starting the workflow. Because you need to wait for human confirmation, you should use Azure Durable Function instead of Azure Functions.

[**2.**](https://learning.oreilly.com/library/view/exam-ref-az-204/9780136798255/ch01.xhtml#ch01que2) You can deploy Azure Durable Functions to Azure App Service Plans. Starting with the Standard pricing tier, you can configure Autoscale rules for your Azure App Service Plan. Using autoscale rules, you can add or remove resources to the App Service Plan based on your needs. In this scenario, you can add more resources based on CPU consumption or during specific days. Because no specific pattern has been described in the scenario, you should first study the usage pattern before configure the appropriate autoscale rules.