Azure PowerShell Creating administration scripts is a powerful way to optimize your workflow. You can automate everyday, repetitive tasks, and once a script has been verified, it will run consistently, likely reducing errors. Azure PowerShell is ideal for one-off interactive tasks and/or the automation of repeated tasks. PowerShell is a cross-platform shell that provides services like the shell window and command parsing. Azure PowerShell is an optional add-on package that adds the Azure-specific commands (referred to as cmdlets). You can learn more about installing and using Azure PowerShell in a separate training module. For example, you can use the New-AzureRmVM cmdlet to create a new Azure virtual machine.

New-AzureRmVm -ResourceGroupName "TestResourceGroup" -Name "test-wp1-eus-vm" -Location "East US" -VirtualNetworkName "test-wp1-eus-network" ` -SubnetName "default" -SecurityGroupName "test-wp1-eus-nsg" -PublicIpAddressName "test-wp1-eus-pubip" -OpenPorts 80,3389

Azure CLI

Another option for scripting and command-line Azure interaction is the Azure CLI. The Azure CLI is Microsoft's cross-platform command-line tool for managing Azure resources such as virtual machines and disks from the command line. It's available for macOS, Linux, and Windows, or in the browser using the Cloud Shell. Like Azure PowerShell, the Azure CLI is a powerful way to streamline your administrative workflow. Unlike Azure PowerShell, the Azure CLI does not need PowerShell to function. For example, you can create an Azure VM with the az vm create command.

az vm create --resource-group TestResourceGroup --name test-wp1-eus-vm --image win2016datacenter --admin-username jonc --admin-password aReallyGoodPasswordHere

What is a fault domain?

A fault domain is a logical group of hardware in Azure that shares a common power source and network switch. You can think of it as a rack within an on-premises datacenter. The first two VMs in an availability

What is an update domain?

An update domain is a logical group of hardware that can undergo maintenance or be rebooted at the same time. Azure will automatically place availability sets into update domains to minimize the impact when the Azure platform introduces host operating system changes. Azure then processes each update domain one at a time. Availability sets are a powerful feature to ensure the services running in your VMs are always available to your customers. However, they aren't foolproof. What if something happens to the data or the software running on the VM itself? For that, we'll need to look at other disaster recovery and backup techniques.

Create virtual machine

Create a VM with New-AzureRmVM. Provide names for each of the resources and the New-AzureRmVM cmdlet creates if they don't already exist. When prompted, provide a username and password to be used as the logon credentials for the VM:

New-AzureRmVm -ResourceGroupName "myResourceGroup" -Name "myVM" -Location "East US" -VirtualNetworkName "myVnet" -SubnetName "mySubnet" -SecurityGroupName "myNetworkSecurityGroup" -PublicIpAddressName "myPublicIpAddress" -OpenPorts 80,3389

Configure Azure Disk Encryption for VMs

What is encryption? Encryption is about converting meaningful information into something that appears meaningless, such as a random sequence of letters and numbers. The process of encryption uses some form of key as part of the algorithm that creates the encrypted data. A key is also needed to perform the decryption. Keys may be symmetric, where the same key is used for encryption and decryption, or asymmetric, where different keys are used. An example of the latter is the public-private key pairs used in digital certificates.

Symmetric encryption

Algorithms that use symmetric keys, such as Advanced Encryption Standard (AES), are typically faster than public key algorithms, and are often used for protecting large data stores. Because there's only one key, procedures must be in place to prevent the key from becoming publicly known.

Asymmetric encryption

With asymmetric algorithms, only the private key member of the pair must be kept private and secure; as its name suggests, the public key can be made available to anyone without compromising the encrypted data. The downside of public key algorithms, however, is that they're much slower than symmetric algorithms, and cannot be used to encrypt large amounts of data.

Azure disk encryption technologies

The main encryption-based disk protection technologies for Azure VMs are:

● Storage Service Encryption (SSE)

● Azure Disk Encryption (ADE)

Review questions

Module 1 review questions

Azure VMs

What is an availability set and what are some of their benefits?

**> Click to see suggested answer**

An availability set is a logical feature used to ensure that a group of related VMs are deployed so that they aren't all subject to a single point of failure and not all upgraded at the same time during a host operating system upgrade in the datacenter. VMs placed in an availability set should perform an identical set of functionalities and have the same software installed. You can create availability sets through the Azure portal in the disaster recovery section. Also, you can build them using Resource Manager templates, or any of the scripting or API tools. When you place VMs into an availability set, Azure guarantees to spread them across Fault Domains and Update Domains.

**Azure Disk Encryption**

Can you name and describe the main encryption-based disk protection technologies?

> Click to see suggested answer

Storage Service Encryption: Azure Storage Service Encryption (SSE) is an encryption service built into Azure used to protect data at rest. The Azure storage platform automatically encrypts data before it's stored to several storage services, including Azure Managed Disks. Encryption is enabled by default using 256-bit AES encryption, and is managed by the storage account administrator. Azure Disk Encryption:

Azure Disk Encryption (ADE) is managed by the VM owner. It controls the encryption of Windows and Linux VM-controlled disks, using BitLocker on Windows VMs and DM-Crypt on Linux VMs. BitLocker Drive Encryption is a data protection feature that integrates with the operating system, and addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers. Similarly, DM-Crypt encrypts data at rest for Linux before writing to storage.

**Azure Batch overview**

**Introduction to Azure Batch**

Use Azure Batch to run large-scale parallel and high-performance computing (HPC) batch jobs efficiently in Azure. Azure Batch creates and manages a pool of compute nodes (virtual machines), installs the applications you want to run, and schedules jobs to run on the nodes. There is no cluster or job scheduler software to install, manage, or scale. Instead, you use Batch APIs and tools, command-line scripts, or the Azure portal to configure, manage, and monitor your jobs. Developers can use Batch as a platform service to build SaaS applications or client apps where large-scale execution is required. For example, build a service with Batch to run a Monte Carlo risk simulation for a financial services company, or a service to process many images. There is no additional charge for using Batch. You only pay for the underlying resources consumed, such as the virtual machines, storage, and networking.

Run a batch job by using Azure CLI and Azure portal

Running Batch jobs with Azure CLI

The Azure CLI is used to create and manage Azure resources from the command line or in scripts. This section of the course shows how to use the Azure CLI to create a Batch account, a pool of compute nodes (virtual machines), and a job that runs tasks on the pool. Each sample task runs a basic command on one of the pool nodes. After completing this, you will understand the key concepts of the Batch service and be ready to try Batch with more realistic workloads at larger scale. If you don't have an Azure subscription, create a free account before you begin.

Create a resource group Create a resource group with the az group create command. An Azure resource group is a logical container into which Azure resources are deployed and managed. The following example creates a resource group named myResourceGroup in the eastus2 location.

az group create --name myResourceGroup --location eastus2

Create a storage account

You can link an Azure Storage account with your Batch account. Although not required for this quickstart, the storage account is useful to deploy applications and store input and output data for most real-world workloads. Create a storage account in your resource group with the az storage account create command.

az storage account create --resource-group myResourceGroup --name mystorageaccount --location eastus2 --sku Standard\_LRS

Create a Batch account

Create a Batch account with the az batch account create command. You need an account to create compute resources (pools of compute nodes) and Batch jobs. The following example creates a Batch account named mybatchaccount in myResourceGroup, and links the storage account you created.

az batch account create --name mybatchaccount --storage-account mystorageaccount --resource-group myResourceGroup --location eastus2

Review questions Module 2 review questions

Azure Batch service resources

Most Batch solutions use Azure Storage for storing resource files and output files. For example, your Batch tasks (including standard tasks, start tasks, job preparation tasks, and job release tasks) typically specify resource files that reside in a storage account. What types of Azure Storage accounts will Azure Batch support?

> Click to see suggested answer Batch supports the following Azure Storage account options:

● General-purpose v2 (GPv2) accounts

● General-purpose v1 (GPv1) accounts

● Blob storage accounts (currently supported for pools in the Virtual Machine configuration)

You can associate a storage account with your Batch account when you create the Batch account, or later. Consider your cost and performance requirements when choosing a storage account. For example, the GPv2 and blob storage account options support greater capacity and scalability limits compared with GPv1. These account options can improve the performance of Batch solutions that contain a large number of parallel tasks that read from or write to the storage account.

Coding for Batch accounts

Azure subscriptions and the individual Azure services like Batch all have default quotas that limit the number of certain entities within them. You want to create a new Batch account in a region and you want to check your Azure subscription to see whether you are able to add an account in that region. How would you code for that?

> Click to see suggested answer

In the code snippet below, we first use BatchManagementClient.Account.ListAsync to get a collection of all Batch accounts that are within a subscription. Once we've obtained this collection, we determine how many accounts are in the target region. Then we use BatchManagementClient.Subscriptions to obtain the Batch account quota and determine how many accounts (if any) can be created in that region.

// Get a collection of all Batch accounts within the subscription BatchAccountListResponse listResponse = await batchManagementClient.Account.ListAsync(new AccountListParameters()); IList<AccountResource> accounts = listResponse.Accounts; Console.WriteLine("Total number of Batch accounts under subscription id {0}: {1}", creds.SubscriptionId, accounts.Count);

// Get a count of all accounts within the target region string region = "westus"; int accountsInRegion = accounts.Count(o => o.Location == region);

// Get the account quota for the specified region SubscriptionQuotasGetResponse quotaResponse = await batchManagementClient. Subscriptions.GetSubscriptionQuotasAsync(region); Console.WriteLine("Account quota for {0} region: {1}", region, quotaResponse.AccountQuota);

// Determine how many accounts can be created in the target region Console.WriteLine("Accounts in {0}: {1}", region, accountsInRegion); Console.WriteLine("You can create {0} accounts in the {1} region.", quotaResponse.AccountQuota - a

**Create containerized solutions**

What is Kubernetes? Kubernetes is a rapidly evolving platform that manages container-based applications and their associated networking and storage components. The focus is on the application workloads, not the underlying infrastructure components. Kubernetes provides a declarative approach to deployments, backed by a robust set of APIs for management operations. You can build and run modern, portable, microservices-based applications that benefit from Kubernetes orchestrating and managing the availability of those application components. Kubernetes supports both stateless and stateful applications as teams progress through the adoption of microservices-based applications. As an open platform, Kubernetes allows you to build your applications with your preferred programming language, OS, libraries, or messaging bus. Existing continuous integration and continuous delivery (CI/ CD) tools can integrate with Kubernetes to schedule and deploy releases. Azure Kubernetes Service (AKS) provides a managed Kubernetes service that reduces the complexity for deployment and core management tasks, including coordinating upgrades. The AKS cluster masters are managed by the Azure platform, and you only pay for the AKS nodes that run your applications. AKS is built on top of the open-source Azure Container Service Engine (acs-engine).

Kubernetes cluster architecture

A Kubernetes cluster is divided into two components:

● Cluster master nodes provide the core Kubernetes services and orchestration of application workloads.

● Nodes run your application workloads.

Cluster master

When you create an AKS cluster, a cluster master is automatically created and configured. This cluster master is provided as a managed Azure resource abstracted from the user. There is no cost for the cluster master, only the nodes that are part of the AKS cluster. The cluster master includes the following core Kubernetes components:

● kube-apiserver - The API server is how the underlying Kubernetes APIs are exposed. This component provides the interaction for management tools, such as kubectl or the Kubernetes dashboard.

● etcd - To maintain the state of your Kubernetes cluster and configuration, the highly available etcd is a key value store within Kubernetes.

● kube-scheduler - When you create or scale applications, the Scheduler determines what nodes can run the workload and starts them.

● kube-controller-manager - The Controller Manager oversees a number of smaller Controllers that perform actions such as replicating pods and handling node operations.

AKS provides a single-tenant cluster master, with a dedicated API server, Scheduler, etc. You define the number and size of the nodes, and the Azure platform configures the secure communication between the cluster master and nodes. Interaction with the cluster master occurs through Kubernetes APIs, such as kubectl or the Kubernetes dashboard.

This managed cluster master means that you do not need to configure components like a highly available etcd store, but it also means that you cannot access the cluster master directly. Upgrades to Kubernetes are orchestrated through the Azure CLI or Azure portal, which upgrades the cluster master and then the nodes. To troubleshoot possible issues, you can review the cluster master logs through Azure Log Analytics.

If you need to configure the cluster master in a particular way or need direct access to them, you can deploy your own Kubernetes cluster using aks-engine. Nodes and node pools To run your applications and supporting services, you need a Kubernetes node. An AKS cluster has one or more nodes, which is an Azure virtual machine (VM) that runs the Kubernetes node components and container runtime:

● The kubelet is the Kubernetes agent that processes the orchestration requests from the cluster master and scheduling of running the requested containers.

● Virtual networking is handled by the kube-proxy on each node. The proxy routes network traffic and manages IP addressing for services and pods.

Nodes and node pools

To run your applications and supporting services, you need a Kubernetes node. An AKS cluster has one or more nodes, which is an Azure virtual machine (VM) that runs the Kubernetes node components and container runtime:

● The kubelet is the Kubernetes agent that processes the orchestration requests from the cluster master and scheduling of running the requested containers.

● Virtual networking is handled by the kube-proxy on each node. The proxy routes network traffic and manages IP addressing for services and pods.

● The container runtime is the component that allows containerized applications to run and interact with additional resources such as the virtual network and storage. In AKS, Docker is used as the container runtime.

Node pools

Nodes of the same configuration are grouped together into node pools. A Kubernetes cluster contains one or more node pools. The initial number of nodes and size are defined when you create an AKS cluster, which creates a default node pool. This default node pool in AKS contains the underlying VMs that run your agent nodes. When you scale or upgrade an AKS cluster, the action is performed against the default node pool. For upgrade operations, running containers are scheduled on other nodes in the node pool until all the nodes are successfully upgraded.

Pods

Kubernetes uses pods to run an instance of your application. A pod represents a single instance of your application. Pods typically have a 1:1 mapping with a container, although there are advanced scenarios where a pod may contain multiple containers. These multi-container pods are scheduled together on the same node, and allow containers to share related resources.

When you create a pod, you can define resource limits to request a certain amount of CPU or memory resources. The Kubernetes Scheduler tries to schedule the pods to run on a node with available resources to meet the request. You can also specify maximum resource limits that prevent a given pod from consuming too much compute resource from the underlying node. A best practice is to include resource limits for all pods to help the Kubernetes Scheduler understand what resources are needed and permitted.

A pod is a logical resource, but the container(s) are where the application workloads run. Pods are typically ephemeral, disposable resources, and individually scheduled pods miss some of the high availability and redundancy features Kubernetes provides. Instead, pods are usually deployed and managed by Kubernetes Controllers, such as the Deployment Controller.

Deployments and YAML manifests

A deployment represents one or more identical pods, managed by the Kubernetes Deployment Controller. A deployment defines the number of replicas (pods) to create, and the Kubernetes Scheduler ensures that if pods or nodes encounter problems, additional pods are scheduled on healthy nodes.

You can update deployments to change the configuration of pods, container image used, or attached storage. The Deployment Controller drains and terminates a given number of replicas, creates replicas from the new deployment definition, and continues the process until all replicas in the deployment are updated.

Most stateless applications in AKS should use the deployment model rather than scheduling individual pods. Kubernetes can monitor the health and status of deployments to ensure that the required number of replicas run within the cluster. When you only schedule individual pods, the pods are not restarted if they encounter a problem, and are not rescheduled on healthy nodes if their current node encounters a problem.

If an application requires a quorum of instances to always be available for management decisions to be made, you don't want an update process to disrupt that ability. Pod Disruption Budgets can be used to define how many replicas in a deployment can be taken down during an update or node upgrade. For example, if you have 5 replicas in your deployment, you can define a pod disruption of 4 to only permit one replica from being deleted/rescheduled at a time. As with pod resource limits, a best practice is to define pod disruption budgets on applications that require a minimum number of replicas to always be present.

Package management with Helm

A common approach to managing applications in Kubernetes is with Helm. You can build and use existing public Helm charts that contain a packaged version of application code and Kubernetes YAML manifests to deploy resources. These Helm charts can be stored locally, or often in a remote repository, such as an Azure Container Registry Helm chart repo.

To use Helm, a server component called Tiller is installed in your Kubernetes cluster. The Tiller manages the installation of charts within the cluster. The Helm client itself is installed locally on your computer, or can be used within the Azure Cloud Shell. You can search for or create Helm charts with the client, and then install them to your Kubernetes cluster

AKS security concepts for apps and clusters

To protect your customer data as you run application workloads in Azure Kubernetes Service (AKS), the security of your cluster is a key consideration. Kubernetes includes security components such as network policies and Secrets. Azure then adds in components such as network security groups and orchestrated cluster upgrades. These security components are combined to keep your AKS cluster running the latest OS security updates and Kubernetes releases, and with secure pod traffic and access to sensitive credentials. This section introduces the core concepts that secure your applications in AKS:

● Master components security

● Node security

● Cluster upgrades

● Network security

● Kubernetes Secrets

Scaling options for apps in AKS

As you run applications in Azure Kubernetes Service (AKS), you may need to increase or decrease the amount of compute resources. As the number of application instances you need change, the number of underlying Kubernetes nodes may also need to change. You may also need to quickly provision a large number of additional application instances. This section introduces the core concepts that help you scale applications in AKS:

● Manually scale

● Horizontal pod autoscaler (HPA)

● Cluster autoscaler

● Azure Container Instance (ACI) integration with AKS

Manually scale pods or nodes

You can manually scale replicas (pods) and nodes to test how your application responds to a change in available resources and state. Manually scaling resources also lets you define a set amount of resources to use to maintain a fixed cost, such as the number of nodes. To manually scale, you define the replica or node count, and the Kubernetes API schedules creating additional pods or draining nodes.

Horizontal pod autoscaler

Kubernetes uses the horizontal pod autoscaler (HPA) to monitor the resource demand and automatically scale the number of replicas. By default, the horizontal pod autoscaler checks the Metrics API every 30 seconds for any required changes in replica count. When changes are required, the number of replicas is increased or decreased accordingly. Horizontal pod autoscaler works with AKS clusters that have deployed the Metrics Server for Kubernetes 1.8+.

Create AKS cluster

Use the az aks create command to create an AKS cluster. The following example creates a cluster named myAKSCluster with one node. Container health monitoring is also enabled using the –enable-addons monitoring parameter.

az aks create --resource-group myAKSCluster --name myAKSCluster --nodecount 1 --enable-addons monitoring --generate-ssh-keys

After several minutes, the command completes and returns JSON-formatted information about the cluster

Connect to the cluster

To manage a Kubernetes cluster, use kubectl, the Kubernetes command-line client.

If you're using Azure Cloud Shell, kubectl is already installed. If you want to install it locally, use the az aks install-cli command.

To configure kubectl to connect to your Kubernetes cluster, use the az aks get-credentials command. This step downloads credentials and configures the Kubernetes CLI to use them.

*az aks get-credentials --resource-group myAKSCluster --name myAKSCluster*

To verify the connection to your cluster, use the kubectl get command to return a list of the cluster nodes. It can take a few minutes for the nodes to appear.

*kubectl get nodes*

*/\*Creating an AKS cluster on Azure*

*//Create a resource group*

*az group create --name myResourceGroup --location eastus*

*//Create an AKS cluster*

*az aks create --resource-group myResourceGroup --name myAKSCluster --node-count 2 --enable-addons monitoring --generate-ssh-keys*

*//to install kubectl*

*az aks install-cli*

*//Configure and connect with kubernetes cluster*

*az aks get-credentials --resource-group myResourceGroup --name myAKSCluster*

*//Get nodes*

*kubectl get nodes*

*//Deploy and run the application, download and clone the file from github*

*kubectl apply -f azure-vote.yaml (https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough)*

*//test the application*

*kubectl get service azure-vote-front --watch*

*//once it gives external IP address then launch it with that.*

*//finally we want to see the AKS dashboard*

*az aks browse --resource-group myResourceGroup --name myAKSCluster*

*try opening the site with 127.0.0.1:8001*

*//if it throws error then run below command*

*kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --serviceaccount=kube-system:kubernetes-dashboard*

*//get pods*

*kubectl get pods*

Deploy an AKS cluster using Azure Portal

Now, we'll cover deploying an AKS cluster using the Azure portal. The first step is to sign in to the Azure portal: <https://portal.azure.com>.

Create an AKS cluster

In the top left-hand corner of the Azure portal, select Create a resource > Kubernetes Service.

To create an AKS cluster, complete the following steps: 1. Basics - Configure the following options:

● PROJECT DETAILS: Select an Azure subscription, then select or create an Azure resource group, such as myResourceGroup. Enter a Kubernetes cluster name, such as myAKSCluster.

● CLUSTER DETAILS: Select a region, Kubernetes version, and DNS name prefix for the AKS cluster.

● SCALE: Select a VM size for the AKS nodes. The VM size cannot be changed once an AKS cluster has been deployed.

● Select the number of nodes to deploy into the cluster. For this tutorial, set Node count to 1. Node count can be adjusted after the cluster has been deployed.

2. Select Next: Authentication when complete.

3. Authentication: Configure the following options: ● Create a new service principal or Configure to use an existing one. When using an existing SPN, you need to provide the SPN client ID and secret. ● Enable the option for Kubernetes role-based access controls (RBAC). These controls provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

4. Select Next: Networking when complete.

5. Networking: Configure the following networking options, which should be set as default: ● Http application routing - Select Yes to configure an integrated ingress controller with automatic public DNS name creation. ● Network configuration - Select the Basic network configuration using the kubenet Kubernetes plugin, rather than advanced networking configuration using Azure CNI.

6. Select Next: Monitoring when complete.

7. When deploying an AKS cluster, Azure Container Insights can be configured to monitor health of the AKS cluster and pods running on the cluster.

● Select Yes to enable container monitoring and select an existing Log Analytics workspace, or create a new one.

● Select Review + create and then Create when ready. It takes a few minutes to create the AKS cluster and to be ready for use. Browse to the AKS cluster resource group, such as myResourceGroup, and select the AKS resource, such as myAKSCluster.

Publish a container image to Azure Container Registry

Azure Container Registry overview

Azure Container Registry is a managed Docker registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your private Docker container images. Use container registries in Azure with your existing container development and deployment pipelines. Use Azure Container Registry Build (ACR Build) to build container images in Azure. Build on demand, or fully automate builds with source code commit and base image update build triggers.

Use cases Pull images from an Azure container registry to various deployment targets:

● Scalable orchestration systems that manage containerized applications across clusters of hosts, including Kubernetes, DC/OS, and Docker Swarm.

● Azure services that support building and running applications at scale, including Azure Kubernetes Service (AKS), App Service, Batch, Service Fabric, and others.

Developers can also push to a container registry as part of a container development workflow. For example, target a container registry from a continuous integration and deployment tool such as Azure DevOps Services or Jenkins.

Configure ACR Tasks to automatically rebuild application images when their base images are updated. Use ACR Tasks to automate image builds when your team commits code to a Git repository.

**Azure Container Registry Tasks**

Azure Container Registry Tasks (ACR Tasks) is a suite of features within Azure Container Registry that provides streamlined and efficient Docker container image builds in Azure. Use ACR Tasks to extend your development inner-loop to the cloud by offloading docker build operations to Azure. Configure build tasks to automate your container OS and framework patching pipeline, and build images automatically when your team commits code to source control.

Multi-step tasks, a preview feature of ACR Tasks, provides step-based task definition and execution for building, testing, and patching container images in the cloud. Task steps define individual container image build and push operations. They can also define the execution of one or more containers, with each step using the container as its execution environment.

**Review questions**

**Module 3 review questions**

**Kubernetes cluster architecture**

Azure Kubernetes Service (AKS) provides a managed Kubernetes service that reduces the complexity for deployment and core management tasks, including coordinating upgrades. The AKS cluster masters are managed by the Azure platform. What are the two components that make up a Kubernetes cluster?

> Click to see suggested answer A Kubernetes cluster is divided into two components:

● Cluster master nodes provide the core Kubernetes services and orchestration of application workloads. When you create an AKS cluster, a cluster master is automatically created and configured. This cluster master is provided as a managed Azure resource abstracted from the user.

● Nodes run your application workloads. To run your applications and supporting services, you need a Kubernetes node. An AKS cluster has one or more nodes, which is an Azure virtual machine (VM) that runs the Kubernetes node components and container runtime.

Azure Container Registry

How would you use the Azure Container Registry in your workflow?

> Click to see suggested answer Azure Container Registry is a managed Docker registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your private Docker container images.

Use container registries in Azure with your existing container development and deployment pipelines. Use Azure Container Registry Build (ACR Build) to build container images in Azure. Build on demand, or fully automate builds with source code commit and base image update build triggers.

Developers can also push to a container registry as part of a container development workflow. For example, target a container registry from a continuous integration and deployment tool such as Azure DevOps Services or Jenkins.

Session2



APIM Documents

Review questions

Module 1 review questions

Azure App Service plans

In App Service, an app runs in an App Service plan. An App Service plan defines a set of compute resources for a web app to run. These compute resources are analogous to the server farm in conventional web hosting. One or more apps can be configured to run on the same computing resources (or in the same App Service plan). Which App Service plan is available only to function apps?

> Click to see suggested answer

There are a few categories of pricing tiers:

● Consumption: This tier is only available to function apps. It scales the functions dynamically depending on workload.

● Shared compute: Free and Shared, the two base tiers, runs an app on the same Azure VM as other App Service apps, including apps of other customers. These tiers allocate CPU quotas to each app that runs on the shared resources, and the resources cannot scale out.

● Dedicated compute: The Basic, Standard, Premium, and PremiumV2 tiers run apps on dedicated Azure VMs. Only apps in the same App Service plan share the same compute resources. The higher the tier, the more VM instances are available to you for scale-out.

● Isolated: This tier runs dedicated Azure VMs on dedicated Azure Virtual Networks, which provides network isolation on top of compute isolation to your apps. It provides the maximum scale-out capabilities.

Managing web client requests Web clients are sending requests to your app configured with multiple endpoints. What would you use to control how those requests are routed, and what are the available routing methods?

> Click to see suggested answer

You can use Azure Traffic Manager to control how requests from web clients are distributed to apps in Azure App Service. When App Service endpoints are added to an Azure Traffic Manager profile, Azure Traffic Manager keeps track of the status of your App Service apps (running, stopped, or deleted) so that it can decide which of those endpoints should receive traffic. Routing methods Azure Traffic Manager uses four different routing methods. These methods are described in the following list as they pertain to Azure App Service.

● Priority: use a primary app for all traffic, and provide backups in case the primary or the backup apps are unavailable.

● Weighted: distribute traffic across a set of apps, either evenly or according to weights, which you define.

● Performance: when you have apps in different geographic locations, use the “closest” app in terms of the lowest network latency.

● Geographic: direct users to specific apps based on which geographic location their DNS query originates from

Create Azure App Service mobile apps

Review questions

Module 2 review questions

App Service push notifications Push notifications are delivered through platform-specific infrastructures called Platform Notification Systems (PNSes). They offer barebone push functionalities to deliver a message to a device with a provided handle, and have no common interface.

Can you describe how a push works?

> Click to see suggested answer At a high level, here is how push works:

1. The client app decides it wants to receive notification. Hence, it contacts the corresponding PNS to retrieve its unique and temporary push handle. The handle type depends on the system (for example, WNS has URIs while APNS has tokens).

2. The client app stores this handle in the app back-end or provider.

3. To send a push notification, the app back-end contacts the PNS using the handle to target a specific client app.

4. The PNS forwards the notification to the device specified by the handle.

5. Offline sync What is the general pattern you need to follow to enable offline sync regardless of the app platform?

> Click to see suggested answer

The steps for enabling offline sync follow a similar pattern for whichever platform you are targeting.

1. Ensure the client app supports offline features

2. Ensure the app can disconnect from the backend

3. Ensure the app can reconnect to your Mobile App backend

Create Azure App Service API apps

Creating APIs

API Management overview

API Management (APIM) helps organizations publish APIs to external, partner, and internal developers to unlock the potential of their data and services. Businesses everywhere are looking to extend their operations as a digital platform, creating new channels, finding new customers and driving deeper engagement with existing ones. API Management provides the core competencies to ensure a successful API program through developer engagement, business insights, analytics, security, and protection. You can use Azure API Management to take any backend and launch a full-fledged API program based on it.

Overview

To use API Management, administrators create APIs. Each API consists of one or more operations, and each API can be added to one or more products. To use an API, developers subscribe to a product that contains that API, and then they can call the API's operation, subject to any usage policies that may be in effect. Common scenarios include:

● Securing mobile infrastructure by gating access with API keys, preventing DOS attacks by using throttling, or using advanced security policies like JWT token validation.

● Enabling ISV partner ecosystems by offering fast partner onboarding through the developer portal and building an API facade to decouple from internal implementations that are not ripe for partner consumption.

● Running an internal API program by offering a centralized location for the organization to communicate about the availability and latest changes to APIs, gating access based on organizational accounts, all based on a secured channel between the API gateway and the backend. The system is made up of the following components:

● The API gateway is the endpoint that:

● Accepts API calls and routes them to your backends.

● Verifies API keys, JWT tokens, certificates, and other credentials.

● Enforces usage quotas and rate limits.

● Transforms your API on the fly without code modifications.

● Caches backend responses where set up.

● Logs call metadata for analytics purposes.

● The Azure portal is the administrative interface where you set up your API program. Use it to:

● Define or import API schema.

● Package APIs into products.

● Set up policies like quotas or transformations on the APIs.

● Get insights from analytics.

● Manage users.

● The Developer portal serves as the main web presence for developers, where they can:

● Read API documentation.

● Try out an API via the interactive console.

● Create an account and subscribe to get API keys.

● Access analytics on their own usage.

APIs and operations

APIs are the foundation of an API Management service instance. Each API represents a set of operations available to developers. Each API contains a reference to the back-end service that implements the API, and its operations map to the operations implemented by the back-end service. Operations in API Management are highly configurable, with control over URL mapping, query and path parameters, request and response content, and operation response caching. Rate limit, quotas, and IP restriction policies can also be implemented at the API or individual operation level.

Products

Products are how APIs are surfaced to developers. Products in API Management have one or more APIs, and are configured with a title, description, and terms of use. Products can be Open or Protected. Protected products must be subscribed to before they can be used, while open products can be used without a subscription. When a product is ready for use by developers, it can be published. Once it is published, it can be viewed (and in the case of protected products subscribed to) by developers. Subscription approval is configured at the product level and can either require administrator approval, or be auto-approved.

Groups are used to manage the visibility of products to developers. Products grant visibility to groups, and developers can view and subscribe to the products that are visible to the groups in which they belong.

Groups

Groups are used to manage the visibility of products to developers. API Management has the following immutable system groups:

● Administrators - Azure subscription administrators are members of this group. Administrators manage API Management service instances, creating the APIs, operations, and products that are used by developers.

● Developers - Authenticated developer portal users fall into this group. Developers are the customers that build applications using your APIs. Developers are granted access to the developer portal and build applications that call the operations of an API.

● Guests - Unauthenticated developer portal users, such as prospective customers visiting the developer portal of an API Management instance fall into this group. They can be granted certain read-only access, such as the ability to view APIs but not call them. In addition to these system groups, administrators can create custom groups or leverage external groups in associated Azure Active Directory tenants. Custom and external groups can be used alongside system groups in giving developers visibility and access to API products. For example, you could create one custom group for developers affiliated with a specific partner organization and allow them access to the APIs from a product containing relevant APIs only. A user can be a member of more than one group.

Developers

Developers represent the user accounts in an API Management service instance. Developers can be created or invited to join by administrators, or they can sign up from the Developer portal. Each developer is a member of one or more groups, and can subscribe to the products that grant visibility to those groups. When developers subscribe to a product, they are granted the primary and secondary key for the product. This key is used when making calls into the product's APIs.

Policies

Policies are a powerful capability of API Management that allow the Azure portal to change the behavior of the API through configuration. Policies are a collection of statements that are executed sequentially on the request or response of an API. Popular statements include format conversion from XML to JSON and call rate limiting to restrict the number of incoming calls from a developer, and many other policies are available. Policy expressions can be used as attribute values or text values in any of the API Management policies, unless the policy specifies otherwise. Some policies such as the Control flow and Set variable policies are based on policy expressions. For more information, see Advanced policies and Policy expressions.

Developer portal

The developer portal is where developers can learn about your APIs, view and call operations, and subscribe to products. Prospective customers can visit the developer portal, view APIs and operations, and sign up. The URL for your developer portal is located on the dashboard in the Azure portal for your API Management service instance.

API Management terminology

● Backend API - An HTTP service that implements your API and its operations.

● Frontend API/APIM API - An APIM API does not host APIs, it creates facades for your APIs in order to customize the facade according to your needs without touching the back end API.

● APIM product - a product contains one or more APIs as well as a usage quota and the terms of use. You can include a number of APIs and offer them to developers through the Developer portal.

● APIM API operation - Each APIM API represents a set of operations available to developers. Each APIM API contains a reference to the back end service that implements the API, and its operations map to the operations implemented by the back end service.

● Version - Sometimes you want to publish new or different API features to some users, while others want to stick with the API that currently works for them.

● Revision - When your API is ready to go and starts to be used by developers, you usually need to take care in making changes to that API and at the same time not to disrupt callers of your API. It's also useful to let developers know about the changes you made.

● Developer portal - Your customers (developers) should use the Developer portal to access your APIs. The Developer portal can be customized.

Create an Azure API Management service instance

Azure API Management (APIM) helps organizations publish APIs to external, partner, and internal developers to unlock the potential of their data and services. API Management provides the core competencies to ensure a successful API program through developer engagement, business insights, analytics, security, and protection. APIM enables you to create and manage modern API gateways for existing backend services hosted anywhere.

Using Swagger to document an API Getting started with Swashbuckle

This section of the course focuses on Swashbuckle to generate Swagger objects in ASP.NET Core. There are three main components to Swashbuckle:

● Swashbuckle.AspNetCore.Swagger: a Swagger object model and middleware to expose SwaggerDocument objects as JSON endpoints.

● Swashbuckle.AspNetCore.SwaggerGen: a Swagger generator that builds SwaggerDocument objects directly from your routes, controllers, and models. It's typically combined with the Swagger endpoint middleware to automatically expose Swagger JSON.

● Swashbuckle.AspNetCore.SwaggerUI: an embedded version of the Swagger UI tool. It interprets Swagger JSON to build a rich, customizable experience for describing the Web API functionality. It includes built-in test harnesses for the public methods. Package installation Here's how to install the Swashbuckle.AspNetCore package in Visual studio:

● From the Package Manager Console window:

● Go to View > Other Windows > Package Manager Console

● Navigate to the directory in which the TodoApi.csproj file exists

● Execute the following command: Install-Package Swashbuckle.AspNetCore

● From the Manage NuGet Packages dialog:

● Right-click the project in Solution Explorer > Manage NuGet Packages

● Set the Package source to “nuget.org”

● Enter “Swashbuckle.AspNetCore” in the search box

● Select the “Swashbuckle.AspNetCore” package from the Browse tab and click Install

Review questions

Module 3 review questions

API Management groups

Groups are used to manage the visibility of products to developers. Products grant visibility to groups, and developers can view and subscribe to the products that are visible to the groups in which they belong. Can you name the three immutable system groups and their capabilities?

> Click to see suggested answer API Management has the following immutable system groups:

● Administrators - Azure subscription administrators are members of this group. Administrators manage API Management service instances, creating the APIs, operations, and products that are used by developers.

● Developers - Authenticated developer portal users fall into this group. Developers are the customers that build applications using your APIs. Developers are granted access to the developer portal and build applications that call the operations of an API.

● Guests - Unauthenticated developer portal users, such as prospective customers visiting the developer portal of an API Management instance fall into this group. They can be granted certain read-only access, such as the ability to view APIs but not call them.

API Debug information

What do you need to enable in order to provide debug information for undocumented public types and members?

> Click to see suggested answer Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message. For example, the following message indicates a violation of warning code 1591:

warning CS1591: Missing XML comment for publicly visible type or member 'TodoController.GetAll()'

Implement Azure functions

Azure Functions overview

Introduction to Azure Functions

Azure Functions is a solution for easily running small pieces of code, or “functions,” in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development language of choice, such as C#, F#, Node.js, Java, or PHP. Pay only for the time your code runs and trust Azure to scale as needed. Azure Functions lets you develop serverless applications on Microsoft Azure.

What can I do with Functions?

Functions is a great solution for processing data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and microservices. Consider Functions for tasks like image or order processing, file maintenance, or for any tasks that you want to run on a schedule.

Functions provides templates to get you started with key scenarios, including the following:

● HTTPTrigger - Trigger the execution of your code by using an HTTP request.

● TimerTrigger - Execute cleanup or other batch tasks on a predefined schedule.

● GitHub webhook - Respond to events that occur in your GitHub repositories. Generic webhook - Process webhook HTTP requests from any service that supports webhooks.

● CosmosDBTrigger - Process Azure Cosmos DB documents when they are added or updated in collections in a NoSQL database.

● BlobTrigger - Process Azure Storage blobs when they are added to containers. You might use this function for image resizing.

● QueueTrigger - Respond to messages as they arrive in an Azure Storage queue.

● EventHubTrigger - Respond to events delivered to an Azure Event Hub. Particularly useful in application instrumentation, user experience or workflow processing, and Internet of Things (IoT) scenarios.

● ServiceBusQueueTrigger - Connect your code to other Azure services or on-premises services by listening to message queues.

● ServiceBusTopicTrigger - Connect your code to other Azure services or on-premises services by subscribing to topics. Azure Functions supports triggers, which are ways to start execution of your code, and bindings, which are ways to simplify coding for input and output data.

Integrations

Azure Functions integrates with various Azure and 3rd-party services. These services can trigger your function and start execution, or they can serve as input and output for your code. The following service integrations are supported by Azure Functions:

● Azure Cosmos DB

● Azure Event Hubs

● Azure Event Grid

● Azure Notification Hubs

● Azure Service Bus (queues and topics)

● Azure Storage (blob, queues, and tables)

● On-premises (using Service Bus)

● Twilio (SMS messages)

How much does Functions cost? Azure Functions has two kinds of pricing plans. Choose the one that best fits your needs:

● Consumption plan - When your function runs, Azure provides all of the necessary computational resources. You don't have to worry about resource management, and you only pay for the time that your code runs.

● App Service plan - Run your functions just like your web apps. When you are already using App Service for your other applications, you can run your functions on the same plan at no additional cost. For more information about hosting plans, see Azure Functions hosting plan comparison1.

Review questions

Module 4 review questions

Azure Functions consumption plans

Azure Functions run within a service plan. What service plans support Azure Functions and how to they differ?

> Click to see suggested answer Azure Functions runs in two different modes: Consumption plan and Azure App Service plan. When you're using a Consumption plan, instances of the Azure Functions host are dynamically added and removed based on the number of incoming events. This serverless plan scales automatically, and you're charged for compute resources only when your functions are running. On a Consumption plan, a function execution times out after a configurable period of time.

In the dedicated App Service plan, your function apps run on dedicated VMs on Basic, Standard, Premium, and Isolated SKUs, which is the same as other App Service apps. Dedicated VMs are allocated to your function app, which means the functions host can be always running. App Service plans support Linux. Durable Functions The primary use case for

Durable Functions is simplifying complex, stateful coordination problems in serverless applications. Function chaining and Fan-in/fan-out are two of the typical application patterns that can benefit from Durable Functions. Can you briefly describe how they operate?

> Click to see suggested answer

● Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function.

● Fan-out/fan-in refers to the pattern of executing multiple functions in parallel, and then waiting for all to finish. Often some aggregation work is done on results returned from the functions.









**Mobile PNS service creation along with Push notification**.

az appservice plan create --name mywebappplan --resource-group myResourceGroup --sku FREE

az webapp create --name mytestwebapp101 --resource-group myResourceGroup --plan mywebappplan

az webapp deployment source config --name mytestwebapp101 --resource-group myResourceGroup --repo-url "https://github.com/Azure-Samples/php-docs-hello-world" --branch master --manual-integration

//Mobile deployment

https://docs.microsoft.com/en-us/appcenter/sdk/push/uwp

1. go to azure-> create a resource-> search for Mobile app quickstart

2. after the resource is ready -> click on quickstart

3. select the OS which we need

4. download the solution-> install nuget packages

5. creating a new azure sql db , with azure sql server

//https://azure.github.io/AppService/update/2019/02/26/Changes-to-data-connections-UX.html

6. add dataconnection string

Data Source=tcp:mobileserver101.database.windows.net,1433;Initial Catalog={your\_catalogue};User ID={your\_username};Password={your\_password}

Data Source=tcp:mobileserver101.database.windows.net,1433;Initial Catalog=mobiledb;User ID=mobileadmin;Password=Passw0rd

DefaultEndpointsProtocol=https;AccountName=testfps;AccountKey=zCSqMYSzFWaGBBpK2IopyN3Te7Zecvx8TsdBg/Y1PmLfBo/pcQknmPZFKFI2Dtr+4mm9MhUhpcKuyAR6Q0VdyA==

7. Add the connection string to your Web/mobile app

=>In App Service, you can manage connection strings for your application by using the Configuration option in the menu.

=>To add a connection string:

=>Click on the Application settings tab.

=>Click on [+] New connection string.

=>You will need to provide Name, Value and Type for your connection string.

If your are adding a connection string to a SQL Azure database choose SQLAzure under type.

If your are adding a connection to an Azure Storage account, chose Custom under type.

=>NOTE If you are adding a connection string because you are planning on using the Easy API or Easy Table features, then the connection strings used by this features expect the following specific names:

Azure SQL database: MS\_TableConnectionString

Azure Storage account: MS\_AzureStorageAccountConnectionString

https://apps.dev.microsoft.com/#/application/82aa66a2-55f4-41b9-a26f-6fc682ed96a8

Push notifications:

2 types

1. system/out of app notifications

2. in-app push notifications

-------------------------------------------------------------------------------------------------------------------------------

1. open visual studio

2. create a new project->go to generic store template

3. open server explorer

right click on project-> add connected service->it will open services manager

create a new service (if you want to create a new mobile service)

(it will map it)

4. enter all the details and click create

open azure portal-> validate it

5. again go back to VS click on app.cs file it will show the key

right click project-> add-> push notifications

it will show Select an app name for WNS( notification service)

->click next-> finish

6. open a existing project-> press F5->

now it will open a window -> enter some value->

Part 3:

What is Table storage Azure Table storage stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of Table storage include:

● Storing TBs of structured data capable of serving web scale applications

● Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access

● Quickly querying data using a clustered index

● Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries You can use Table storage to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table storage concepts Table storage contains the following components:

● URL format: Azure Table Storage accounts use this format: http://<storage account>.table. core.windows.net/<table>

● Azure Cosmos DB Table API accounts use this format: http://<storage account>.table. cosmosdb.azure.com/<table>

● You can address Azure tables directly using this address with the OData protocol. For more information, see OData.org1.

● Accounts: All access to Azure Storage is done through a storage account.

● All access to Azure Cosmos DB is done through a Table API account.

● Table: A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties.

● Entity: An entity is a set of properties, similar to a database row. An entity in Azure Storage can be up to 1MB in size. An entity in Azure Cosmos DB can be up to 2MB in size.

● Properties: A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has three system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

Choosing an appropriate PartitionKey

Your choice of PartitionKey should balance the need to enable the use of EGTs (to ensure consistency) against the requirement to distribute your entities across multiple partitions (to ensure a scalable solution). At one extreme, you could store all your entities in a single partition, but this may limit the scalability of your solution and would prevent the table service from being able to load-balance requests. At the other extreme, you could store one entity per partition, which would be highly scalable and which enables the table service to load-balance requests, but which would prevent you from using entity group transactions. An ideal PartitionKey is one that enables you to use efficient queries and that has sufficient partitions to ensure your solution is scalable. Typically, you will find that your entities will have a suitable property that distributes your entities across sufficient partitions. Note: For example, in a system that stores information about users or employees, UserID may be a good PartitionKey. You may have several entities that use a given UserID as the partition key. Each entity that stores data about a user is grouped into a single partition, and so these entities are accessible via entity group transactions, while still being highly scalable.

Optimizing queries for the Table service

The Table service automatically indexes your entities using the PartitionKey and RowKey values in a single clustered index, hence the reason that point queries are the most efficient to use. However, there are no indexes other than that on the clustered index on the PartitionKey and RowKey. Many designs must meet requirements to enable lookup of entities based on multiple criteria. For example, locating employee entities based on email, employee id, or last name.

Sorting data in the Table service

The Table service returns entities sorted in ascending order based on PartitionKey and then by RowKey. These keys are string values and to ensure that numeric values sort correctly, you should convert them to a fixed length and pad them with zeroes. For example, if the employee id value you use as the RowKey is an integer value, you should convert employee id 123 to 00000123.

Authorization in Azure Storage

Authorize with Shared Key

Every request made against a storage service must be authorized, unless the request is for a blob or container resource that has been made available for public or signed access. One option for authorizing a request is by using Shared Key. Use the Shared Key authorization scheme to make requests against the Table service using the REST API. Shared Key authorization for the Table service in version 2009-09-19 and later uses the same signature string as in previous versions of the Table service. An authorized request requires two headers: the Date or x-ms-date header and the Authorization header. The following sections describe how to construct these headers.

Specifying the Date Header

All authorized requests must include the Coordinated Universal Time (UTC) timestamp for the request. You can specify the timestamp either in the x-ms-date header, or in the standard HTTP/HTTPS Date header. If both headers are specified on the request, the value of x-ms-date is used as the request's time of creation. The storage services ensure that a request is no older than 15 minutes by the time it reaches the service. This guards against certain security attacks, including replay attacks. When this check fails, the server returns response code 403 (Forbidden).

Note: The x-ms-date header is provided because some HTTP client libraries and proxies automatically set the Date header, and do not give the developer an opportunity to read its value in order to include it in the authorized request. If you set x-ms-date, construct the signature with an empty value for the Date header.

Specifying the Authorization Header

An authorized request must include the Authorization header. If this header is not included, the request is anonymous and may only succeed against a container or blob that is marked for public access, or against a container, blob, queue, or table for which a shared access signature has been provided for delegated access. To authorize a request, you must sign the request with the key for the account that is making the request and pass that signature as part of the request.

The format for the Authorization header is as follows:

Authorization="[SharedKey|SharedKeyLite] <AccountName>:<Signature>"

Where SharedKey or SharedKeyLite is the name of the authorization scheme, AccountName is the name of the account requesting the resource, and Signature is a Hash-based Message Authentication Code (HMAC) constructed from the request and computed by using the SHA256 algorithm, and then encoded by using Base64 encoding. Note: It is possible to request a resource that resides beneath a different account, if that resource is publicly accessible. The following sections describe how to construct the Authorization header.

Constructing the Signature String

How you construct the signature string depends on which service and version you are authorizing against and which authorization scheme you are using. When constructing the signature string, keep in mind the following:

● The VERB portion of the string is the HTTP verb, such as GET or PUT, and must be uppercase.

● For Shared Key authorization for the Blob, Queue, and File services, each header included in the signature string may appear only once. If any header is duplicated, the service returns status code 400 (Bad Request).

● The values of all standard HTTP headers must be included in the string in the order shown in the signature format, without the header names. These headers may be empty if they are not being specified as part of the request; in that case, only the new-line character is required.

● If the x-ms-date header is specified, you may ignore the Date header, regardless of whether it is specified on the request, and simply specify an empty line for the Date portion of the signature string. It is acceptable to specify both x-ms-date and Date; in this case, the service uses the value of x-ms-date.

● If the x-ms-date header is not specified, specify the Date header in the signature string, without including the header name.

● All new-line characters (\n) shown are required within the signature string.

● The signature string includes canonicalized headers and canonicalized resource strings. Canonicalizing these strings puts them into a standard format that is recognized by Azure Storage. For detailed information on constructing the CanonicalizedHeaders and CanonicalizedResource strings that make up part of the signature string, see the appropriate sections later in this topic.

Table Service (Shared Key authorization)

The format of the signature string for Shared Key against the Table service is the same for all versions. The Shared Key signature string for a request against the Table service differs slightly from that for a request against the Blob or Queue service, in that it does not include the CanonicalizedHeaders portion of the string. Additionally, the Date header in this case is never empty even if the request sets the x-ms-date header. If the request sets x-ms-date, that value is also used for the value of the Date header. To encode the signature string for a request against the Table service made using the REST API, use the following format: StringToSign = VERB + "\n" + Content-MD5 + "\n" + Content-Type + "\n" + Date + "\n" + CanonicalizedResource;

Note: Beginning with version 2009-09-19, the Table service requires that all REST calls include the DataServiceVersion and MaxDataServiceVersion headers.

Establishing a stored access policy

A stored access policy provides an additional level of control over service-level shared access signatures (SAS) on the server side. Establishing a stored access policy serves to group shared access signatures and to provide additional restrictions for signatures that are bound by the policy. You can use a stored access policy to change the start time, expiry time, or permissions for a signature, or to revoke it after it has been issued. The following storage resources support stored access policies:

● Blob containers

● File shares

● Queues

● Tables Note:

Note that a stored access policy on a container can be associated with a shared access signature granting permissions to the container itself or to the blobs it contains. Similarly, a stored access policy on a file share can be associated with a shared access signature granting permissions to the share itself or to the files it contains. Stored access policies are currently not supported for account SAS.

Creating or Modifying a Stored Access Policy

To create or modify a stored access policy, call the Set ACL operation for the resource with a request body that specifies the terms of the access policy. The body of the request includes a unique signed identifier of your choosing, up to 64 characters in length, and the optional parameters of the access policy, as follows: <?xml version="1.0" encoding="utf-8"?> <SignedIdentifiers> <SignedIdentifier> <Id>unique-64-char-value</Id> <AccessPolicy> <Start>start-time</Start> <Expiry>expiry-time</Expiry> <Permission>abbreviated-permission-list</Permission> </AccessPolicy> </SignedIdentifier> </SignedIdentifiers>

Table entity range restrictions (startpk, startrk, endpk, and endrk) cannot be specified in a stored access policy. A maximum of five access policies may be set on a container, table, or queue at any given time. Each SignedIdentifier field, with its unique Id field, corresponds to one access policy. Attempting to set more than five access policies at one time results in the service returning status code 400 (Bad Request).

Modifying or Revoking a Stored Access Policy

To modify the parameters of the stored access policy, you can call the access control list operation for the resource type to replace the existing policy, specifying a new start time, expiry time, or set of permissions. For example, if your existing policy grants read and write permissions to a resource, you can modify it to grant only read permissions for all future requests. In this case, the signed identifier of the new policy, as specified by the ID field, would be identical to the signed identifier of the policy you are replacing. To revoke a stored access policy, you can either delete it, or rename it by changing the signed identifier. Changing the signed identifier breaks the associations between any existing signatures and the stored access policy. Deleting or renaming the stored access policy immediately effects all of the shared access signatures associated with it. To remove a single access policy, call the resource's Set ACL operation, passing in the set of signed identifiers that you wish to maintain on the container. To remove all access policies from the resource, call the Set ACL operation with an empty request body.

Cross-Origin Resource Sharing (CORS) Support for the Azure Storage Services

Beginning with version 2013-08-15, the Azure storage services support Cross-Origin Resource Sharing (CORS) for the Blob, Table, and Queue services. The File service supports CORS beginning with version 2015-02-21. CORS is an HTTP feature that enables a web application running under one domain to access resources in another domain. Web browsers implement a security restriction known as same-origin policy that prevents a web page from calling APIs in a different domain; CORS provides a secure way to allow one domain (the origin domain) to call APIs in another domain. See the CORS specification2 for details on CORS. You can set CORS rules individually for each of the storage services, by calling Set Blob Service Properties, Set File Service Properties, Set Queue Service Properties, and Set Table Service Properties. Once you set the CORS rules for the service, then a properly authenticated request made against the service from a different domain will be evaluated to determine whether it is allowed according to the rules you have specified. Important: CORS is not an authentication mechanism. Any request made against a storage resource when CORS is enabled must either have a proper authentication signature, or must be made against a public resource. CORS is not supported for Premium Storage accounts.

Table service REST API

Table services resources

The Table service exposes the following resources via the REST API:

● Account. The storage account is a uniquely identified entity within the storage system. The storage account is the parent namespace for the Table service. All tables are associated with an account.

● Tables. The Tables resource represents the set of tables within a given storage account.

● Entity. An individual table stores data as a collection of entities.

Resource URI Syntax

The base URI for Table service resources is the storage account: https://myaccount.table.core.windows.net

To list the tables in a given storage account, to create a new table, or to delete a table, refer to the set of tables in the specified storage account:

https://myaccount.table.core.windows.net/Tables

To return a single table, name that table within the Tables collection, as follows: https://myaccount.table.core.windows.net/Tables('MyTable')

To query entities in a table, or to insert, update, or delete an entity, refer to that table directly within the storage account. This basic syntax refers to the set of all entities in the named table: https://myaccount.table.core.windows.net/MyTable()

The format for addressing data resources for queries conforms to that specified by the OData Protocol Specification. You can use this syntax to filter entities based on criteria specified on the URI. Note that all values for query parameters must be URL encoded before they are sent to the Azure storage services.

Supported HTTP Operations

Each resource supports operations based on the HTTP verbs GET, PUT, HEAD, and DELETE. The verb, syntax, and supported HTTP version(s) for each operation appears on the reference page for each operation. For a complete list of operation reference pages, see Table Service REST API3.

Query timeout and pagination

The Table service supports the following two types of query operations:

● The Query Tables operation returns the list of tables within the specified storage account. The list of tables may be filtered according to criteria specified on the request.

● The Query Entities operation returns a set of entities from the specified table. Query results may be filtered according to criteria specified on the request.

● A query against the Table service may return a maximum of 1,000 items at one time and may execute for a maximum of five seconds. If the result set contains more than 1,000 items, if the query did not complete within five seconds, or if the query crosses the partition boundary, the response includes headers which provide the developer with continuation tokens to use in order to resume the query at the next item in the result set. Continuation token headers may be returned for a Query Tables operation or a Query Entities operation.

● Note that the total time allotted to the request for scheduling and processing the query is 30 seconds, including the five seconds for query execution.

● It is possible for a query to return no results but to still return a continuation header.

● The continuation token headers are shown in the following table.

Inserting and updating entities

To insert or update an entity, you include with the request an an OData ATOM or OData JSON entity that specifies the properties and data for the entity. The Insert Entity operation inserts a new entity with a unique primary key, formed from the combination of the PartitionKey and the RowKey. The Update Entity operation replaces an existing entity with the same PartitionKey and RowKey. The Merge Entity operation updates the properties of an existing entity, but does not replace the entity. The Insert Or Merge Entity operation creates a new entity with a unique primary key or updates the properties of an existing entity, but does not replace the entity. The Insert Or Replace Entity operation creates a new entity with a unique primary key or replaces an existing entity.

Constructing the Atom Feed

The Atom feed for an insert or update operation defines the entity's properties by specifying their names and data types, and sets the values for those properties. The content element contains the entity's property definitions, which are specified within the m:properties element. The property's type is specified by the m:type attribute. Here is an example of an Atom feed for an Insert Entity operation: <?xml version="1.0" encoding="utf-8" standalone="yes"?>

<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">

<title />

<author> <name />

</author> <id />

<content type="application/xml">

<m:properties>

<d:Address>Mountain View</d:Address>

<d:Age m:type="Edm.Int32">23</d:Age>

<d:AmountDue m:type="Edm.Double">200.23</d:AmountDue>

<d:BinaryData m:type="Edm.Binary" m:null="true" />

<d:CustomerCode m:type="Edm.Guid">c9da6455-213d-42c9-9a793e9149a57833</d:CustomerCode> <d:CustomerSince m:type="Edm. DateTime">2008-07-10T00:00:00</d:CustomerSince>

<d:IsActive m:type="Edm.Boolean">true</d:IsActive>

<d:NumOfOrders m:type="Edm.Int64">255</d:NumOfOrders> <d:PartitionKey>mypartitionkey</d:PartitionKey>

<d:RowKey>myrowkey1</d:RowKey>

</m:properties> </content> </entry>

Note: Atom payloads are supported only in versions prior to 2015-12-11. Beginning with version 201512-11, payloads must be in JSON. Constructing the JSON Feed

To insert or update an entity using the OData JSON format, create a JSON object with property names as keys together with their property values. You may need to include the property type if it cannot be inferred through OData JSON type detection heuristics.

The JSON payload corresponding to the ATOM example above is as follows:

{ "Address":"Mountain View",

"Age":23,

"AmountDue":200.23,

"CustomerCode@odata.type":"Edm.Guid",

"CustomerCode":"c9da6455-213d-42c9-9a79-3e9149a57833", "CustomerSince@odata.type":"Edm.DateTime",

"CustomerSince":"2008-07-10T00:00:00",

"IsActive":true,

"NumOfOrders@odata.type":"Edm.Int64",

"NumOfOrders":"255",

"PartitionKey":"mypartitionkey",

"RowKey":"myrowkey"

}

Review Questions Module 1

review questions

Storage tables REST API

Storage tables offer a set of transactional functionality that mirrors the traditional Create, Read, Update, and Delete methods found in many other data sources. Can you name the basic operations that can be performed on entities by using the representational state transfer (REST) pattern.

> Click to see suggested answer

Base URL https://[account].table.core.windows.net/[table]

Method Endpoint

GET https://[account].table.core.windows. net/table

PUT https://[account].table.core.windows. net/table

POST https://[account].table.core.windows. net/[table]

DELETE https://[account].table.core.windows. net/table

MERGE https://[account].table.core.windows. net/table

Connecting to Azure Storage

Every request made against Microsoft Azure Storage must be authorized, unless the request is for a binary large object (blob) or container resource that has been made available for public or signed access. What are some of the options for authorizing a request?

> Click to see suggested answer

One option for authorizing a request is by using a shared key with the REST API. Another method of authorizing access to a storage account is by using a connection string. A connection string includes the authentication information required for your application to access data in an Azure storage account at run time. You can configure connection strings to:

● Connect to the Azure Storage Emulator.

● Access an Azure storage account.

● Access specified resources in Azure via a Shared Access Signature (SAS).

Develop solutions that use Azure Cosmos DB storage

Azure Cosmos DB overview

Azure Cosmos DB Microsoft

Azure Cosmos DB is a database service native to Azure that focuses on providing a high-performance database regardless of your selected API or data model. Azure Cosmos DB offers multiple APIs and models that can be used interchangeably for various application scenarios.

Core functionality

Global replication Azure Cosmos DB has a feature referred to as turnkey global distribution that automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure.

Consistency levels

Commercial distributed databases fall into two categories: databases that do not offer well-defined, provable consistency choices at all and databases that offer two extreme programmability choices (strong versus eventual consistency). The former burdens application developers with the minutia of their replication protocols and expects them to make difficult tradeoffs among consistency, availability, latency, and throughput. The latter pressure them to choose one of the two extremes.

Azure Cosmos DB provides five consistency levels: strong, bounded-staleness, session, consistent prefix, and eventual. Bounded-staleness, session, consistent prefix, and eventual are referred to as relaxed consistency models, because they provide less consistency than strong, which is the most highly consistent model available.

The consistency levels range from very strong consistency—where reads are guaranteed to be visible across replicas before a write is fully committed across all replicas—to eventual consistency, where writes are readable immediately, and replicas are eventually consistent with the primary.

Consistency Level Description

Strong When a write operation is performed on your primary database, the write operation is replicated to the replica instances. The write operation is committed (and visible) on the primary only after it has been committed and confirmed by all replicas.

Bounded Stateless This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.

Session This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.

Consistent Prefix This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.

Eventual This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary. This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize it's transactions.

Choose the right consistency level for your application

Distributed databases relying on replication for high availability, low latency or both, make the

fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Most

commercially available distributed databases ask developers to choose between the two extreme

consistency models: strong consistency and eventual consistency. Azure Cosmos DB allows developers

to choose among the five well-defined consistency models: strong, bounded staleness, session,

consistent prefix, and eventual. Each of these consistency models is well-defined, intuitive and can be

used for specific real-world scenarios. Each of the five consistency models provide availability and

performance tradeoffs and are backed by comprehensive SLAs. The following simple considerations will

help you make the right choice in many common scenarios. SQL API and Table API Consider the

following points if your application is built by using Cosmos DB SQL API or Table API

● For many real world scenarios, session consistency is optimal and it's the recommended option.

● If your application requires strong consistency, it is recommended that you use bounded staleness

consistency level.

● If you need stricter consistency guarantees than the ones provided by session consistency and single

digit millisecond latency for writes, it is recommended that you use bounded staleness consistency level.

Consistency guarantees in practice

You may get stronger consistency guarantees in practice. Consistency guarantees for a read operation

correspond to the freshness and ordering of the database state that you request. Read-consistency is tied

to the ordering and propagation of the write/update operations.

● When the consistency level is set to

bounded staleness, Cosmos DB guarantees that the clients always read the value of a previous write, with

a lag bounded by the staleness window.

● When the consistency level is set to strong, the staleness

window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the

write operation.

● For the remaining three consistency levels, the staleness window is largely dependent

on your workload. For example, if there are no write operations on the database, a read operation with

eventual, session, or consistent prefix consistency levels is likely to yield the same results as a read

operation with strong consistency level.

If your Cosmos DB account is configured with a consistency level other than the strong consistency, you

can find out the probability that your clients may get strong and

consistent reads for your workloads by looking at the Probabilistic Bounded Staleness (PBS) metric. This

metric is exposed in the Azure portal.

Probabilistic bounded staleness shows how eventual is your eventual consistency. This metric provides an

insight into how often you can get a stronger consistency

than the consistency level that you have currently configured on your Cosmos DB account. In other words,

you can see the probability (measured in milliseconds) of getting strongly consistent reads for a

combination of write and read regions.

Consistency levels and Azure Cosmos DB APIs Five consistency

models offered by Azure Cosmos DB are natively supported by the Azure Cosmos DB SQL API. When you

use Azure Cosmos DB, the SQL API is the default. Azure Cosmos DB also provides native support for wire

protocol-compatible APIs for popular databases. Databases include MongoDB, Apache Cassandra,

Gremlin, and Azure Table storage. These databases don't offer precisely defined consistency models or

SLA-backed guarantees for consistency levels. They typically provide only a subset of the five consistency

models offered by Azure Cosmos DB. For the SQL API, Gremlin API, and Table API, the default consistency

level configured on the Azure Cosmos DB account is used. The following sections show the mapping

between the data consistency requested by an OSS client driver for Apache Cassandra 4.x and MongoDB

3.4. This document also shows the corresponding Azure Cosmos DB consistency levels for Apache

Cassandra and MongoDB.

Mapping between Apache Cassandra and Azure Cosmos DB consistency levels

This table shows the “read consistency” mapping between the Apache Cassandra 4.x client and the default

consistency level in Azure Cosmos DB. The table shows multi-region and single-region deployments.

Azure Cosmos DB supported APIs

Today, Azure Cosmos DB can be accessed by using five different APIs. The underlying data structure in

Azure Cosmos DB is a data model based on atom record sequences that enabled Azure Cosmos DB to

support multiple data models. Because of the flexible nature of atom record sequences, Azure Cosmos

DB will be able to support many more models and APIs over time.

MongoDB API

The MongoDB API in Azure Cosmos DB acts as a massively scalable MongoDB service powered by the

Azure Cosmos DB platform. It is compatible with existing MongoDB libraries, drivers, tools, and

applications.

Table API

The Table API in Azure Cosmos DB is a key-value database service built to provide premium capabilities

(for example, automatic indexing, guaranteed low latency, and global distribution) to existing Azure

Table storage applications without making any app changes.

Gremlin API

The Gremlin API in Azure Cosmos DB is a fully managed, horizontally scalable graph database service

that makes it easy to build and run applications that work with highly connected datasets supporting

Open Graph APIs (based on the Apache TinkerPop specification, Apache Gremlin).

Collections

In the Azure Cosmos DB SQL API, databases are essentially containers for collections. Collections are

where you place individual documents. A collection is intrinsically elastic—it automatically grows and

shrinks as you add or remove documents. Each collection is assigned a throughput value, and that value

dictates the maximum throughput for that collection and its corresponding documents. Alternatively,

you can assign the throughput at the database level and share the throughput values among the

collections in the database. If you have a set of documents that needs throughput beyond the limits of

an individual collection, you can distribute the documents among multiple collections. Each collection

has its own distinct throughput level. If a particular collection is seeing spikes in throughput, you can

manage its throughput level in isolation by increasing or decreasing the value. This change to the

throughput level of a particular collection will not cause side effects for the other collections. This allows

you to adjust to meet the performance needs of any workload in isolation

Collection types

Azure Cosmos DB containers can be created as fixed or unlimited in the Azure portal. Fixed-size

containers have a maximum limit of 10 GB and a 10,000 RU/s throughput. To create a container as

unlimited, you must specify a partition key and a minimum throughput of 1,000 RU/s. Azure Cosmos DB

containers can also be configured to share throughput among the containers in a database. If you

created a fixed container with no partition key or a throughput less than 1,000 RU/s, the container will

not automatically scale. To migrate the data from a fixed container to an unlimited container, you need

to use the data migration tool or the Change Feed library.

Partitioning

Azure Cosmos DB provides containers for storing data called collections (for documents), graphs, or

tables. Containers are logical resources and can span one or more physical partitions or servers. The

number of partitions is determined by Azure Cosmos DB based on the storage size and throughput

provisioned for a container or set of containers. If you are already familiar with the sharding pattern, the

idea of dynamic partitioning is not very different

Create and update documents by using code

Manage collections and documents by using the Microsoft .NET SDK

To get started with the Azure Cosmos DB SQL API, you will need the Microsoft.Azure.DocumentDB.

Core1 package from NuGet. First, you will need to add the following using directives to the top of your

class file:

using Microsoft.Azure.Documents;

using Microsoft.Azure.Documents.Client;

Then, you can create a DocumentClient instance by using the endpoint from your Azure Cosmos DB

account and one of your keys:

DocumentClient client = new DocumentClient(new Uri("[endpoint]"), "[key]");

To reference any resource in the software development kit (SDK), you will need a URI. The UriFactory

class contains a series of static helper methods that can create URIs for common Azure Cosmos DB

resources. In this example, we will create a URI for a collection:

Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName, collectionName);

Now, you can use the CreateDocumentAsync method of the DocumentClient class to insert a C# object

into the collection. You can use any C# type you want for your documents, because the SDK doesn't

require a specific base type:

var document = new { firstName = "Alex", lastName = "Leh" }

await this.client.CreateDocumentAsync(collectionUri, document);

If you want to query the database, you can perform SQL queries by using the SqlQuerySpec class:

var query = client.CreateDocumentQuery<Family>( collectionUri,

new SqlQuerySpec()

{ QueryText = "SELECT \* FROM f WHERE (f.surname = @lastName)",

Parameters = new SqlParameterCollection()

{

new SqlParameter("@lastName", "Andt")

}

},

DefaultOptions

);

var families = query.ToList();

Alternatively, you can use the language-integrated query (LINQ) feature of C# with the SDK. The LINQ

expressions will be automatically translated into the appropriate SQL query:

var query = client.CreateDocumentQuery<Family>(collectionUri)

.Where(d => d.Surname = "Andt")

.Select(d => new { Name = d.Id, City = d.Address?.City)

.AsDocumentQuery();

var families = query.ToList();

Review Questions

Module 2 review questions

Azure Cosmos DB Core functionality

Azure Cosmos DB has a feature referred to as turnkey global distribution that automatically replicates

data to other Azure datacenters across the globe without the need to manually write code or build a

replication infrastructure. Can you name, and describe, the different consistency levels?

> Click to see suggested answer

Consistency Level Description

Strong When a write operation is performed on your

primary database, the write operation is replicated to the replica instances. The write operation is

committed (and visible) on the primary only after it has been committed and confirmed by all replicas.

Bounded Stateless This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.

Session This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.

Consistent Prefix This level has loose consistency but guarantees that when updates show up in replicas,

they will show up in the correct order (that is, as prefixes of other updates) without any gaps.

Eventual This level has the loosest consistency and essentially commits any write operation against the primary

immediately. Replica transactions are asynchronously handled and will eventually (over time) be

consistent with the primary. This tier has the best performance, because the primary database does not

need to wait for replicas to commit to finalize it's transactions.

Partitioning Azure Cosmos DB provide

containers for storing data called collections (for documents), graphs, or tables. Containers are logical

resources and can span one or more physical partitions or servers. Can you describe the differences

between physical and logical partitions?

> Click to see suggested answer

A physical partition is a fixed amount of reserved solid-state drive (SSD) backend storage combined with

a variable amount of compute resources (CPU and memory). Each physical partition is replicated for

high availability.

A logical partition is a partition within a physical partition that stores all the data associated with a single

partition key value. Partition ranges can be dynamically subdivided to seamlessly grow the database as

the application grows while simultaneously maintaining high availability.

Azure SQL overview

The Azure SQL Database service SQL Database is a general-purpose relational database managed service

in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML. SQL

Database delivers dynamically scalable performance within two different purchasing models: a vCore

based purchasing model and a DTU-based purchasing model. SQL Database also provides options such

as column store indexes for extreme analytic analysis and reporting, and in-memory OLTP for extreme

transactional processing. Microsoft handles all patching and updating of the SQL code base seamlessly

and abstracts away all management of the underlying infrastructure. Azure SQL Database provides the

following deployment options for an Azure SQL database:

● As a single database with its own set of resources managed via a logical server

● As a pooled database in an elastic pool with a shared set of resources managed via a logical server

● As a part of a collection of databases known as a managed instance that contains system and user

databases and sharing a set of resources The following illustration shows these deployment options

Review Questions

Module 3 review questions

Azure SQL database SQL Database is a general-purpose relational database managed service in

Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML. What are the

three deployment options for Azure SQL database?

> Click to see suggested answer Azure SQL Database

provides the following deployment options for an Azure SQL database:

● As a single database with its own set of resources managed via a logical server

● As a pooled database in an elastic pool with a shared set of resources managed via a logical server

● As a part of a collection of databases known as a managed instance that contains system and user

databases and sharing a set of resources

DbContext implementation

To use Entity Framework to query, insert, update, and delete data using .NET objects, you first need to

create a model that maps the entities and relationships defined in your model to tables in a database.

After you have a model, the primary class your application interacts with is System.Data.Entity.

DbContext (often referred to as the context class). You can use a DbContext class associated to a model

to (name as many as you can):

>Click to see suggested answer

● Write and execute queries.

● Materialize query results as entity objects.

● Track changes that are made to those objects.

● Persist object changes back on the database.

● Bind objects in memory to UI controls. The recommended way to work with the context is to define a

class that derives from DbContext and exposes DbSet properties that represent collections of the

specified entities in the context.







Implement Azure Security

Microsoft identity platform overview

Microsoft identity platform is an evolution of the Azure Active

Directory (Azure AD) identity service and developer platform. It allows developers to build applications

that sign in all Microsoft identities, get tokens to call Microsoft Graph, other Microsoft APIs, or APIs that

developers have built. It’s a full-featured platform that consists of an authentication service, open

source libraries, application registration and configuration (through a developer portal and application

API), full developer documentation, code samples, and other developer content. The Microsoft identity

platform supports industry standard protocols such as OAuth 2.0 and OpenID Connect.

Application types

in Azure AD Azure Active Directory (Azure AD) supports authentication for a variety of modern app

architectures, all of them based on industry-standard protocols OAuth 2.0 or OpenID Connect. The

following diagram illustrates the scenarios and application types, and how different components can be

added:

These are the five primary application scenarios supported by Azure AD:

● Single-page application (SPA):

A user needs to sign in to a single-page application that is secured by Azure AD.

● Web browser to web

application: A user needs to sign in to a web application that is secured by Azure AD.

● Native

application to web API: A native application that runs on a phone, tablet, or PC needs to authenticate a

user to get resources from a web API that is secured by Azure AD.

● Web application to web API: A web

application needs to get resources from a web API secured by Azure AD.

● Daemon or server application

to web API: A daemon application or a server application with no web user interface needs to get

resources from a web API secured by Azure AD.

App registration

Any application that outsources authentication to Azure AD must be registered in a directory. This step

involves telling Azure AD about your application, including the URL where it’s located, the URL to send

replies after authentication, the URI to identify your application, and more. This information is required

for a few key reasons:

● Azure AD needs to communicate with the application when handling sign-on or exchanging tokens.

The information passed between Azure AD and the application includes the following:

● Application ID URI - The identifier for an application. This value is sent to Azure AD during

authentication to indicate which application the caller wants a token for. Additionally, this value is

included in the token so that the application knows it was the intended target.

● Reply URL and Redirect URI - For a web API or web application, the Reply URL is the location where

Azure AD will send the authentication response, including a token if authentication was successful. For a

native application, the Redirect URI is a unique identifier to which Azure AD will redirect the user-agent

in an OAuth 2.0 request.

● Application ID - The ID for an application, which is generated by Azure AD when the application is

registered. When requesting an authorization code or token, the Application ID and Key are sent to

Azure AD during authentication.

● Key - The key that is sent along with an Application ID when authenticating to Azure AD to call a web

API.

● Azure AD needs to ensure the application has the required permissions to access your directory data,

other applications in your organization, and so on

Types of permissions Azure AD defines two kinds of permissions:

● Delegated permissions - Are used by apps that have a signed-in user present. For these apps, either the user or an administrator consents to the permissions that the app requests and the app is delegated permission to act as the signed-in user when making calls to an API. Depending on the API, the user may not be able to consent to the API directly and would instead require an administrator to provide “admin consent”.

● Application permissions - Are used by apps that run without a signed-in user present; for example, apps that run as background services or daemons. Application permissions can only be consented by an administrator because they are typically powerful and allow access to data across user-boundaries, or data that would otherwise be restricted to administrators.

Effective permissions are the permissions that your app will have when making requests to an API.

● For delegated permissions, the effective permissions of your app will be the least privileged intersection of the delegated permissions the app has been granted (through consent) and the privileges of the currently signed-in user. Your app can never have more privileges than the signed-in user. Within organizations, the privileges of the signed-in user may be determined by policy or by membership in one or more administrator roles. To learn which administrator roles can consent to delegated permissions, see Administrator role permissions in Azure AD. For example, assume your app has been granted the User.ReadWrite.All delegated permission in Microsoft Graph. This permission nominally grants your app permission to read and update the profile of every user in an organization. If the signed-in user is a global administrator, your app will be able to update the profile of every user in the organization. However, if the signed-in user is not in an administrator role, your app will be able to update only the profile of the signed-in user. It will not be able to update the profiles of other users in the organization because the user that it has permission to act on behalf of does not have those privileges.

● For application permissions, the effective permissions of your app are the full level of privileges implied by the permission. For example, an app that has the User.ReadWrite.All application permission can update the profile of every user in the organization.

Types of consent Applications in Azure AD rely on consent in order to gain access to necessary resources or APIs. There are a number of kinds of consent that your app may need to know about in order to be successful. If you are defining permissions, you will also need to understand how your users will gain access to your app or API.

● Static user consent - Occurs automatically during the OAuth 2.0 authorize flow when you specify the resource that your app wants to interact with. In the static user consent scenario, your app must have already specified all the permissions it needs in the app's configuration in the Azure portal. If the user (or administrator, as appropriate) has not granted consent for this app, then Azure AD will prompt the user to provide consent at this time.

● Dynamic user consent - Is a feature of the v2 Azure AD app model. In this scenario, your app requests a set of permissions that it needs in the OAuth 2.0 authorize flow for v2 apps. If the user has not consented already, they will be prompted to consent at this time. Learn more about dynamic consent.

● Important: Dynamic consent can be convenient, but presents a big challenge for permissions that require admin consent, since the admin consent experience doesn't know about those permissions at consent time. If you require admin privileged permissions or if your app uses dynamic consent, you must register all of the permissions in the Azure portal (not just the subset of permissions that require admin consent). This enables tenant admins to consent on behalf of all their users.

● Admin consent - Is required when your app needs access to certain high-privilege permissions. Admin consent ensures that administrators have some additional controls before authorizing apps or users to access highly privileged data from the organization.

Implement OAuth2 authentication

Authorize access to web applications using OpenID Connect

OpenID Connect is a simple identity layer built on top of the OAuth 2.0 protocol. OAuth 2.0 defines mechanisms to obtain and use access tokens to access protected resources, but they do not define standard methods to provide identity information. OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. It provides information about the end user in the form of an id\_token that verifies the identity of the user and provides basic profile information about the user.

OpenID Connect is our recommendation if you are building a web application that is hosted on a server and accessed via a browser.

Register your application with your AD tenant

First, you need to register your application with your Azure Active Directory (Azure AD) tenant. This will give you an Application ID for your application, as well as enable it to receive tokens.

● Sign in to the Azure portal.

● Choose your Azure AD tenant by clicking on your account in the top right corner of the page, followed by clicking on the Switch Directory navigation and then select the appropriate tenant.

● Skip this step, if you've only one Azure AD tenant under your account or if you've already selected the appropriate Azure AD tenant.

● In the left hand navigation pane, click on Azure Active Directory.

● Click on App Registrations and click on New application registration.

● Follow the prompts and create a new application. It doesn't matter if it is a web application or a native application for this tutorial.

● For Web Applications, provide the Sign-On URL, which is the base URL of your app, where users can sign in e.g <http://localhost:12345>.

● For Native Applications provide a Redirect URI, which Azure AD will use to return token responses. Enter a value specific to your application, .e.g <http://MyFirstAADApp>.

● Once you've completed registration, Azure AD will assign your application a unique client identifier, the Application ID. You need this value in the next sections, so copy it from the application page.

● To find your application in the Azure portal, click App registrations, and then click View all applications.

Validate the id\_token

Just receiving an id\_token is not sufficient to authenticate the user; you must validate the signature and verify the claims in the id\_token per your app's requirements. The Azure AD endpoint uses JSON Web Tokens (JWTs) and public key cryptography to sign tokens and verify that they are valid.

You can choose to validate the id\_token in client code, but a common practice is to send the id\_token to a backend server and perform the validation there. Once you've validated the signature of the id\_token, there are a few claims you are required to verify.

You may also wish to validate additional claims depending on your scenario. Some common validations include:

● Ensuring the user/organization has signed up for the app.

● Ensuring the user has proper authorization/privileges

● Ensuring a certain strength of authentication has occurred, such as multi-factor authentication. Once you have validated the id\_token, you can begin a session with the user and use the claims in the id\_token to obtain information about the user in your app. This information can be used for display, records, personalization, etc.

Send a sign-out request

When you wish to sign the user out of the app, it is not sufficient to clear your app's cookies or otherwise end the session with the user. You must also redirect the user to the end\_session\_endpoint for sign-out. If you fail to do so, the user will be able to reauthenticate to your app without entering their credentials again, because they will have a valid single sign-on session with the Azure AD endpoint. You can simply redirect the user to the end\_session\_endpoint listed in the OpenID Connect metadata document:

GET https://login.microsoftonline.com/common/oauth2/logout? post\_logout\_redirect\_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F

Single sign-out When you redirect the user to the end\_session\_endpoint, Azure AD clears the user's session from the browser. However, the user may still be signed in to other applications that use Azure AD for authentication. To enable those applications to sign the user out simultaneously, Azure AD sends an HTTP GET request to the registered LogoutUrl of all the applications that the user is currently signed in to. Applications must respond to this request by clearing any session that identifies the user and returning a 200 response. If you wish to support single sign out in your application, you must implement such a LogoutUrl in your application's code. You can set the LogoutUrl from the Azure portal:

1. Navigate to the Azure Portal.

2. Choose your Active Directory by clicking on your account in the top right corner of the page.

3. From the left hand navigation panel, choose Azure Active Directory, then choose App registrations and select your application.

4. Click on Settings, then Properties and find the Logout URL text box. Token Acquisition Many web apps need to not only sign the user in, but also access a web service on behalf of that user using OAuth. This scenario combines OpenID Connect for user authentication while simultaneously acquiring an authorization\_code that can be used to get access\_tokens using the OAuth Authorization Code Flow.

Get Access Tokens

To acquire access tokens, you need to modify the sign-in request from above:

// Line breaks for legibility only

GET [https://login.microsoftonline.com/{tenant}/oauth2/authorize](https://login.microsoftonline.com/%7btenant%7d/oauth2/authorize)?

client\_id=6731de76-14a6-49ae-97bc-6eba6914391e // Your registered Application Id &response\_type=id\_token+code

&redirect\_uri=http%3A%2F%2Flocalhost%3a12345 // Your registered Redirect Uri, url encoded

&response\_mode=form\_post // `form\_post' or 'fragment'

&scope=openid

&resource=https%3A%2F%2Fservice.contoso.com%2F // The identifier of the protected resource (web API) that your application needs access to

&state=12345 // Any value, provided by your app

&nonce=678910 // Any value, provided by your app

By including permission scopes in the request and using response\_type=code+id\_token, the authorize endpoint ensures that the user has consented to the permissions indicated in the scope query parameter, and return your app an authorization code to exchange for an access token. Successful response A successful response using response\_mode=form\_post looks like: POST /myapp/ HTTP/1.1 Host: localhost Content-Type: application/x-www-form-urlencoded

id\_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WWmNB...&code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUYBrq...&state=12345

Understanding the OAuth2 implicit grant flow in  Azure Active Directory

The OAuth2 implicit grant is notorious for being the grant with the longest list of security concerns in the OAuth2 specification. And yet, that is the approach implemented by ADAL JS and the one we recommend when writing SPA applications. What gives? It’s all a matter of tradeoffs: and as it turns out, the implicit grant is the best approach you can pursue for applications that consume a Web API via JavaScript from a browser.

What is the OAuth2 implicit grant?

The quintessential OAuth2 authorization code grant is the authorization grant that uses two separate endpoints. The authorization endpoint is used for the user interaction phase, which results in an authorization code. The token endpoint is then used by the client for exchanging the code for an access token, and often a refresh token as well. Web applications are required to present their own application credentials to the token endpoint, so that the authorization server can authenticate the client. The OAuth2 implicit grant is a variant of other authorization grants. It allows a client to obtain an access token (and id\_token, when using OpenId Connect) directly from the authorization endpoint, without contacting the token endpoint nor authenticating the client. This variant was designed for JavaScript based applications running in a Web browser: in the original OAuth2 specification, tokens are returned in a URI fragment. That makes the token bits available to the JavaScript code in the client, but it guarantees they won’t be included in redirects toward the server. Returning tokens via browser redirects directly from the authorization endpoint. It also has the advantage of eliminating any requirements for cross origin calls, which are necessary if the JavaScript application is required to contact the token endpoint.

An important characteristic of the OAuth2 implicit grant is the fact that such flows never return refresh tokens to the client. The next section shows how this isn’t necessary and would in fact be a security issue.

Suitable scenarios for the OAuth2 implicit grant

The OAuth2 specification declares that the implicit grant has been devised to enable user-agent applications – that is to say, JavaScript applications executing within a browser. The defining characteristic of such applications is that JavaScript code is used for accessing server resources (typically a Web API) and for updating the application user experience accordingly. Think of applications like Gmail or Outlook Web Access: when you select a message from your inbox, only the message visualization panel changes to display the new selection, while the rest of the page remains unmodified. This characteristic is in contrast with traditional redirect-based Web apps, where every user interaction results in a full page postback and a full page rendering of the new server response.

Applications that take the JavaScript based approach to its extreme are called single-page applications, or SPAs. The idea is that these applications only serve an initial HTML page and associated JavaScript, with all subsequent interactions being driven by Web API calls performed via JavaScript. However, hybrid approaches, where the application is mostly postback-driven but performs occasional JS calls, are not uncommon – the discussion about implicit flow usage is relevant for those as well.

Redirect-based applications typically secure their requests via cookies, however, that approach does not work as well for JavaScript applications. Cookies only work against the domain they have been generated for, while JavaScript calls might be directed toward other domains. In fact, that will frequently be the case: think of applications invoking Microsoft Graph API, Office API, Azure API – all residing outside the domain from where the application is served. A growing trend for JavaScript applications is to have no backend at all, relying 100% on third party Web APIs to implement their business function.

Currently, the preferred method of protecting calls to a Web API is to use the OAuth2 bearer token approach, where every call is accompanied by an OAuth2 access token. The Web API examines the incoming access token and, if it finds in it the necessary scopes, it grants access to the requested operation. The implicit flow provides a convenient mechanism for JavaScript applications to obtain access tokens for a Web API, offering numerous advantages in respect to cookies:

● Tokens can be reliably obtained without any need for cross origin calls – mandatory registration of the redirect URI to which tokens are return guarantees that tokens are not displaced

● JavaScript applications can obtain as many access tokens as they need, for as many Web APIs they target – with no restriction on domains

● HTML5 features like session or local storage grant full control over token caching and lifetime management, whereas cookies management is opaque to the app

● Access tokens aren’t susceptible to Cross-site request forgery (CSRF) attacks

The implicit grant flow does not issue refresh tokens, mostly for security reasons. A refresh token isn’t as narrowly scoped as access tokens, granting far more power hence inflicting far more damage in case it is leaked out. In the implicit flow, tokens are delivered in the URL, hence the risk of interception is higher than in the authorization code grant. However, a JavaScript application has another mechanism at its disposal for renewing access tokens without repeatedly prompting the user for credentials. The application can use a hidden iframe to perform new token requests against the authorization endpoint of Azure AD: as long as the browser still has an active session (read: has a session cookie) against the Azure AD domain, the authentication request can successfully occur without any need for user interaction.

Implement managed identities for Azure resources

Managed identities for Azure resources overview

Note: Managed identities for Azure resources is a feature of Azure Active Directory. Each of the Azure services that support managed identities for Azure resources are subject to their own timeline. Make sure you review the availability5 status of managed identities for your resource and known issues6 before you begin.

A common challenge when building cloud applications is how to manage the credentials in your code for authenticating to cloud services. Keeping the credentials secure is an important task. Ideally, the credentials never appear on developer workstations and aren't checked into source control. Azure Key Vault provides a way to securely store credentials, secrets, and other keys, but your code has to authenticate to Key Vault to retrieve them.

The managed identities for Azure resources feature in Azure Active Directory (Azure AD) solves this problem. The feature provides Azure services with an automatically managed identity in Azure AD. You can use the identity to authenticate to any service that supports Azure AD authentication, including Key Vault, without any credentials in your code.

The managed identities for Azure resources feature is free with Azure AD for Azure subscriptions. There's no additional cost.

Note: Managed identities for Azure resources is the new name for the service formerly known as Managed Service Identity (MSI)

Terminology

The following terms are used throughout the managed identities for Azure resources documentation set:

● Client id - a unique identifier generated by Azure AD that is tied to an application and service principal during its initial provisioning.

● Principal id - the object id of the service principal object for your managed identity that is used to grant role based access to an Azure resource.

● Azure Instance Metadata Service (IMDS) - a REST Endpoint accessible to all IaaS VMs created via the Azure Resource Manager. The endpoint is available at a well-known non-routable IP address (169.254.169.254) that can be accessed only from within the VM.

How the managed identities for Azure resources works There are two types of managed identities:

● A system-assigned managed identity is enabled directly on an Azure service instance. When the identity is enabled, Azure creates an identity for the instance in the Azure AD tenant that's trusted by the subscription of the instance. After the identity is created, the credentials are provisioned onto the instance. The lifecycle of a system-assigned identity is directly tied to the Azure service instance that it's enabled on. If the instance is deleted, Azure automatically cleans up the credentials and the identity in Azure AD.

● A user-assigned managed identity is created as a standalone Azure resource. Through a create process, Azure creates an identity in the Azure AD tenant that's trusted by the subscription in use. After the identity is created, the identity can be assigned to one or more Azure service instances. The lifecycle of a user-assigned identity is managed separately from the lifecycle of the Azure service instances to which it's assigned.

Implement multi-factor authentication

Multi-factor authentication

When a user logs into an application, they typically provide a username and password. The password is provided by the user as a piece of evidence to the authentication system that the user is who they claim to be. The password is considered one factor proving the user’s identity. A user could have other factors that proves their identity, such as:

● A physical badge from the company.

● Knowledge of the answers to security questions.

● A mobile device, registered with the company, that can receive notifications, phone calls, or SMS messages.

● Their physical appearance that can be captured by a camera device.

● Their fingerprint that could be captured by a biometric scanner. Unfortunately, a single factor can potentially be compromised either intentionally or unintentionally. A badge can be stolen and used by an unauthorized party. During a robbery, someone could ask you to use your fingerprint on a device. A mobile company could accidentally send SMS messages to another device. In security best practices, it is recommended to use two or more factors when authenticating users. This practice is referred to as multi-factor authentication. Using an enterprise as an example, the company could require employees to scan their badges and then enter their passwords as two factors of authentication. In the world of security, it is often recommended to have two of the following factors:

● Knowledge – Something that only the user knows (security questions, password, or PIN).

● Possession – Something that only the user has (corporate badge, mobile device, or security token).

● Inherence – Something that only the user is (fingerprint, face, voice, or iris). The security of two-step verification lies in its layered approach. Compromising multiple authentication factors presents a significant challenge for attackers. Even if an attacker manages to learn the user's password, it is useless without possession of the additional authentication method.

Multi-factor authentication with Azure AD

Azure Multi-Factor Authentication (MFA) is a two-step verification solution that is built in to Azure AD. Administrators can configure approved authentication methods to ensure that at least two factors are used while still keeping the sign-in process as streamlined as possible. There are two ways to enable MFA:

● The first option is to enable each user for MFA. When users are enabled individually, they perform two-step verification each time they sign in. There are a few exceptions, such as when they sign in from trusted IP addresses or when the remembered devices feature is turned on.

● The second option is to set up a conditional access policy that requires two-step verification under certain conditions. This method uses the Azure AD Identity Protection risk policy to require two-step verification based only on the sign-in risk for all cloud applications. Once MFA is enabled, administrators can choose which methods of authentication are available to users. Once users enroll, they must choose at least one method from the list that the administrator has enabled. These methods include:

Method Description

Call to phone Places an automated voice call. The user answers the call and presses # on the phone keypad to authenticate. The phone number is not synchronized to on-premises Active Directory.

Text message to phone Sends a text message that contains a verification code. The user is prompted to enter the verification code into the sign-in interface. This process is called one-way SMS. Two-way SMS means that the user must text back a particular code.

Notification through mobile app Sends a push notification to your phone or registered device. The user views the notification and selects Verify to complete the verification.

Verification code from mobile app The Microsoft Authenticator app generates a new OAuth verification code every 30 seconds. The user enters the verification code into the sign-in interface.

Review questions

Module 1 review questions

Microsoft identity platform

Azure Active Directory (Azure AD) supports authentication for a variety of modern app architectures, all of them based on industry-standard protocols OAuth 2.0 or OpenID Connect. What are the two categories of applications that can be developed and integrated with Azure AD?

> Click to see suggested answer

● Single tenant application - A single tenant application is intended for use in one organization. These are typically line-of-business (LoB) applications written by an enterprise developer. A single tenant application only needs to be accessed by users in one directory, and as a result, it only needs to be provisioned in one directory. These applications are typically registered by a developer in the organization.

● Multi-tenant application - A multi-tenant application is intended for use in many organizations, not just one organization. These are typically software-as-a-service (SaaS) applications written by an independent software vendor (ISV). Multi-tenant applications need to be provisioned in each directory where they will be used, which requires user or administrator consent to register them. This consent process starts when an application has been registered in the directory and is given access to the Graph API or perhaps another web API. When a user or administrator from a different organization signs up to use the application, they are presented with a dialog that displays the permissions the application requires. The user or administrator can then consent to the application, which gives the application access to the stated data, and finally registers the application in their directory.

OAuth2 grant flows

The OAuth2 implicit grant is notorious for being the grant with the longest list of security concerns in the OAuth2 specification. For what type of application scenarios is it actually the recommended approach?

> Click to see suggested answer

The implicit grant presents more risks than other grants. However, the higher risk profile is largely due to the fact that it is meant to enable applications that execute active code, served by a remote resource to a browser. If you are planning an SPA architecture, have no backend components or intend to invoke a Web API via JavaScript, use of the implicit flow for token acquisition is recommended.

Claims-based authorization

Claims

Authorization is the process of determining which entities have permission to change, view, or otherwise access a computer resource. For example, in a business, only managers may be allowed to access the files of their employees. In the past, this was simple to accomplish with identity databases using protocols like Lightweight Directory Access Protocol (LDAP) or tools like Active Directory Domain Services. Whenever a user attempted to access an application, the application would query the identity database.

In a world where identity is usually managed by third-party providers, like Microsoft Azure Active Directory, Facebook, Google, LinkedIn, and Twitter, this information needs to be shared in a standardized way to applications. In the simplest workflow, the user needs to access an application, so they first log in using their social identity. Once they are logged in, the identity provider is trusted by the organization’s application and can share claims about that user with the application.

Claims-based authorization

Claims-based authorization is an approach where the authorization decision to grant or deny access is based on arbitrary logic that uses data available in claims to make the decision. Claims-based authorization, at its simplest, checks the value of a claim and allows access to a resource based on that value. For example, if you want access to a night club, the authorization process might be:

● The door security officer evaluates the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

In a relying party application, authorization determines what resources an authenticated identity is allowed to access and what operations it is allowed to perform on those resources. Improper or weak authorization leads to information disclosure and data tampering.

Claim-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying claims that the current user must possess and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based; the developer must build and register a policy expressing the claims requirements.

Role-based access control (RBAC) authorization

Role-based access control overview

Access management for cloud resources is a critical function for any organization that is using the cloud. Role-based access control (RBAC) helps you manage who has access to Azure resources, what they can do with those resources, and what areas they have access to. RBAC is an authorization system built on Azure Resource Manager that provides fine-grained access management of resources in Azure.

What can I do with RBAC? Here are some examples of what you can do with RBAC:

● Allow an application to access all resources in a resource group

● Allow one user to manage virtual machines in a subscription and another user to manage virtual networks

● Allow a DBA group to manage SQL databases in a subscription

● Allow a user to manage all resources in a resource group, such as virtual machines, websites, and subnets

Best practice for using RBAC

Using RBAC, you can segregate duties within your team and grant only the amount of access to users that they need to perform their jobs. Instead of giving everybody unrestricted permissions in your Azure subscription or resources, you can allow only certain actions at a particular scope. When planning your access control strategy, it's a best practice to grant users the least privilege to get their work done. The following diagram shows a suggested pattern for using RBAC.

Scope

Scope is the boundary that the access applies to. When you assign a role, you can further limit the actions allowed by defining a scope. This is helpful if you want to make someone a Website Contributor, but only for one resource group. In Azure, you can specify a scope at multiple levels: management group, subscription, resource group, or resource. Scopes are structured in a parent-child relationship.

When you grant access at a parent scope, those permissions are inherited to the child scopes. For example:

● If you assign the Owner role to a user at the management group scope, that user can manage everything in all subscriptions in the management group.

● If you assign the Reader role to a group at the subscription scope, the members of that group can view every resource group and resource in the subscription.

● If you assign the Contributor role to an application at the resource group scope, it can manage resources of all types in that resource group, but not other resource groups in the subscription.

Role assignments

A role assignment is the process of attaching a role definition to a user, group, service principal, or managed identity at a particular scope for the purpose of granting access. Access is granted by creating a role assignment, and access is revoked by removing a role assignment. The following diagram shows an example of a role assignment. In this example, the Marketing group has been assigned the Contributor role for the pharma-sales resource group. This means that users in the Marketing group can create or manage any Azure resource in the pharma-sales resource group. Marketing users do not have access to resources outside the pharma-sales resource group, unless they are part of another role assignment.

You can create role assignments using the Azure portal, Azure CLI, Azure PowerShell, Azure SDKs, or REST APIs. You can have up to 2000 role assignments in each subscription. To create and remove role assignments, you must have Microsoft.Authorization/roleAssignments/\* permission. This permission is granted through the Owner or User Access Administrator roles.

Deny assignments

Previously, RBAC was an allow-only model with no deny, but now RBAC supports deny assignments in a limited way. Similar to a role assignment, a deny assignment attaches a set of deny actions to a user, group, service principal, or managed identity at a particular scope for the purpose of denying access. A role assignment defines a set of actions that are allowed, while a deny assignment defines a set of actions that are not allowed. In other words, deny assignments block users from performing specified actions even if a role assignment grants them access. Deny assignments take precedence over role assignments. Currently, deny assignments are read-only and can only be set by Azure.

How RBAC determines if a user has access to a resource

The following are the high-level steps that RBAC uses to determine if you have access to a resource on the management plane. This is helpful to understand if you are trying to troubleshoot an access issue.

1. A user (or service principal) acquires a token for Azure Resource Manager.

2. The token includes the user's group memberships (including transitive group memberships).

3. The user makes a REST API call to Azure Resource Manager with the token attached.

4. Azure Resource Manager retrieves all the role assignments and deny assignments that apply to the resource upon which the action is being taken.

5. Azure Resource Manager narrows the role assignments that apply to this user or their group and determines what roles the user has for this resource.

6. Azure Resource Manager determines if the action in the API call is included in the roles the user has for this resource.

7. If the user doesn’t have a role with the action at the requested scope, access is not granted. Otherwise, Azure Resource Manager checks if a deny assignment applies.

8. If a deny assignment applies, access is blocked. Otherwise access is granted.

Classic subscription administrator roles, Azure RBAC roles, and Azure AD administrator roles

If you are new to Azure, you may find it a little challenging to understand all the different roles in Azure. This article helps explain the following roles and when you would use each:

● Classic subscription administrator roles

● Azure role-based access control (RBAC) roles

● Azure Active Directory (Azure AD) administrator roles

How the roles are related

To better understand roles in Azure, it helps to know some of the history. When Azure was initially released, access to resources was managed with just three administrator roles: Account Administrator, Service Administrator, and Co-Administrator. Later, role-based access control (RBAC) for Azure resources was added. Azure RBAC is a newer authorization system that provides fine-grained access management to Azure resources. RBAC includes many built-in roles, can be assigned at different scopes, and allows you to create your own custom roles. To manage resources in Azure AD, such as users, groups, and domains, there are several Azure AD administrator roles.

Review questions

Module 2 review questions

Authentication vs. authorization

The terms authentication and authorization can be confusing, can you describe what role each plays?

> Click to see suggested answer

To keep it simple, authentication is the act of verifying someone’s identity. When you authenticate someone, you are determining who they are. Authorization is the act of verifying that someone has access to a certain subsystem or operation. When you authorize someone, you are determining what they can do.

Role-based authorization in ASP.NET

Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class. Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles that the current user must be a member of to access the requested resource. Can you come up with some code snippets that illustrate how to do that?

> Click to see suggested answer

The following code limits access to any actions on the AdministrationController to users who are members of the Administrator role:

[Authorize(Roles = "Administrator")] public class AdministrationController : Controller { }

You can specify multiple roles as a comma separated list: [Authorize(Roles = "HRManager,Finance")]

public class SalaryController : Controller { }





