Exam Prep AZ-204

Written by Rob Das, in 2021
Last Updated in 2024

Forward

This document contains my study notes for the Microsoft certification exam, "AZ-204 Developing Solutions for Microsoft Azure". As I proceed through my career as an Azure developer, my study notes serve as a single point of reference to refresh my memory on the exam topics as needed. These study notes are my textbook.

I have made these study notes publicly and freely available to share with everyone. The study notes were my main tool for passing the exam and I hope they will help you to do the same.

I compiled these study notes in 2021 and used them to pass the certification exam for Microsoft Certified: Azure Developer Associate in late 2021. To create these study notes, I began by downloading the complete list of skills measured for AZ-204 in 2021, which is published by Microsoft at:

https://learn.microsoft.com/en-us/certifications/resources/study-guides/AZ-204

The list of skills measured was then used to create the table of contents for this document. I then studied each skill one-by-one and took extensive study notes. The notes were collected from various sources including all the recommended Microsoft Learning Paths, Microsoft online documentation, you tube videos, and online courses from various sites for which I had subscriptions. The notes were compiled using screen shots, copy-and-paste, and my own explanations as my knowledge and understanding grew. I make no claim as to the accuracy and currency of these study notes, they are simply the work of an enthusiastic student (myself) in the process of learning.

I have no wish to profit from these study notes, I just want to share them with others who might find them useful. As such, I will not be making any efforts to update these study notes as additional topics are added by Microsoft each year. I hope these study notes will make an excellent starting point, but you will have to check the current list of skills measured to find out what skills have been changed or added.

Good Luck and Happy Examing! 😊

Rob Das,
Certified Azure Developer,
Owner/Manager, DTEK Consulting Services Ltd.
http://azuredev.ca

21 April, 2023

Table of Contents

Study Notes for AZ-204 (circa 2021)

Top Level Skill Domains

1. Develop Azure compute solutions (25-30%)
2. Develop for Azure storage (15-20%)
3. Implement Azure security (20-25%)
4. Monitor, troubleshoot, and optimize Azure solutions (15-20%)
5. Connect to and consume Azure services and third-party services (15-20%)

1. Develop Azure compute solutions (25-30%)

Azure *compute* is an on-demand service for running cloud-based applications. It provides computing resources via virtual machines and containers. It also provides serverless computing to run apps without requiring setup or configuration of infrastructure.
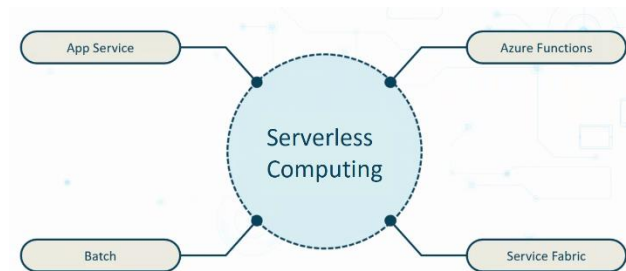
App Service hosts web apps, background applications, mobile app backends, APIs

Azure Functions respond to events

Batch is a VM based, scalable job scheduler

Service Fabric uses containers to deploy microservice architected applications

1.1 Implement IaaS solutions

- provision VMs
- configure, validate, and deploy ARM templates
- configure container images for solutions
- publish an image to the Azure Container Registry
- run containers by using Azure Container Instance

https://docs.microsoft.com/en-us/learn/paths/azure-administrator-prerequisites/

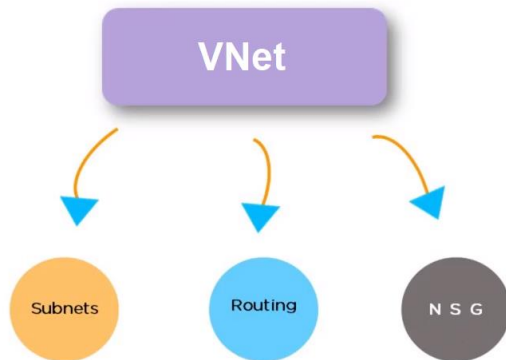https://docs.microsoft.com/en-us/learn/modules/network-fundamentals/

https://docs.microsoft.com/en-us/learn/modules/network-fundamentals-2/

- Subnet
    - logically segments the VNet using IP address ranges
    - can further be subdivided into *private* and *public* subnet
    - private subnet is isolated – blocked from the internet – but you can override using NAT
- Routing
    - you cannot access routers directly in Azure but can you create routing rules in your VNet
    - Azure automatically creates route tables for each subnet in order to manage traffic
- Network Security Group
    - serves as firewall to protect virtual machines within VNet
    - restricts inbound and outbound traffic based on IP address, port, and/or protocol

Provision VMs
Skill: provision VMs

https://docs.microsoft.com/en-us/learn/paths/deploy-a-website-with-azure-virtual-machines

https://docs.microsoft.com/en-us/azure/backup/backup-azure-vms-first-look-arm

To deploy (provision) VM's you need:

- Storage account – with at least one virtual disk for hosting the base OS
- Virtual network (VNet)
- Network interface – to communicate on the network
- network security group (NSG)
- Public IP address – *optional*

You also need to think about location, size, pricing, OS. You need to select Resource Group, Image (Windows or Linux)

Create a resource > Compute > choose template (ex. Win 2019 0r Ubuntu) > follow the wizard

Azure CLI
https://docs.microsoft.com/en-us/cli/azure/vm#az_vm_create

Create resource group and vm. Open port. Get public IP.

```
#create resource group
az group create \
    --name "test-rg" \
    --location "centralus"

#create win vm                                  #create linux vm
az vm create \                                  az vm create \
    --resource-group "test-rg" \                    --resource-group "test-rg" \
    --name "win2016-vm" \                           --name "linux-vm" \
    --image "win2016datacenter" \                   --image "UbuntuLTS" \
    --admin-username "robi" \                       --admin-username "robi" \
    --admin-password "aReallyGoodPasswordHere"      --authentication-type "ssh" \
                                                    --ssh-key-value ~/.ssh/id_rsa.pub

#open port
# creates nsg, adds and applies rule to vm
az vm open-port \                               az vm open-port \
    --resource-group "test-rg" \                    --resource-group "test-rg" \
    --name "win2016-vm" \                           --name "linux-vm" \
    --port "3389"                                   --port "22"

#get public ip
az vm list-ip-addresses \
    --resource-group "test-rg" \
    --name "win2016-vm"
```

Powershell
https://docs.microsoft.com/en-us/azure/virtual-machines/linux/quick-create-powershell

```
New-AzResourceGroup `
        -Name test-rg `
        -Location "East US"

New-AzVm `
    -ResourceGroupName "test-rg" `
    -Name "win2016-vm" `
    -Image "Win2019Datacenter" `
    -Location "East US" `
    -VirtualNetworkName "test-wp1-eus-network" `
    -SubnetName "default" `
    -SecurityGroupName "test-wp1-eus-nsg" `
    -PublicIpAddressName "test-wp1-eus-pubip" `
    -OpenPorts 80,3389

Get-AzPublicIpAddress `
    -ResourceGroupName "test-rg" `
    -Name "win2016-vm"
```

You can interact with every type of resource in Azure programmatically, where the creation and management of VMs form part of a larger application with complex logic.

Azure REST API
The Azure REST API provides developers with operations categorized by resource as well as the ability to create and manage VMs. Operations are exposed as URIs with corresponding HTTP methods (GET, PUT, POST, DELETE, and PATCH) and a corresponding response.

Azure Client SDK
The Azure Client SDK encapsulates the Azure REST API, making it much easier for developers to interact with Azure. The Azure Client SDKs are available for a variety of languages and frameworks, including C#, Java, Node.js, PHP, Python, Ruby, and Go.

Availability Set
https://docs.microsoft.com/en-us/learn/modules/configure-virtual-machine-availability/

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.
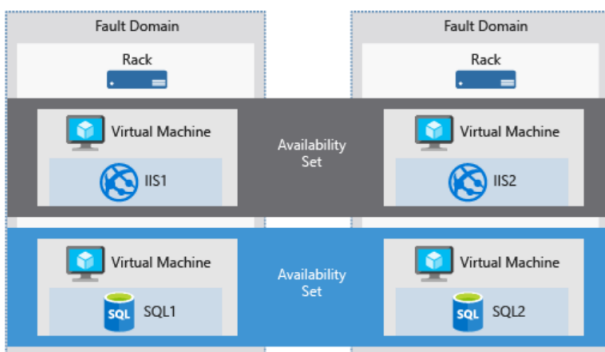
A fault domain is a physical group of hardware that shares a common set of hardware components, and that share a single point of failure. You can think of it as a rack within an on-premises datacenter.

An update domain is a logical group of hardware that can undergo maintenance, or be rebooted at the same time.

By default, new VMs are locked down. Apps can make outgoing requests, but the only inbound traffic allowed is from the virtual network and from Azure Load Balancer. To open ports (SSH, RDP, HTTP, FTP, etc.) two steps:

1. Create a network security group.
2. Create an inbound rule allowing traffic on the ports you need.

VMs – ARM Templates
Skill: configure, validate, and deploy ARM templates

https://docs.microsoft.com/en-gb/learn/paths/az-204-implement-iaas-solutions/

ARM template file structure:

schema	the location of the JSON schema file that describes the version of the template language and the structure of the JSON data	required
contentVersion	the version of your template – to document significant changes and to ensure you're deploying the right template	required
resources	the actual items you want to deploy or update in a resource group or a subscription	required
apiProfile	defines a collection of API versions for resource types	opt.
parameters	define values that are provided during deployment – can be provided by a parameter file, command-line parameters, or in Azure portal	opt.
variables	define values that are used to simplify template language expressions	opt.
functions	user-defined functions that are available within the template – simplify your template when complicated expressions are used repeatedly	opt.
output	the values that will be returned at the end of the deployment	opt.

In its simplest structure, a template has the following elements:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "",
  "apiProfile": "",
  "parameters": {  },
  "variables": {  },
  "functions": [  ],
  "resources": [  ],
  "outputs": {  }
}
```

ARM templates are JSON files that define the resources to deploy for your solution – for VMs, they are templates from which to create an exact copy of a VM or to create and deploy specific configurations.

Create resource templates from the Automation section for a specific VM:
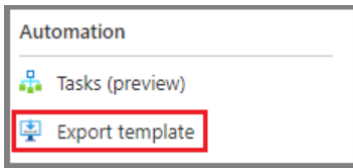
You have the option to save the resource template for later use, or immediately deploy a new VM based on this template. You can take that template and easily re-create multiple versions of your infrastructure, such as staging and production. You can parameterize fields such as the VM name and load the template repeatedly, using different parameters for each environment.

Exam Tip: Understand ARM template parameters and parameter files, functions, variables.
https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/template-syntax

https://docs.microsoft.com/en-us/learn/paths/deploy-manage-resource-manager-templates/

deploy an ARM template to Azure in one of the following ways:

- Deploy a local template
- Deploy a linked template
- Deploy in a continuous deployment pipeline

A deployment of Azure resources requires a resource group and an ARM template Json File. To deploy the template defined in the Template file, use either the Azure CLI command `az deployment group create` or the Azure PowerShell command `New-AzResourceGroupDeployment`.

```
# CLI example
az deployment group create \
  --name blanktemplate \
  --resource-group myResourceGroup \
  --template-file "{provide-the-path-to-the-template-file}"
```

```
# powershell example
New-AzResourceGroupDeployment `
  -Name {name of your resource group} `
  -Location "{location}"
```

Create Docker Images
Skill: configure container images for solutions

https://docs.microsoft.com/en-us/learn/modules/intro-to-docker-containers/

Container Image: A container image is a portable package that contains software (source-code, dependencies, libraries) as a single binary deployable package. It's this image that, when built and run, becomes our container. It serves as the deployment mechanism for your application. The term Container image is sometimes abbreviated to just *image*.

A container image is immutable. Once you've built a container image, it can't be changed. This is our guarantee that the container image we use in production is the same image used in development and QA.

Container: A container is an isolated environment that allows us to run images. It's a running instance of a container image. Inside that container is a compiled, running instance of your application.

Azure Container Registry (ACR): is a centralized storage service based on the open source docker registry. It makes your image accessible to all the servers that can execute your image. It allows you to build, store, and manage container images. You can think of ACR as a storage and cataloging service.

Azure Container Instances (ACI): Azure service that allows us to run one or more containers in Azure with no servers or infrastructures to manage. ACI serves as a container orchestrator that can pull down container images from ACR, compile and run them as containers. You can think of ACI as a runtime environment.

Docker is a containerization platform used to develop, ship, and run containers.
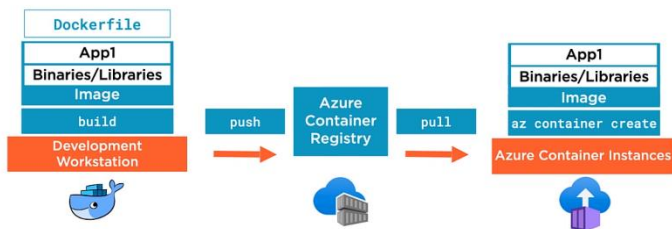
Building a Container Image
A Dockerfile is a text file that contains the instructions we use to build and run a Docker image.

FROM defines a base container image used for subsequent instructions in the docker file.

RUN executes a command inside the container. In the example, it will create a directory named app inside the container image.

WORKDIR sets the working directory for any subsequent instructions in this docker file.

COPY is used to copy application binaries into the container.

RUN executes scripts inside the container.

EXPOSE tells the container's runtime which port the application in the container is listening on.

ENTRYPOINT is used to define which script or binary to start when the container is started from this container image. Inside the [] is the command to run along with its parameters. Sometimes CMD is used instead of ENTRYPOINT.

Using this example docker file in the current directory, we execute the docker build command:

```
docker build -t webappimage:v1 .
```

This builds the container image from the docker file that's in the current directory, and name and tag the container image with the name webappimage, and the tag v1.

To run an instance of this container image on the local host as a container:

```
docker run –name webapp –publish 8080:80 –detach webappimage:v1
```

The -publish specifies the port that the container is going to listen on, on the local machine, 8080, as well as the port that the application is listening on, inside the container, 80.

With the container running, we can use curl to connect to the published port 8080.

```
curl http://localhost:8080
```

We can stop and remove the running webapp (the container):

```
docker stop webapp
docker rm webapp
```

Publish Image to Azure Container Registry (ACR)
Skill: publish an image to the Azure Container Registry

https://docs.microsoft.com/en-us/learn/modules/deploy-run-container-app-service/
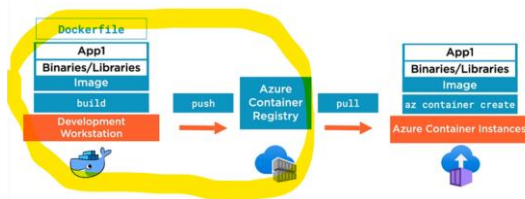
First, we must create an ACR. To create an ACR, we can use Azure Portal, PowerShell, or Azure CLI. For example, using CLI:

```
ACR_NAME='demo-acr'

az acr create \
    --resource-group 'test-rg' \
    --name $ACR_NAME \
    --sku Standard
```

Azure Container Registry (ACR) is a centralized storage service making your image accessible to all the servers that can execute your image. This makes your code independent from the servers and highly portable.

Note: this does not make the code live, it just puts in in a central place to make it available to the servers.

To push your container image to your ACR registry:

```
# 1. Log in to your Azure subscription
az login

# 2. Log in to your registry
az acr login --name $ACR_NAME

# 3. Tag the image that you want to upload to the registry - for example webappimage:v1
docker tag webappimage:v1 $ACR_NAME.azurecr.io/webappimage:v1

# 4. Push the image for example webappimage:v1
docker push $ACR_NAME.azurecr.io/webappimage:v1
```

Alternatively, you can push the image using ACR tasks:

```
az acr build -image "webappimage:v1-acr-task" -registry $ACR_NAME .
```

You can use ACR tasks to offload the build process of your application into Azure Container Registry. ACR tasks can be triggered when your team commits code into source control.

Azure Container Instances – Running Containers
Skill: run containers by using Azure Container Instance
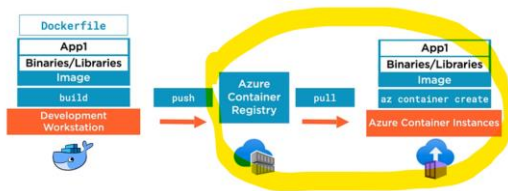
Before you can deploy your code to an Azure Container Instance or to a Web App, you need to set up access using access keys. We need to enable access to the container registry itself. Typically, this is done by creating a service principal. We need a container registry administrator user id and password. ACI will use this to pull down the container image from ACR.

```
#deploy container into ACI
SERVER=$(az acr show -query loginServer)
az container create \
    --resource-group test-rg \
    --name "my-app" \
    --ports 80 \
    --image $SERVER/webappimage:v1
```

Once deployed, the app in the container will be running in ACI. Get the URL as follows:

```
URL=$(az container show --name 'my-app' --query ipAddress.fqdn)
echo $URL
```

To test the app, point your browser at this URL.

1.2 Create Azure App Service Web Apps

- create an Azure App Service Web App
- enable diagnostics logging
- deploy code to a web app
- configure web app settings including SSL, API, and connection strings
- implement autoscaling rules, including scheduled autoscaling, and autoscaling by operational or system metrics

Azure App Service is a fully managed PaaS environment for hosting web applications, webapps-in-containers, mobile app backends, and custom-made APIs.

When working in containers, be aware of these Docker environment variables for App Service:

WEBSITES_CONTAINER_START_TIME_LIMIT	time to wait before evaluating the health and then restarting the container
WEBSITES_ENABLE_APP_SERVICE_STORAGE	if **true**, the **/home** dir will be shared across container instances and files will persist
WEBSITE_WEBDEPLOY_USE_SCM	set to **false** to deploy a container-based web app using WebDeploy/MSDeploy

The *App Service plan* has one or more VM's behind the scenes. Microsoft manages the compute layer, making sure the machines are patched, that they're highly available, and that they're functional. We can deploy multiple web apps onto an app service plan, and that app service plan is scalable. There are two types of app service plans: non-isolated and isolated. Non-isolated app service plans are not inside a virtual network – no isolation at the network layer. Isolated app service plans are fully isolated and dedicated environments for running highly scalable, high performance web apps. You can put isolated apps into virtual networks for enhanced security. The app service plan is where you choose the OS and the pricing tier.

> Web Jobs is another feature of Azure App Service that enables you to run a program or script in the same instance as a web app, API app, or mobile app. Azure Functions provides another way to run programs and scripts.   – OUT OF SCOPE

> App Service Environment (ASE) is an Azure App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale. Helps   – OUT OF SCOPE
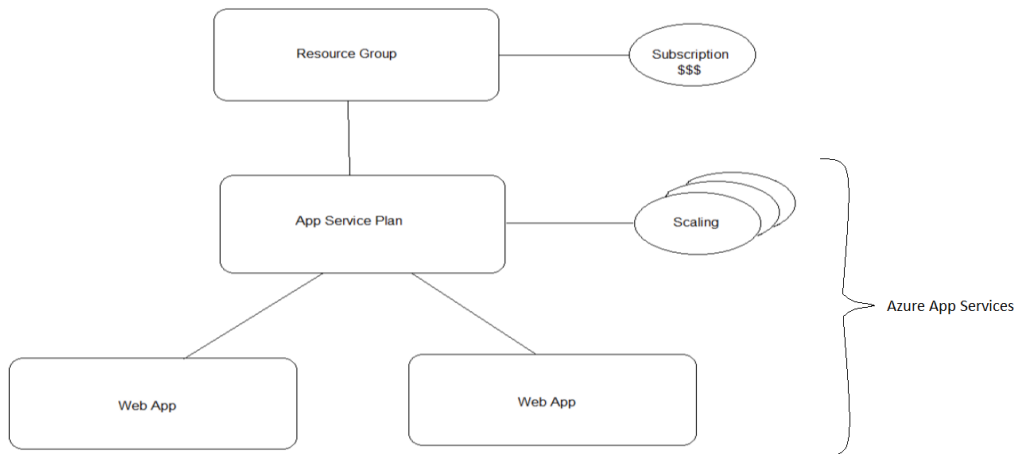
Create an Azure App Service Web App

Skill: create an Azure App Service Web App

https://docs.microsoft.com/en-us/learn/paths/create-azure-app-service-web-apps/

https://docs.microsoft.com/en-us/learn/paths/deploy-a-website-with-azure-app-service/

https://docs.microsoft.com/en-us/learn/modules/host-a-web-app-with-azure-app-service/

You need to create a resource group, app service plan, and then a web app.

From Azure Portal
You can create a Web App from the Azure portal > Create Resource > **Web App** > follow the wizard. Creating a web app allocates resources in App Service Plan, which you can use to host any web-based application that is supported by Azure. To publish, you can either push code selecting your runtime stack (.NET Core, Java, Node, …), or you can use a container image as your application.

From Azure CLI
You can create a web app using Azure CLI.
```
as group create --name my-resource-group --location westus2

az appservice plan create --name my-app-service-plan --resource-group my-resource-group \
    --sku s1 --is-linux

az webapp create --group my-resource-group --plan my-app-service-plan \
    --name my-web-app --runtime "node"
```

From PowerShell
You can create a web app using PowerShell.
```
New-AzResourceGroup -Name 'my-rg' -Location 'westus'

New-AzAppServicePlan -Name 'my-plan' -Location 'westus'`
    -ResourceGroupName 'my-rg' -Tier S1

New-AzWebApp -Name 'my-web-app' -Location 'westus' `
    -AppServicePlan 'my-plan' -ResourceGroupName 'my-rg'
```

From ARM Template
When it comes to building web apps with infrastructure as code, the native solution on Azure is using *ARM templates*. From Azure Portal > Create Resource > **Template** > Web App. Follow the wizard to select a resource group and provide the inputs needed to create the ARM template. Once we have an ARM template, we can then deploy it from the Portal, from CLI, or from PowerShell.

Enable Diagnostic Logging
Skill: enable diagnostics logging

https://docs.microsoft.com/en-us/learn/modules/capture-application-logs-app-service/

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-5.0

https://docs.microsoft.com/en-us/azure/app-service/troubleshoot-dotnet-visual-studio

https://docs.microsoft.com/en-us/azure/app-service/troubleshoot-diagnostic-logs

https://mderriey.com/2020/08/08/a-look-at-the-aspnet-core-logging-provider-for-app-service/

From Azure Portal
To enable app service diagnostic logging in the portal, go to App Service > monitoring > App Service logs.
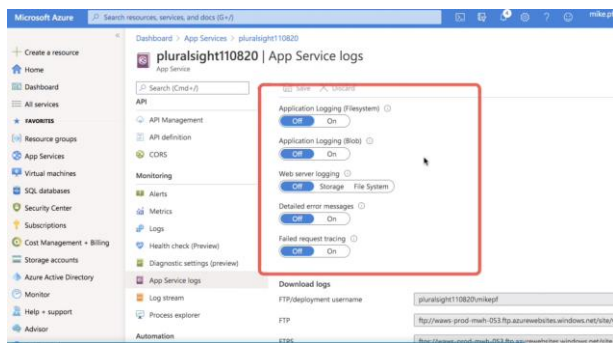
You can enable diagnostics logs using the Azure portal and you can select the level of error log (Disabled, Error, Warning, Information, Verbose).

Options for diagnostic logging:

- **Application Logging** Send a log message directly from your code.
- **Web Server Logging** 3 types of web server diagnostic logs can be enabled: Off, Storage, Filesystem.
- **Detailed error messages** for an HTTP status code of 400 or more, a detailed log is created.
- **Failed request tracing** Failed requests to the server can be enabled.
- **Deployment logging** This log is automatically enabled and gathers all information related to the deployment of your application.

From CLI

```
az webapp log config `
      --web-server-logging filesystem `
      --name my-app `
      --resource-group my-rg
```

Retrieving Diagnostic Logs

Use Windows File Explorer as an FTP client and create a connection to the App Service root folder in Azure. FTP connection credentials are available in the publish profile, which can be downloaded from Portal > App Service > Overview, and clicking on "Get publish profile".
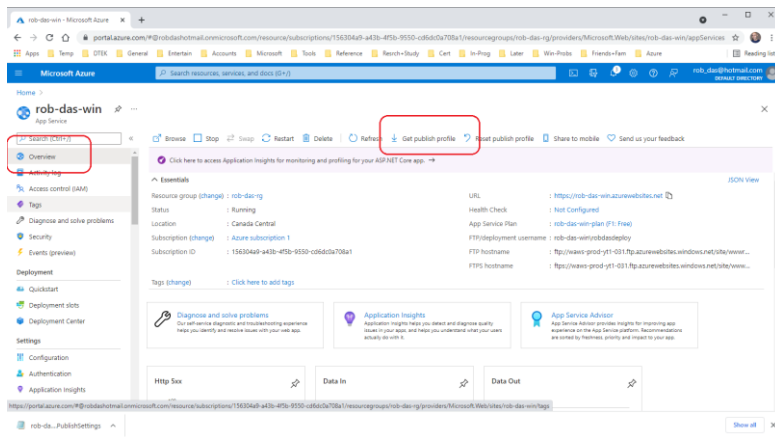
Or from the CLI:

```
az webapp log download --log-file \<_filename_\>.zip  --resource-group \<_resource group name_\> --name \<_app name_\>
```

Writing to Diagnostic Logs with C#

Install the NuGet package Microsoft.Extensions.Logging.AzureAppServices. Add the appropriate provider to the logging system:

```csharp
// Program.cs
using Microsoft.Extensions.Logging;

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(builder => builder.AddAzureWebAppDiagnostics())
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

In code, reference the `ILogger` as usual, with dependency injection.

```csharp
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;

        public HomeController(ILogger<HomeController> logger)
        {
            _logger = logger;
            _logger.LogError("TESTING LOGGING");
        }

        // your code here...
    }
```
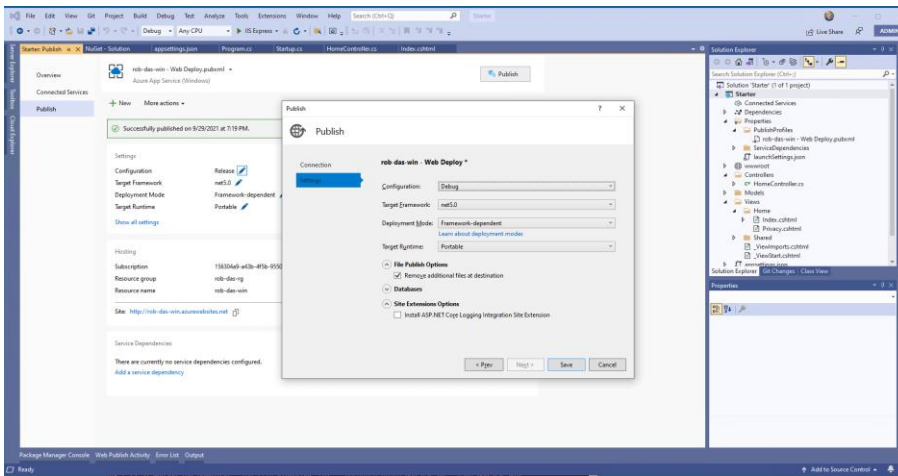
install asp net core logging integration site extension

https://www.visualstudiogeeks.com/devops/installing-aspnet-core-site-extensions-for-azure-app-service-using-arm

https://nugetmusthaves.com/Package/Microsoft.AspNetCore.AzureAppServices.SiteExtension

Deploy Code to a Web App
Skill: deploy code to a web app

https://docs.microsoft.com/en-us/learn/modules/publish-azure-web-app-with-visual-studio/

https://docs.microsoft.com/en-us/learn/modules/stage-deploy-app-service-deployment-slots/

https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots#Auto-Swap

https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots#specify-custom-warm-up

https://docs.microsoft.com/en-us/learn/modules/stage-deploy-app-service-deployment-slots/

For Azure App Services, Azure supports publishing your web app through deployment-as-code to the underlying services either directly to a runtime stack, or as a Docker container. Before configuring automated deployment, you first need to create an Azure app service plan in a resource group, and a web app in the app service. This can be done through the Azure Portal using the usual *+Create a resource* wizard approach.

Azure supports automated-deployment directly from several sources:

- Azure Repos (part of Azure DevOps)
- GitHub
- Bitbucket
- OneDrive
- Dropbox

You authorize Azure App Service to retrieve code from the source repository and then you enable continuous deployment in Azure. You associate your web app to point to a branch of one of these repositories, and anytime the branch is updated, the update automatically deploys to your App Service web application.

Go to Azure Portal > App Services > your App Service > Deployment > Deployment Center.
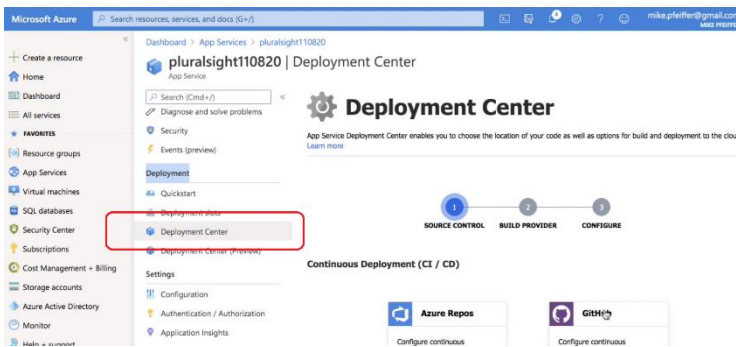
Follow the wizard:

1. Define the source control where we are storing our code. Scroll down under *Continuous Deployment (CI / CD)* and choose one of the sources listed, and then click the Authorize button.
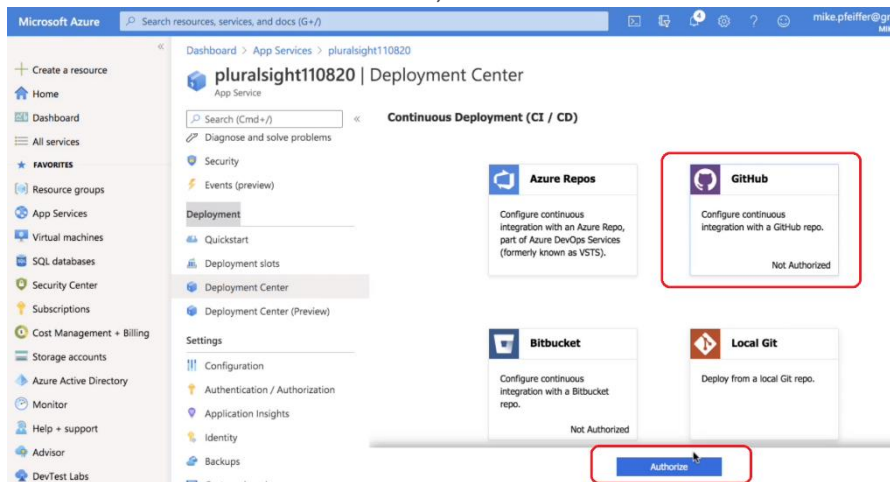
2. Who's going to build the code? For example, what service will do a compile on our .NET application? Click one of the build service choices, and click Continue.
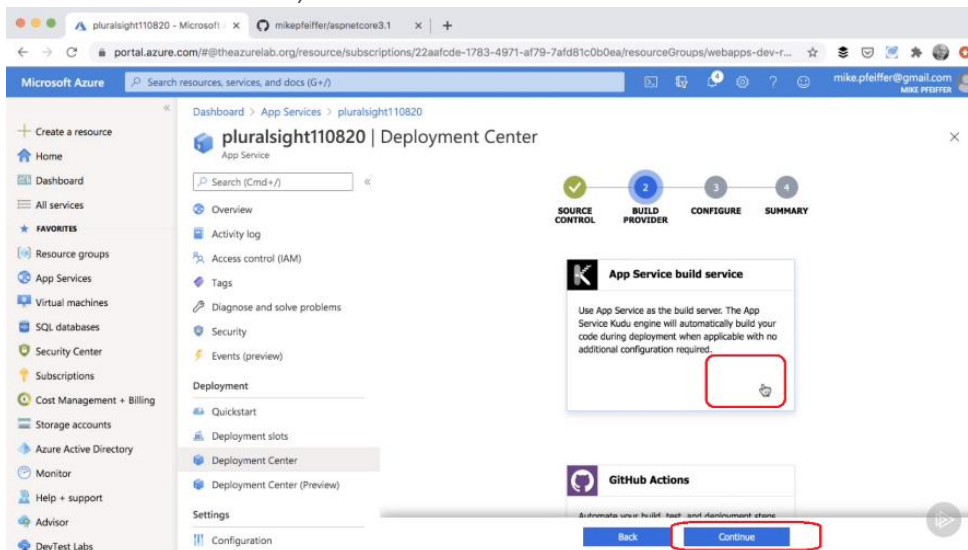
3. Click Finish. That will kick off the first deployment.

Azure also supports manual deployment:

- Git
- CLI (az webapp up)
- zip deploy
- WAR deploy (Java)
- Visual Studio
- FTP

Deployment Slots

Within a single Azure App Service web app, you can create multiple *deployment slots*. Each slot is a separate instance of that web app, and it has a separate hostname. However, each slot shares the resources of the App Service plan. You can deploy a different version of your web app into each slot. One slot is the production slot. This slot is the web app that users see when they connect. When ready to put a new app version into production, deploy it by swapping its slot with the production slot. Unlike a code deployment, a slot swap is instantaneous. If necessary, you can roll back the version by swapping the slots back.

The initial delay for compilation and other server actions before the first page is delivered is called a *cold start*. You can avoid a cold start by using slot swaps to deploy to production. When you swap a slot into production, you "warm-up" the app because your action sends a request to the root of the site to ensure that all compilation and caching tasks finish, and the site responds as fast as if it had been deployed for days. When you swap two slots, the app's configuration settings travels to the new slot. You can override this by configuring them as *slot settings*.

You can create a new slot or you can *clone* another slot. After you clone the settings, the configuration of the two slots can be changed independently. Although you can clone settings to a new slot, you can't clone content. The new slot is effectively a separate web app with a different hostname. You can control access to a slot by using IP address restrictions.

The *swap-with-preview* feature helps you discover problems before your app goes live into production. The swap proceeds in two phases so you can test the app in the source slot to make sure it works with the target slot configuration. If you find no problems, the hostnames for the two sites are swapped.

Auto swap brings swap-based deployment to automated deployment pipelines. When you configure a non-production slot for auto swap, Azure automatically swaps it whenever you push code into that slot. When you use auto swap, you can't test the new app version in the staging slot before the swap. To address this, you can deploy to a separate slot that's dedicated for testing.

Some apps might require *custom warm-up* actions before the swap. The *applicationInitialization* configuration element in web.config lets you specify custom initialization actions. The swap operation waits for this custom warm-up to finish before swapping with the target slot.

```
<system.webServer>
```

```
    <applicationInitialization>
        <add initializationPage="/" hostName="[app hostname]" />
        <add initializationPage="/Home/About" hostName="[app hostname]" />
    </applicationInitialization>
</system.webServer>
```

Web App Settings including SSL, API, and Connection Strings
Skill: configure web app settings including SSL, API, and connection strings

https://docs.microsoft.com/en-gb/learn/modules/configure-web-app-settings/

https://docs.microsoft.com/en-gb/learn/modules/configure-web-app-settings/2-configure-application-settings

https://docs.microsoft.com/en-gb/learn/modules/configure-web-app-settings/6-configure-security-certificates

Configuration of app services, including app settings and connection strings, can be done through the Azure Portal, or scripted using JSON. In the Portal, click on the desired app service, and select Settings > Configuration.
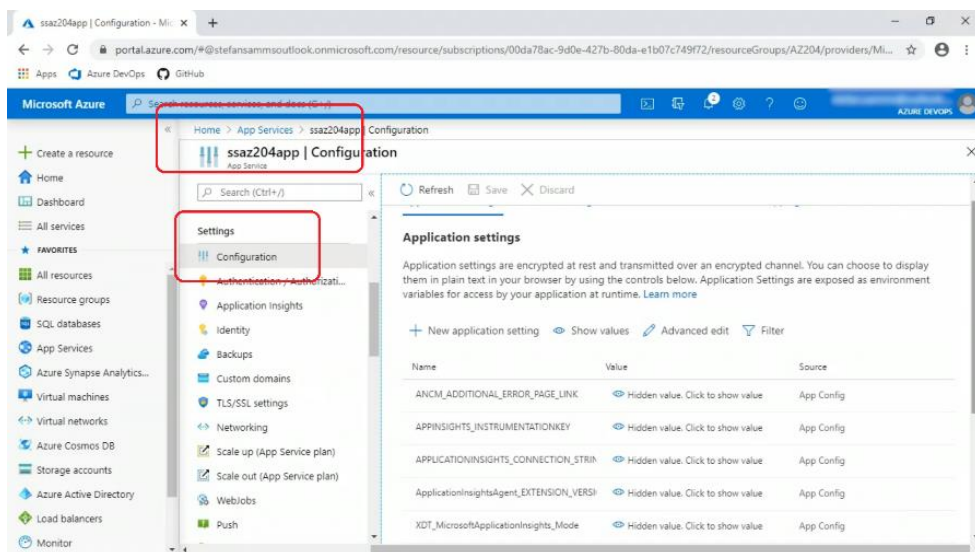
Configure web app security with SSL
To use HTTPS to secure your connection, you need to upload an SSL certificate to the web application in Azure. To use SSL, you need to be using at least the Basic tier service plan type. Public vs. private certificates (managed vs. unmanaged) – we can purchase from a certificate authority such as VeriSign, or we can generate private certificates using our own certificate authority. It's a matter of trust – will the client devices trust a private certificate?   In the properties of your app service plan, you can enforce HTTPS and TLS (transport layer security).

To add an SSL certificate to your App Service, in the portal go to TLS/SSL settings for you App Service. You can access these settings in the Configuration menu on the Settings section in the App Service blade.

If you set a variable in this section that matches a variable in Web.config or appsettings.json files, the value of the variables in the configuration files will be replaced with the value in your Azure Web App settings.1

Configure Database Connection String

To get a connection string for an Azure SQL database, click on "show database connection strings".

Then go to "Connection strings" under Settings/Configuration for your App Service.
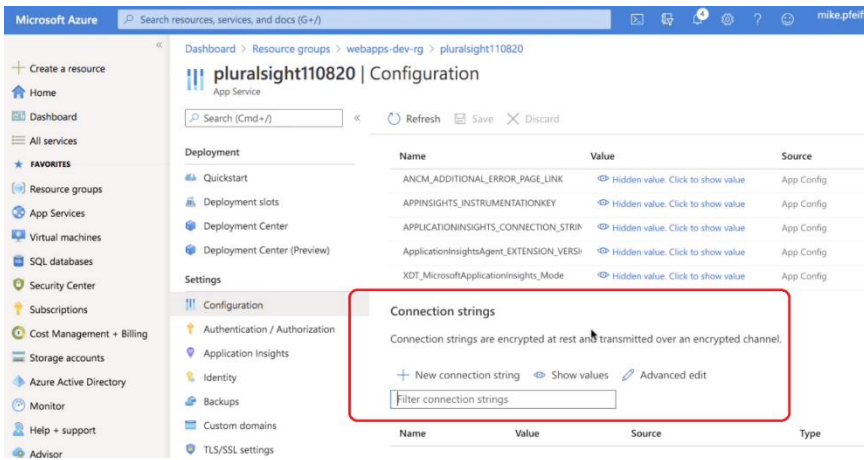
Then give the connection string a name, paste in your connection string and put in your password, and then select a database type.
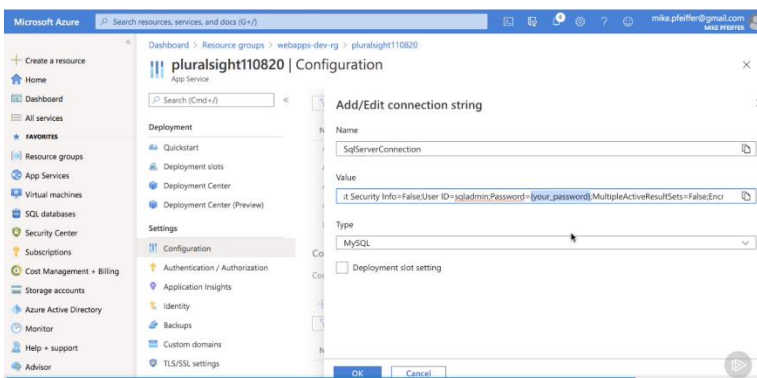
Autoscaling

Skill: implement autoscaling rules, including scheduled autoscaling, and autoscaling by operational or system metrics

https://docs.microsoft.com/en-us/learn/modules/app-service-autoscale-rules/

https://docs.microsoft.com/en-us/learn/modules/app-service-scale-up-scale-out/

https://docs.microsoft.com/en-us/learn/modules/scale-apps-app-service/

Autoscaling is a feature of the App Service Plan used by the web app.

Autoscaling requires you to configure autoscale rules. Autoscaling performs scaling in and out, as opposed to scaling up and down. Autoscaling doesn't have any effect on the CPU power, memory, or storage capacity, it only changes the number of web servers. Autoscaling makes its decisions based on rules that specify the threshold for a metric, and triggers an autoscale event (change number of web servers).

Autoscaling is integrated with Azure Monitor. This means we can configure Azure Monitor to send out notifications when scaling events are triggered.
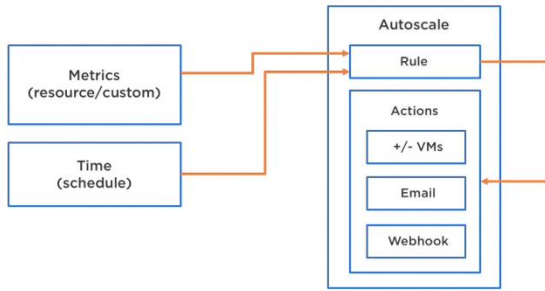
Important: when adding a rule to scale out based on a condition, remember to add a corresponding rule to scale back in when the condition is over.

Scheduled Autoscaling

Scale to a specific instance count according to a schedule.
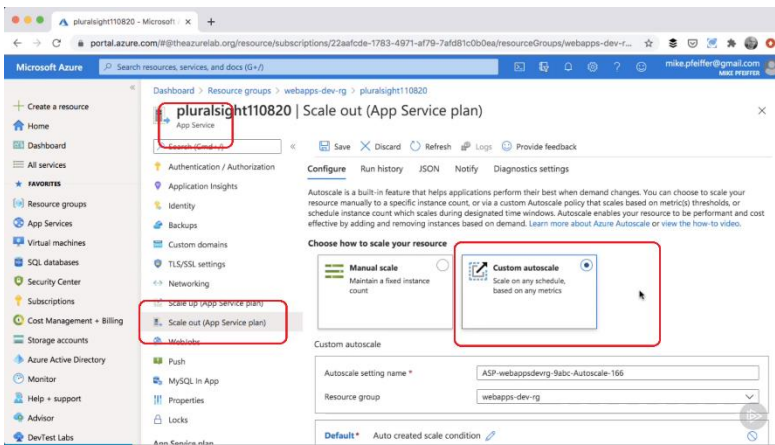
Go to App Service > Scale out (App Service plan) > Custom autoscale > +Add a scale condition
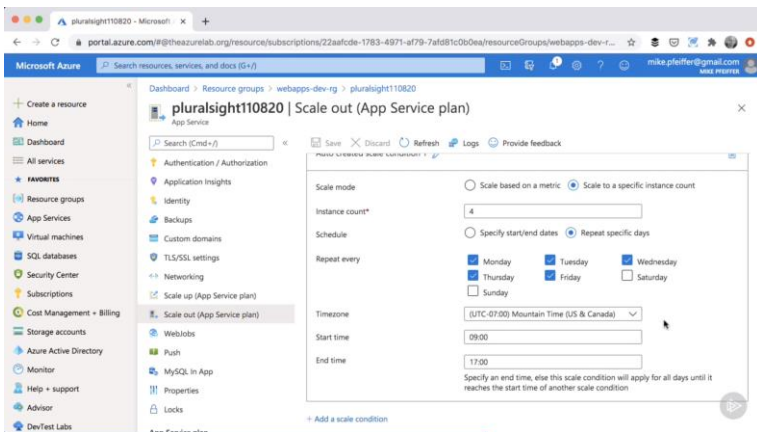
Scaling by Operational Metrics or System Metrics

As before with scheduled autoscaling, but this time when you "+Add a scale condition", choose the "Scale based on a metric" radio button.
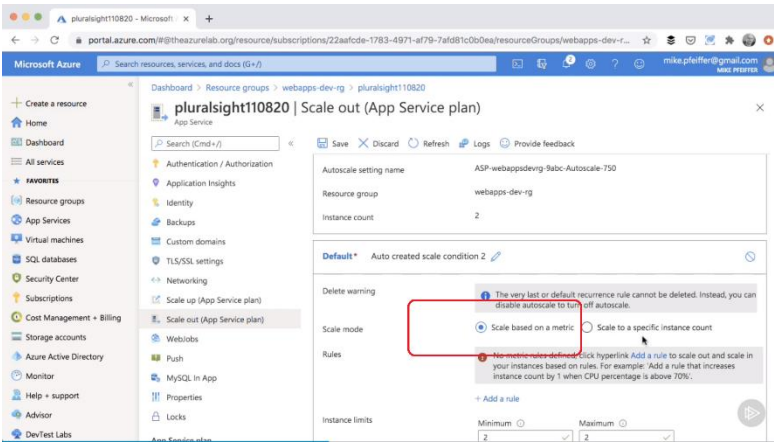
For example, the metric could be CPU percentage
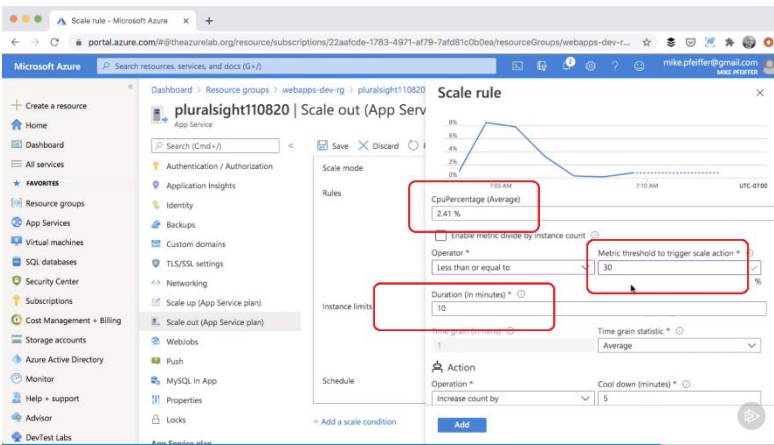
Setting the duration to 10 minutes is done so that it doesn't scale immediate for an anomaly such as a momentary spike in usage. We don't want it to scale unnecessarily.
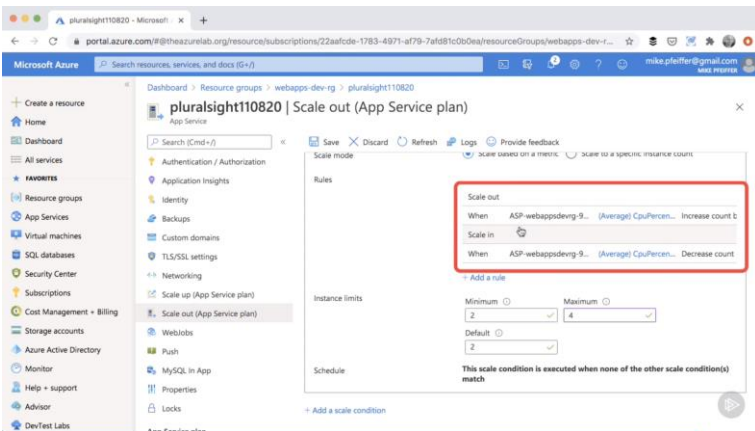
We need to make sure we have both a scale out rule and a scale in rule to return things back to normal.

A custom implementation would collect operational and system metrics, analyze the metrics, and then scale resources accordingly.

Scale based on a metric, such as the length of the disk queue, or the number of HTTP requests awaiting processing

System metrics are measurement types found in the system.

1.3 Implement Azure functions

- create and deploy Azure Function apps
- implement input and output bindings for a function
- implement function triggers by using data operations, timers, and webhooks
- implement Azure Durable Functions
- implement custom handlers

https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json

https://docs.microsoft.com/en-us/learn/paths/create-serverless-applications/

https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages?tabs=csharp
(link to Azure Functions error handling and retries)

Azure Functions allow you to host business logic that can be executed without managing or provisioning infrastructure. With traditional enterprise architecture, you need to consider server hardware infrastructure provisioning and maintenance up front. With serverless computing, this is delegated to the cloud provider letting you focus completely on building the app logic.

Every Azure function has its own *function.json* file, which defines the trigger and provides configuration information such as bindings. Functions have one trigger, and any number of bindings.

The *host.json* metadata file contains global configuration options that affect all functions for a function app.

Create and deploy Azure Function apps
Skill: create and deploy Azure Function apps

https://docs.microsoft.com/en-us/azure/azure-functions/functions-core-tools-reference

https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local

https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference

https://docs.microsoft.com/en-us/learn/paths/create-serverless-applications/

https://docs.microsoft.com/en-us/learn/modules/create-serverless-logic-with-azure-functions/

https://docs.microsoft.com/en-us/learn/modules/create-serverless-logic-with-azure-functions/3-create-an-azure-functions-app-in-the-azure-portal

https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-your-first-function-visual-studio

https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale

Functions are hosted in an execution context called a *function app*. Function apps are used to logically group Azure Functions and link them to a hosting plan and a storage account.

Hosting plan options: app service plan, consumption plan, premium plan, dedicated plan, ASE, Kubernetes.
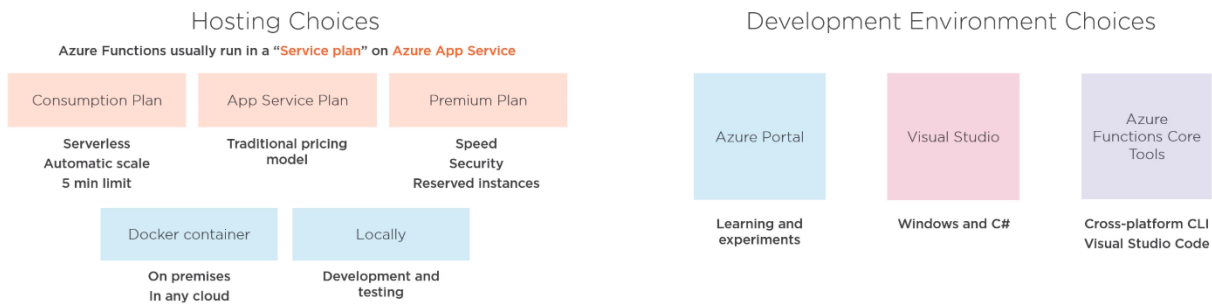
To create a Function App from the Azure Portal:

Create a resource > Compute > Function App

Create Function App page > Enter basic settings > Review + create > Create

From Azure CLI:

```
az functionapp create -g MyResourceGrp -p MyAppServicePlan -n MyFuncAppName -s MyStorageAcct
```

After the Function App has been created, you can publish Azure Functions to it.

From VSCode and command line, there is now a Func command. First install Azure func core tools:

```
npm i azure-functions-core-tools
```

Initialize the function app:

```
func init MyFunctionProject
```

This will create a folder named MyFunctionProject and will initialize the project `MyFunctionApp.csproj`. You will be prompted for initialization parameters such as runtime (dotnet, node, python, …)

Then change directory and create the function:

```
cd MyFunctionProject
func new
```

You will be prompted for the function creation parameters such as template (http trigger, timer, …), function name.

Input/Output Bindings
Skill: implement input and output bindings for a function

https://docs.microsoft.com/en-us/learn/modules/chain-azure-functions-data-using-bindings/

Bindings provide a declarative way to connect to data from within your code. The *type* of binding defines where we are reading or sending data. A binding type can be used as an input, an output or both. However not all types support both input and output. An Azure Function can have multiple input and output binding.
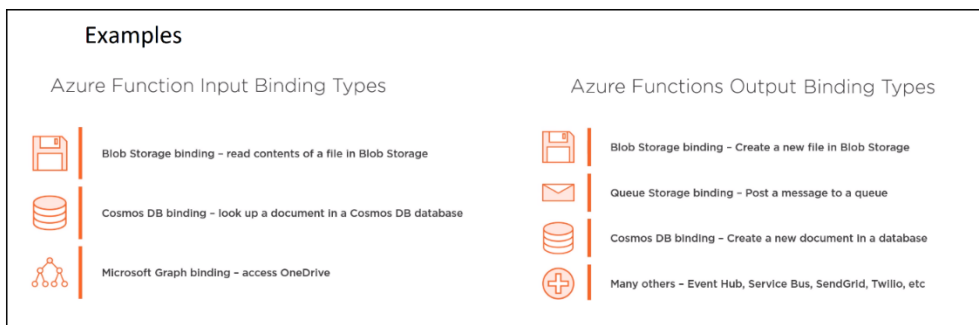
Common binding properties: Name, Type, Direction.

Bindings are defined in JSON. A binding is configured in your function's configuration file, which is named *function.json* and lives in the same folder as your function code. The Name property is how your code accesses the binding's value. For example:

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "name": "Request",
    },
    {
      "type": "http",
      "direction": "out",
      "name": "Response"
    }
  ]
}
```

Triggers
Skill: implement function triggers by using data operations, timers, and webhooks

A trigger is an object that defines how an Azure Function is invoked. Every function must have exactly one trigger associated with it.

Data Operations

Blob, Queue, Cosmos DB and Event Hub trigger types are examples of data operation triggers. For example, a blob trigger is a trigger that executes a function when a file is uploaded or updated in Azure Blob storage. Blob triggers have a Path setting. For example

```
samples-workitems/{name}
```

The first part, samples-workitems, represents the blob container that the trigger monitors. The second part, {name} means that every type of file will cause the trigger to invoke the function. The name represents a parameter in your Azure function that receives the name of the added file.

Timers

A timer trigger is a trigger that executes a function at a consistent interval. Timer triggers can be used to implement scheduled tasks. You supply two pieces of information: timestamp parameter name, a schedule (CRON expression). A CRON expression consists of:

```
{second} {minute} {hour} {day} {month} {day of the week}
```

For example, a CRON expression to create a trigger that executes every five minutes looks like:

```
0 */5 * * * *
```

CRON expression special characters:

- Asterisk * means select every value in a field
- Comma , separates items in a list
- Hyphen – specifies a range
- Slash / specifies an increment

Webhooks – HTTP Request Triggers

HTTP triggers can be used to respond to webhooks. You supply a webhook callback to a third-party app. The callback will be used to call your HTTP trigger Azure Function.

Note: In addition to responding to webhooks, HTTP triggers can also be used to implement APIs (out of scope).

HTTP triggers let you customize the HTTP method (GET, POST), and the Route to control the URL of your Function. For example:

```json
{
    "bindings": [
        {
            "type": "httpTrigger",
            "name": "req",
            "direction": "in",
            "methods": [ "get" ],
            "route": "products/{category:alpha}/{id:int?}"
        },
        {
            "type": "http",
            "name": "res",
            "direction": "out"
        }
    ]
}
```

HTTP triggers can be secured by requiring an authorization key with one of three authorization levels:

- Anonymous: no key required
- Function: key per function
- Admin: key per function app

Example:

```csharp
[FunctionName("HttpTriggerCSharp")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)] HttpRequest req, ILogger log)
{
    string name = req.Query["name"];

    string requestBody = String.Empty;
    using (StreamReader streamReader =  new  StreamReader(req.Body))
    {
        requestBody = await streamReader.ReadToEndAsync();
    }
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    return name;
}
```

Durable Functions
Skill: implement Azure Durable Functions

https://docs.microsoft.com/en-us/learn/modules/create-long-running-serverless-workflow-with-durable-functions/

Durable Functions enables you to implement stateful function workflows in a serverless-environment. These stateful workflows are called orchestrations, and are defined using an orchestration function.

The programming model for durable functions consists of three function types: Client, Orchestrator, and Activity.

- *Client functions* are the entry point. When triggered, they initiate a new orchestration.
- *Orchestrator functions* describe how actions are executed, and the order in which they are run.
- *Activity functions* are the basic units of work in a durable function orchestration.

Durable functions implement many common workflow patterns including function chaining, fan out/fan in, async HTTP API, Monitor, Human Interaction.

Chaining

F1   F2   F3   F4

Chaining orchestrates each of the desired functions in order to have the output of one function act as the input of the next.

Fan out/fan in
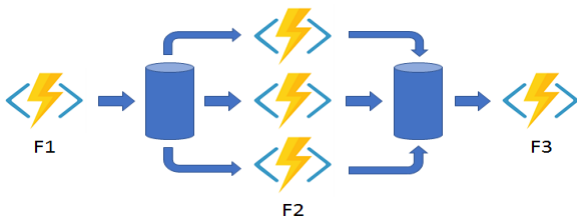
Fan out/in has a single orchestrator function execute multiple functions to execute in parallel (fan out), and then wait until they are all complete, and aggregate the results together (fan in).

Async HTTP APIs
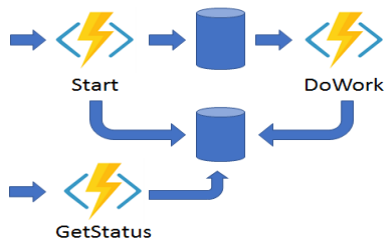
Async HTTP is when the API addresses the problem of coordinating the state of long-running operations with external clients, by redirecting the external clients to an endpoint where they can use polling to learn when the operation is finished.

Monitor
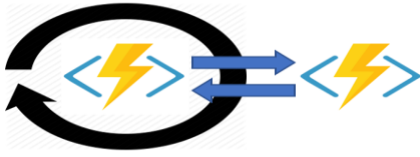
Monitor refers to a flexible, recurring process in a workflow that checks to see when something is completed. An example is polling until specific conditions are met.

Human interaction (requiring manual intervention)

Human interaction is when durable functions are used to trigger a process to notify a party and require their intervention for the process to proceed. If no response is received within a timeframe, then other rules can be applied.

Aggregator (Stateful Entities)
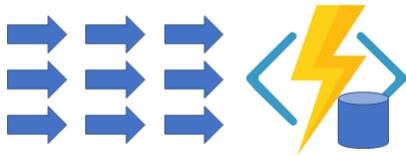
Aggregator is when the API aggregates event data over a period of time into a single, addressable entity.

Implement custom handlers
Skill: implement custom handlers

https://docs.microsoft.com/en-us/azure/azure-functions/functions-custom-handlers

https://azure.microsoft.com/en-ca/updates/azure-functions-custom-handlers-are-now-generally-available/

https://www.youtube.com/watch?v=-2mAFSNCP-I

https://www.youtube.com/watch?v=RPCEH247twU

Every Functions app is executed by a language-specific handler. While Azure Functions supports many language handlers by default, there are cases where you may want to use other languages or runtimes. Custom handlers are lightweight web servers that receive events from the Functions host. Any language that supports HTTP primitives can implement a custom handler.
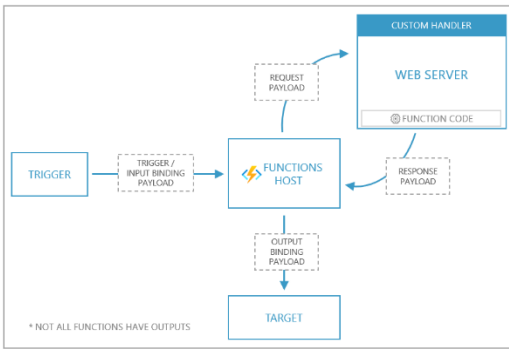
An Azure Function implemented as a custom handler must configure the *host.json*, *local.settings.json*, and *function.json* files. There is also a *proxies.json* file.

A custom handler is defined by configuring the host.json file with details on how to run the web server via the customHandler section. Use the arguments array to pass any arguments to the executable. Arguments support expansion of environment variables (application settings) using %% notation.

Sample host.json custom handler section:

```json
{
  "version": "2.0",
  "customHandler": {
    "description": {
      "defaultExecutablePath": "app/handler.exe",
      "arguments": [
        "--database-connection-string",
        "%DATABASE_CONNECTION_STRING%"
      ],
      "workingDirectory": "app"
    }
  }
}
```

defaultExecutablePath   The executable to start as the custom handler process.

workingDirectory   The working directory in which to start the custom handler process.

arguments   An array of command line arguments to pass to the custom handler process.

local.settings.json defines application settings used when running the function locally. As it may contain secrets, local.settings.json should be excluded from source control. In Azure, use application settings instead. For custom handlers, set FUNCTIONS_WORKER_RUNTIME to Custom in local.settings.json.

Sample local.settings.json:

```json
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "Custom"
  }
}
```

When used with a custom handler, the function.json contents are no different from how you would define a function under any other context. The only requirement is that function.json files must be in a folder named to match the function name.

Request Payload the Functions host sends an HTTP post request to the custom handler with a payload in the body. The payload includes a JSON structure with two members: Data and Metadata. The Data member includes keys that match input and trigger names as defined in the bindings array in the function.json file. The Metadata is generated from the event source.

Example:

```
{
  "Data": {
    "myQueueItem": "{ message: \"Message sent\" }"
  },
  "Metadata": {
    "DequeueCount": 1,
    "ExpirationTime": "2019-10-16T17:58:31+00:00"
  }
}
```

Response Payload The function responses are formatted as key/value pairs. Supported keys include:

Outputs	response values as defined by the bindings array in function.json
Logs	the Function's invocation log
ReturnValues	configured as $return in the function.json file

Example:

```
{
  "Outputs": {"res": "Message enqueued"},
  "Logs": ["Log message 1", "Log message 2"],
  "ReturnValue": "{\"hello\":\"world\"}"
}
```

2. Develop for Azure storage (15-20%)

https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview

https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy

https://docs.microsoft.com/en-gb/learn/modules/explore-azure-blob-storage/5-azure-storage-redundancy

A storage account is a cloud repository for blobs, files, queues, tables.

> FYI: Storage accounts can also be used to contain unmanaged disks. Unmanaged disks are created and managed by you. Azure Managed Disks are managed by Microsoft Azure. Compared to unmanaged disks, Azure-managed disks provide better scalability for VMs with VM scale sets, and breaks the limit of 20,000 IOPS per storage account, which will impact the number of VMs that can be created per storage account. Azure Managed Disks are recommended by Microsoft. OUT OF SCOPE

Storage accounts have 4+2 storage redundancy options:

- Local (LRS) – multiple copies of data in one data center made synchronously
- Zone (ZRS) – multiple copies in multiple data centers within a region
- Geo (GRS) – multiple copies in multiple regions
    - 3 copies in *one data center* within the primary region using LRS
    - 3 async copies in secondary region using LRS
- Geo-zone (GZRS) – multiple copies in multiple regions
    - 3 copies *in separate data centers* within the primary region using ZRS
    - 3 async copies in secondary region using LRS
- Read-Access Geo (RA-GRS) – same as GRS but with ability to read from replicas
- Read-Access Geo-zone (RA-GZRS) – same as GZRS but with ability to read from replicas

note: zone = data center = availability zone
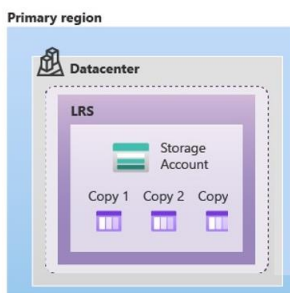
LRS: ZRS:
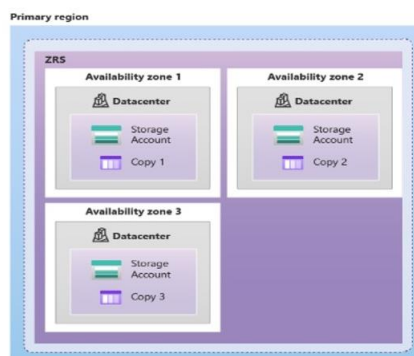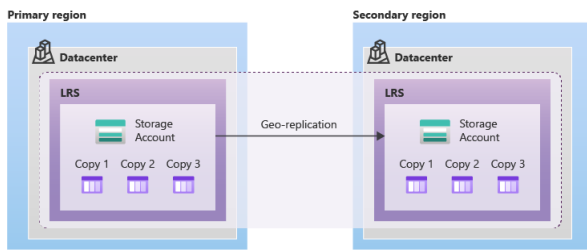
GRS: GZRS:
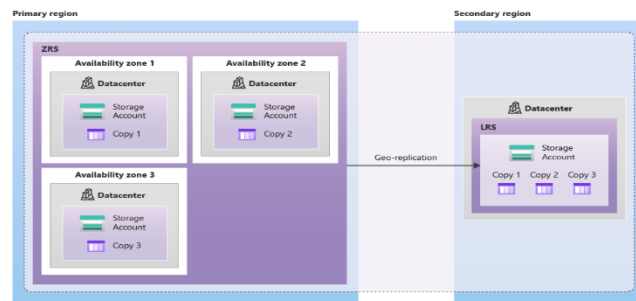
The hot and cool tiers support all redundancy options. The archive tier supports only LRS, GRS, and RA-GRS.

2.1 Develop solutions that use Cosmos DB storage

- select the appropriate API and SDK for your solution
- implement partitioning schemes and partition keys
- set the appropriate consistency level for operations
- perform operations on data and Cosmos DB containers
- manage change feed notification

https://docs.microsoft.com/en-us/learn/paths/az-204-develop-solutions-that-use-azure-cosmos-db/

Azure Cosmos DB is an enterprise grade non-relational database. At the lowest level, all data in any Azure Cosmos DB is stored in an ARS (atom-record-sequence) format, allowing Cosmos DB to achieve multi-modal database functionality.

An *Azure Cosmos DB account* contains a unique DNS name. It is the fundamental unit of global distribution and high availability. It connects your usage to your Azure subscription for billing purposes. An Azure Cosmos DB account is a container for one or more databases which are in turn containers for one or more collections. A collection contains documents. A document is an unstructured set of key/value pairs, read and written in JSON format.

The hierarchy of different entities:
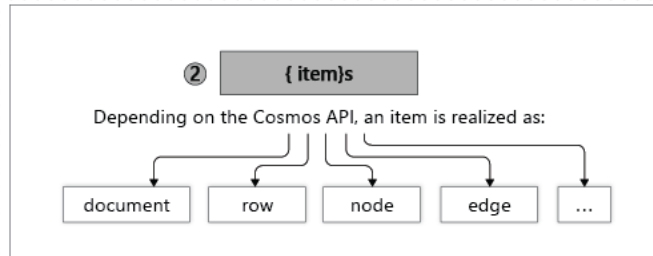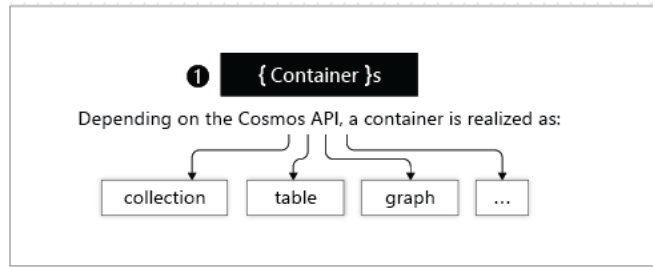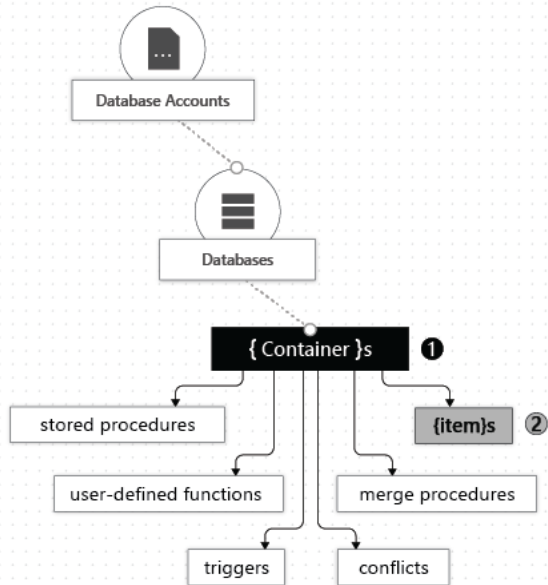
Comparing Azure Cosmos DB to SQL server:

Azure Cosmos DB		SQL Server analogy
cosmos db account	is like	sql server host server
database	is like	database
collection (aka container)	is like	table
document	is like	record
unstructured key/value pair	is like	column value (attribute) of a record

Example: create a Cosmos DB account using Azure CLI.

```
# create a cosmos db account (defaults to sql api)
az cosmosdb create --name pluralsight --resource-group pluralsight

# create a sql database
az cosmosdb sql database create --account-name pluralsight
    --name sampledb

# create a sql database container
az cosmosdb sql container create --resource-group pluralsight
    --account-name pluralsight --database-name sampledb
    --name samplecontainer --partition-key-path "/employeeid"
```

Select Appropriate API and SDK
Skill: select the appropriate API and SDK for your solution

https://docs.microsoft.com/en-us/learn/modules/choose-api-for-cosmos-db/

API

The API is chosen when creating the Cosmos DB account and determines the type of account to create. Azure Cosmos DB provides five main APIs (plus etcd) to suit the needs of your application, each of which currently requires a separate account:

SQL (Core)	relational	the **default** API. You can query the hierarchical JSON documents with a SQL-like language. The SQL API is the preferred choice for JSON documents.
MongoDB	document	allows existing MongoDB SDK clients to interact with the data as if it was an actual MongoDB database. MongoDB supports JSON documents.
Cassandra	SQL-like	Allows you to query using the Cassandra Query Language (CQL). Not relational (no joins).
Azure Table	NoSQL	provides support for applications that are written for Azure Table Storage. Querying is by OData and LINQ queries in code, and the original REST API for GET operations.
Gremlin	graph	provides a graph-based view over the data. You typically use a traversal language to query a graph database, and Azure Cosmos DB supports the Gremlin language (Apache Tinker pop).
Etcd	*Kubernetes*	*Etcd API allows you to access the key-value items in the Kubernetes Etcd configuration folder*

SQL API is the preferred data model if you are creating a new application. If you're working with graphs or tables, or migrating your MongoDB or Cassandra data to Azure, select relevant data models:

Scenario	**SQL (Core)**	**MongoDB**	**Cassandra**	**Azure Table**	**Gremlin**
New projects being created from scratch	✔				
Existing MongoDB, Cassandra, Azure Table, or Gremlin data		✔	✔	✔	✔
Analysis of the relationships between data					✔
All other scenarios	✔				

SDK

Data is stored in JSON documents in Azure Cosmos DB. Documents can be created, retrieved, replaced, or deleted in the portal, or programmatically. Azure Cosmos DB provides client-side SDKs for .NET, .NET Core, Java, Node.js, and Python, each of which supports these operations.

Selecting an SDK:

- **SQL API** utilize the latest Cosmos DB SDK for your platform
- **MongoDB, Cassandra, and Gremlin** use current SDK's for those API's
- **Azure Table API** leverage the current Table Storage SDK

.NET starts with the Microsoft.Azure.DocumentDB NuGet package. This is the client library (SDK) for Azure Cosmos DB DocumentDB (SQL) API. In code:

```
using Microsoft.Azure.Documents.Client;

string endpoint = "https://whatever";  // from portal – put in appsettings.json
string key = "qwertyuiop1234567890";  // from portal – put in appsettings.json
DocumentClient dc = new DocumentClient(new Uri(endpoint), key);
var uri = UriFactory("my-database-name", "my-collection-name");
var result = dc.CreateDocumentCollectionUri(uri, partitionkeyValue)
        .GetAwaiter()
        .GetResult();
var query = dc.CreateDocumentQuery<EmployeeModel>(uri, myOptions)
        .Where(e => e.LastName == "Jack");
foreach (var employee in query)
{
        Console.WriteLine(employee);
}
```

Partitioning Keys and Schemes
Skill: implement partitioning schemes and partition keys

https://docs.microsoft.com/en-us/learn/modules/monitor-and-scale-cosmos-db/

https://docs.microsoft.com/en-us/learn/modules/create-cosmos-db-for-scale/4-how-to-choose-a-partition-key

A partition key is a *document* property that defines the partition strategy. It's set when you create a container and can't be changed.

Understand the following key metrics:

- **Total requests** made per second
- **Total throughput**, the rate at which data is processed measured in Request Units per second (RU/s)
- **Total storage**, measured in kilobytes (KB)

An RU is the amount of CPU, disk I/O, and memory required to read 1 KB of data in 1 second. The cost to do a point read, which is fetching a single item by its ID and partition key value, for a 1 KB item is 1 RU. When you create an Azure Cosmos DB collection, you configure a fixed maximum number of RUs. The sum of RUs that all of the operations consume in the collection must be less than this value. If you exceed this value, requests to the database are throttled.

The partition key is used to automatically partition data among multiple servers for scalability. Choose a JSON property name that has a wide range of values and is likely to have evenly distributed access patterns. Partitioning is the distribution and grouping of your data across the underlying resources. Documents are grouped in a partition based on the value of the partition key. You specify the partition key when you create the collection. An effective partitioning strategy distributes data and access evenly across partitions. Querying documents from within the same partition is less expensive than querying across partitions.

A partitioning strategy enables you to add more partitions to your database when you need them. This scaling strategy is called scale out or horizontal scaling, and is defined by a partition key. A partition key is the value by which Azure organizes your data into logical divisions and serves as the means of routing your request to the correct partition. It should be a value that does not change for the item, and it should have many different values represented (high cardinality) in the container.

Partition key best choice:

- Have a high cardinality. This allows data to distribute evenly across all physical partitions.
- Evenly distribute requests. Keeps the RU/s is evenly divided across all physical partitions.
- Evenly distribute storage. Each partition can grow up to 20 GB in size.

Logical partitions: A logical partition is a set of items within a container that share the same partition key. You can divide a container into pieces based on this partition key criteria. Each of these pieces is a logical partition. All items stored in a logical partition share the same partition key. A container can have as many logical partitions as it needs, but each partition is limited to 20GB of storage. Logical partitions are managed by Cosmos DB, but their use is governed by your partition key strategy.

Physical partitions: a group of replicas of your data that is physically stored on the servers. A physical partition can contain one or more logical partitions. Physical partitions are managed entirely by Azure Cosmos DB.

Replica Set: A physical partition contains multiple replicas of the data, known as a replica set. By having this data replicated, you enable your storage to be durable and fault tolerant. Replica sets are managed entirely by Azure Cosmos DB.

Avoid hot partitions. A hot partition is accessed more than the other partitions. A partition key design that doesn't evenly distribute throughput requests can create hot partitions, which results in inefficient use of the total configured throughput. A good partition design avoids hot partitions.

Other strategy considerations:

- Throughput is distributed evenly across all of your physical partitions
- Multi-item transactions require triggers or stored procedures
- You will want to minimize cross-partition queries for heavier workloads
- Decide upon a partition key strategy before creating your container

The three indexing modes you can use with Azure Cosmos DB are:

- **Consistent**: The index is updated synchronously every time a new document is written to the collection. New queries on the collection use the updated index immediately. Query results are consistent with the updated documents in the collection.
- **Lazy**: The index is updated at a lower priority. The reads and writes from the collection take a higher priority. In lazy mode, writes are cheaper because the index isn't updated immediately. When the index is fully updated depends on the demands on the collection. Query results don't include the updated documents until the index is consistent with the collection.
- **None**: No index is created. Queries are expensive on collections that aren't indexed. If you're using your Azure Cosmos DB collection to read records directly rather than querying the collection, it's possible to avoid the overhead of indexing.

Skill: set the appropriate consistency level for operations

"Distributed databases that rely on replication for high availability, low latency, or both, must make a fundamental tradeoff between the read consistency, availability, latency, and throughput."
– Microsoft Cosmos DB Documentation

Azure Cosmos DB has five different consistency levels: strong, bounded staleness, session, consistent prefix, and eventual.
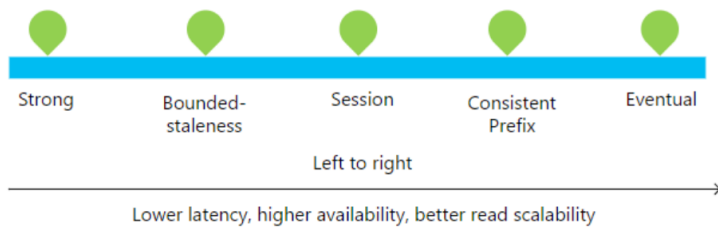
Consistency Level	Guarantees
Strong	Guarantees that reads get the most recent version of an item. Linearizability.
Bounded Staleness	Guarantees that a read has a max lag (either versions or time). Reads lag behind writes by at most k prefixes or t interval.
Session	Guarantees that a client session will read its own writes. Session consistency is scoped to a client session, and provides predictable consistency for that session.
Consistent Prefix	Guarantees that updates are returned in order. This level guarantees that you always read data in the same order that you wrote the data, but there's no guarantee that you can read all the data. If writes were performed in the order A, B, C, then a client sees either A, A,B, or A,B,C, but never out of order like A,C or B,A,C.
Eventual	Provides no guarantee for order. Out of order reads. Use this consistency level if the order of the data is not essential for your application.

Cosmos DB containers
An Azure Cosmos container is the fundamental unit of scalability. A container is specialized into API-specific entities. An item can represent either a document in a collection, a row in a table, or a node or edge in a graph.

	SQL API	Cassandra API	MongoDB API	Gremlin API	Table API
container	Container	Table	Collection	Graph	Table
item	Item	Row	Document	Node or edge	Item

A container is a schema-agnostic container of items. Items in a container can have arbitrary schemas. By default, all items that you add to a container are automatically indexed without requiring explicit index or schema management.

Perform operations on data and Cosmos DB containers
Skill: perform operations on data and Cosmos DB containers

https://docs.microsoft.com/en-us/learn/modules/access-data-with-cosmos-db-and-sql-api/

Azure Cosmos DB uses SQL queries, just like SQL Server, to perform query operations. The returned results would be a JSON document. The source can also be reduced to a smaller subset. For instance, to enumerate only a subtree in each document, the subroot could then become the source, as shown in the following example:

```
SELECT *
  FROM Products.shipping
```

Any defined valid JSON value that can be found in the source is considered for inclusion in the result of the query. For example:

```
SELECT *
  FROM Products.shipping.weight
```

The JOIN clause lets you perform inner joins with the document and the document subroots.

```
SELECT p.productId
  FROM Products p
  JOIN p.shipping
```

the product IDs are returned for each product that has a shipping method. If you added a third product that didn't have a shipping property, the result would be the same because the third item would be excluded for not having a shipping property.

Geospatial queries enable you to perform spatial queries using GeoJSON Points, allowing you to work with a Point, Polygon, or LineString.

Stored procedures are written in JavaScript and are stored in a container on Azure Cosmos DB. Stored procedures are the only way to achieve atomic transactions within Azure Cosmos DB; the client-side SDKs do not support transactions. UDFs can be called only from inside queries.

Manage change feed notification
Skill: manage change feed notification

Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur. While all other server-side programming approaches enable execution on the Cosmos DB engine, change feed processing enables you to react to data changes using server-side code outside of the Cosmos DB engine.

Change feed enables you to be notified for any insert and update on your data. Deletes are not directly supported, but you can leverage a soft-delete flag. A change will appear exactly once in the change feed. Reading data from the database will consume throughput. Partition updates will be in order, but between partitions there is no guarantee. Change feed is not supported for the Azure Table API.

You can trigger an Azure Function with a change feed and start a microservice to process the change.
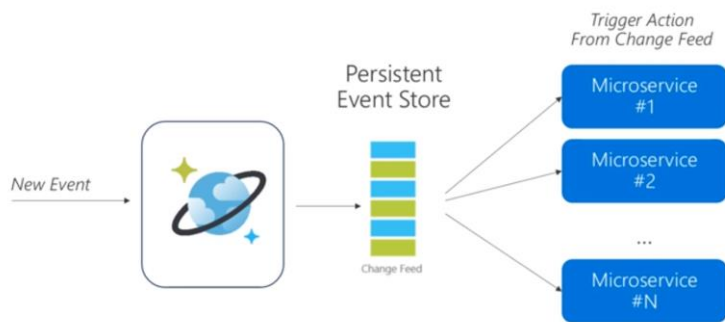
Another use-case for change feed is for creating a materialized view.
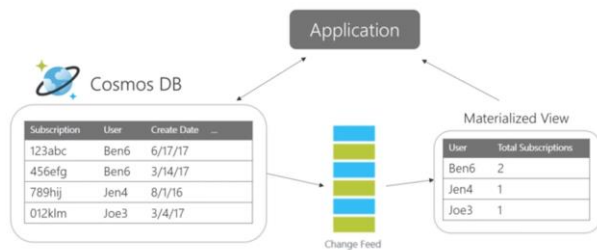
Another use-case for change feed is to replicate data.

Replicating Data
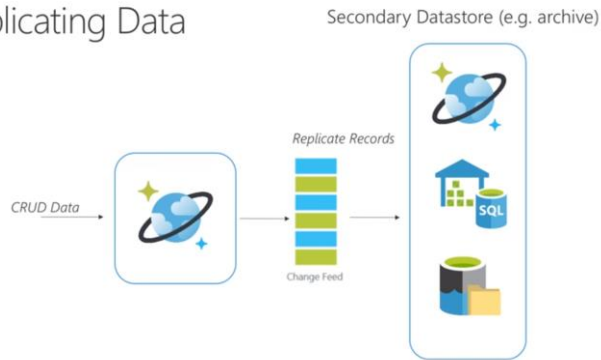
The change feed notification is a design pattern that triggers a notification or a call to an API, when an item is inserted or updated. You can selectively trigger a notification or send a call to an API based on specific criteria. While the Azure Function code would execute during each write and update, the notification would only be sent if specific criteria had been met.

Note: change feed support is also available in Azure *blob* storage. The purpose of blob storage change feed is to provide transaction logs of all the changes to the blobs and the blob metadata. It provides an ordered, read-only log of these changes. Client applications can read these logs enabling you to build solutions that process change events occurring in your Blob Storage account at a low cost.

2.2 Develop solutions that use blob storage

- move items in Blob storage between storage accounts or containers
- set and retrieve properties and metadata
- perform operations on data by using the appropriate SDK
- implement storage policies, data archiving, and retention

Blobs are files for the cloud. Unlike a local file, blobs can be reached from anywhere with an internet connection via REST API over HTTP/HTTPS. Blobs store unstructured data: text files (text files, logs) and binary files (images, virtual disks). Blob storage is accessible by http or https protocol, not SMB or NFS like traditional file servers.

There are three kinds of blobs:

- block blobs composed of blocks of different sizes
- append blobs support only appending new data
- page blobs designed for random access reads and writes (512K pages)
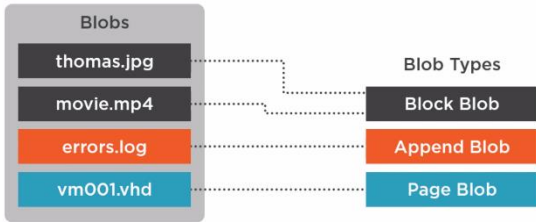
Blobs are stored in blob containers, which in turn are stored in storage accounts.
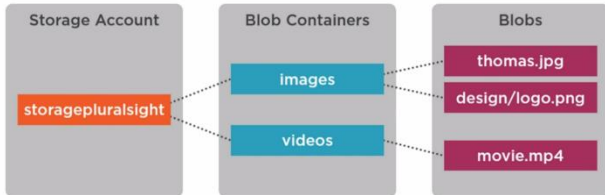
There are three kinds of standard performance storage accounts:
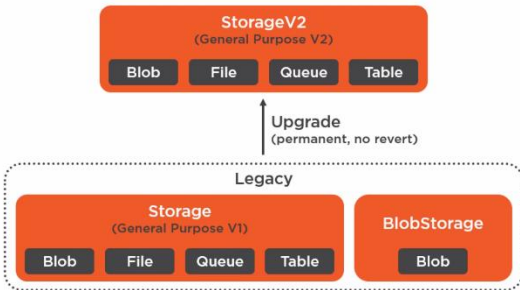
Omitting the legacy storage account kinds, we have the following standard and premium storage account kinds:
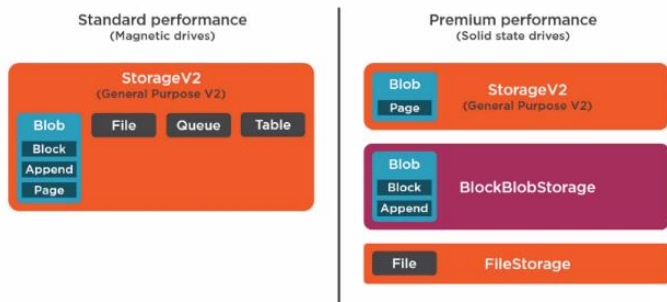
Move Blob storage items between storage accounts or containers
Skill: move items in Blob storage between storage accounts or containers

There are several tools that you can use for performing these tasks: Azure Storage Explorer, AzCopy – a command-line tool for performing bulk copy operations, Python, SSIS – the SQL Server Integration Service Feature Pack for Azure can transfer data.

However, none of these options provide a move operation. If you need to move blobs or containers between different locations, you need to perform a <u>copy</u>, and then <u>delete</u> the source.

Azure CLI

To copy blobs with the Azure CLI, use the `az storage blob copy` command.

```
#copy a single blob                               #copy container with all its blobs
az storage blob copy start                        az storage blob copy start-batch
    --source-account-name   acct1                     --source-account-name   acct1
    --source-account-key    00000000                  --source-account-key    00000000
    --source-container      images                    --source-container      images
    --source-blob           pic.jpg
    --account-name          acct2                     --account-name          acct2
    --account-key           00000000                  --account-key           00000000
    --destination-container pics                      --destination-container pictures
    --destination-blob      pic.jpg
```

AzCopy Command-line Utility

The AzCopy command line utility has a copy command that allows you to copy between containers, storage accounts, and local directories. If you copy a directory instead of a single blob, you additionally specify the recursive flag set to true `--recursive=true`.

```
azcopy copy "<source-path>" "<target-path>" [--recursive=true]
```

To upload a local folder, you specify the local directory as the source, and the URL to a container, including its SAS, as the target.

```
azcopy copy "C:\Documents" "https://mystorageurl.com/mycontainer?[SAS]" –recursive=true
```

When you specify the source directory like this "C:\Documents", azcopy prefixes the blobs with the directory name "Documents". If instead you use a star like this "C:\Documents*", azcopy will not prefix the blobs with a directory name.

Copy a single blob between storage accounts by specifying both source and target urls.

```
azcopy copy "https://url1.com/mycontainer/myfile.txt?[SAS]" "https://url2.com/mycontainer/myfile.txt?[SAS]"
```

Copy all blobs from one container to another by using two container URLs.

```
azcopy copy "https://url1.com/mycontainer1?[SAS]" "https://url2.com/mycontainer2?[SAS]" –recursive=true
```

Copy all containers from one storage account to another by using two storage account URLs.

```
azcopy copy "https://url1.com?[SAS]" "https://url2.com?[SAS]" –recursive=true
```

You can also use the .NET client library to copy blobs, but that means you have to write code.

Set and retrieve properties and metadata
Skill: set and retrieve properties and metadata
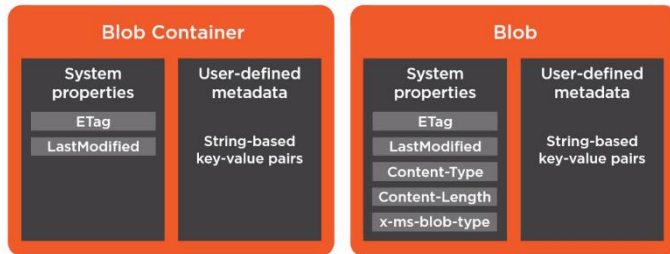
System properties are information that the storage service adds to storage resources, for example LastModified. Blobs and containers also support user-defined metadata in the form of key-value string pairs. Your apps can use metadata for anything you like.

From Azure Portal, under Container/Settings, you can select Properties or Metadata. Under the Overview of a specific blob, you can access Properties. If you scroll down, you can access Metadata.

If you point your browser at a blob, hit F12 > Network tab > Headers, you will see the properties as headers. These properties use standard HTTP header names. You can also see metadata as HTTP headers prefixed with x-ms-meta-
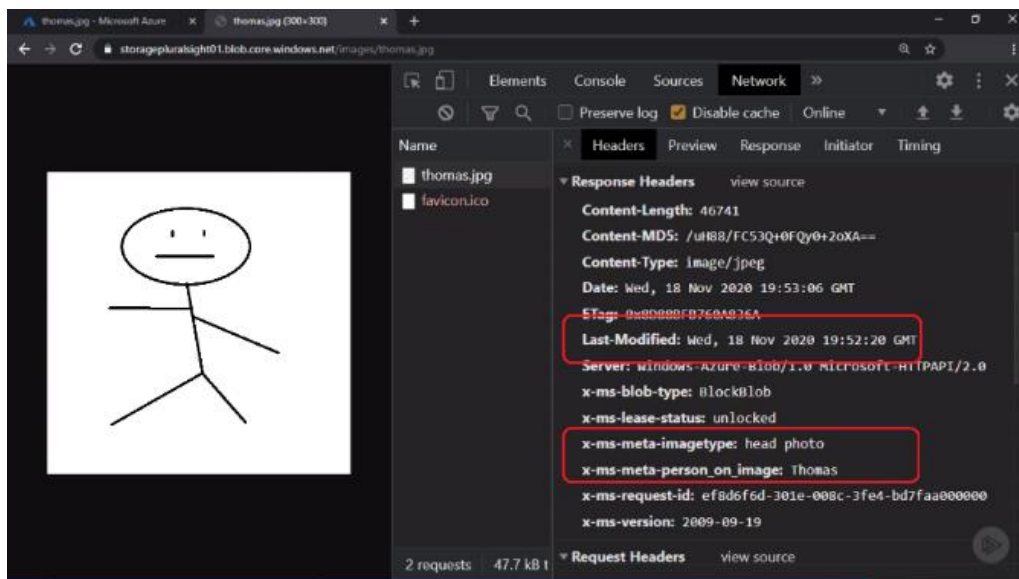
Interact with data using the appropriate SDK
Skill: perform operations on data by using the appropriate SDK

https://docs.microsoft.com/en-us/dotnet/api/overview/azure/Storage.Blobs-readme

The Blob storage API is REST-based and supported by client libraries in many popular languages including .NET, JavaScript, and PHP. Let's look at .NET

It lets you write apps that create and delete blobs and containers, upload and download blob data, and list the blobs in a container.
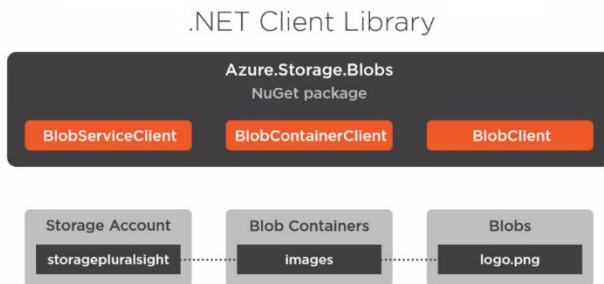
You can set and retrieve properties and metadata in .NET. You need to reference the NuGet package Azure.Storage.Blobs and then, in code, you add the ==connection string of your storage account==. Then you create a BlobClient.

```
BlobServiceClient blobServiceClient =
    new BlobServiceClient(_connectionString); // note: blob service == storage account
BlobContainerClient blobContainerClient =
    blobServiceClient.GetBlobContainerClient(_blobContainerName);
BlobClient blobClient = blobContainerClient.GetBlobClient(_blobName);
```

Using the blob client, you get properties and metadata

```
BlobProperties blobProperties = await blobClient.GetPropertiesAsync();
```

and you set properties and metadata.

```
await blobClient.SetHttpHeadersAsync(blobHttpHeaders);
```

```
await blobClient.SetMetadataAsync(metadata);
```

Important: properties that are not set are deleted, so get a properties object with all the properties, change the ones you want, and then set them all.

You can also create and delete containers using the blob container object methods CreateIfNotExistsAsync() and DeleteIfExistsAsync().

When several users or processes simultaneously access the same storage account, Azure provides a leasing mechanism to handle contention. When you acquire a lease to a blob or container, you get exclusive write and delete access to that blob or container.

Data archiving and retention – hot, cool, and archive storage

Azure provides three different access tiers with different price and performance: hot, cool, archive.

- **Hot**: frequently accessed data.
- **Cool**: infrequently accessed. Stored for at least 30 days.
- **Archive**: rarely accessed and stored for at least 180 days. Available for blobs only – cannot configure a Storage Account with this tier.

Cooler tiers have lower storage costs, but higher access costs. Archive tier blobs are stored offline and must be rehydrated to access them – rehydrating can take up to 15 hours with standard rehydration, or one hour with high priority rehydration ($$$).

A default online access tier of Hot or Cool can be set for the storage account. The default is Hot. The archive access tier can only be set at the blob level. Block blob access Tier can be set to Hot, Cold, or Archive.
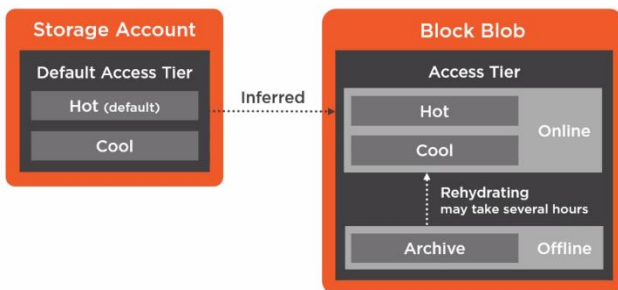
Access Tiers are only available to Storage-V2 and (Legacy) BlobStorage accounts.

Access Tiers for Block Blobs
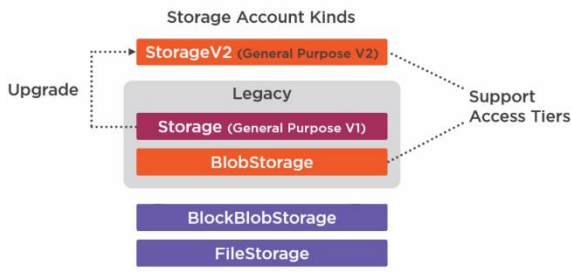
From the Azure portal, you can set the access tier for the storage account by going to the storage account > Settings > Configuration.
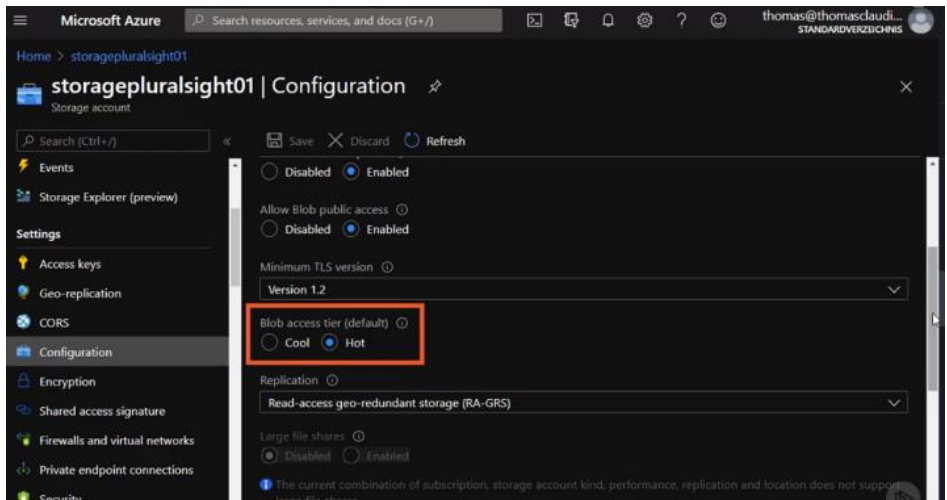
From Azure portal, you can set the access tier for a blob by going to the blob container and then selecting the blob. You can then change the tier to Hot, Cool, Archive.
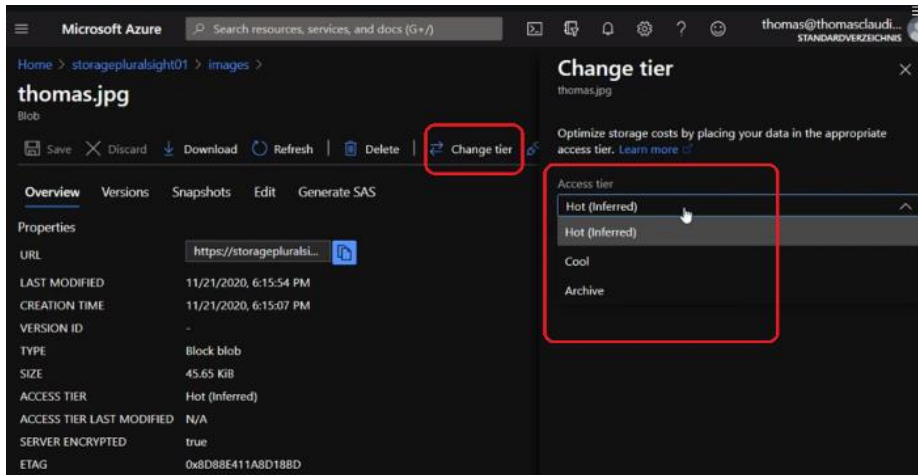
You can also change the tier using .NET:

```
Await blobClient.SetAccessTierAsync(AccessTier.Cool);
```

And from Azure CLI:

```
az storage blob set-tier
    --account-key 12345
    --account-name myacct
    --container mycontainer
    --name mypic.jpg
    --tier Cool
```

You can also change blob tier using lifecycle rules. Go to the storage account > Blob Service > Lifecycle Management > +Add a rule
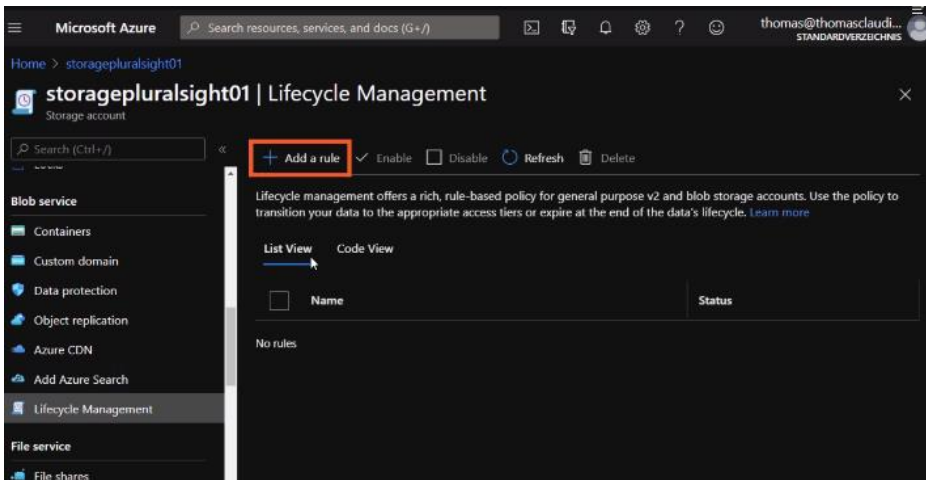
Rehydrating Blob Data From Archive Tier

There are two options for rehydrating archive tier blobs:

- copy archived blob to an online tier (recommended option)
- change a blob's access tier to online.

Priorities available for rehydration:

- standard – takes up to 15 hours
- high priority – may complete in under one hour if blob is under 10 GB in size.

Implement storage policies
Skill: implement storage policies, data archiving, and retention

https://docs.microsoft.com/en-gb/learn/modules/manage-azure-blob-storage-lifecycle/

https://docs.microsoft.com/en-gb/learn/modules/manage-azure-blob-storage-lifecycle/3-blob-storage-lifecycle-policies

https://docs.microsoft.com/en-us/azure/storage/blobs/storage-lifecycle-management-concepts?tabs=azure-portal

https://www.youtube.com/watch?v=R_YrhDD47vk

https://www.youtube.com/watch?v=XIBr4Cn26e8

Lifecycle Management Policies

You want to automate blob transition between Hot, Cool, and Archive storage tiers. There is a default access tier for the entire blob storage account.
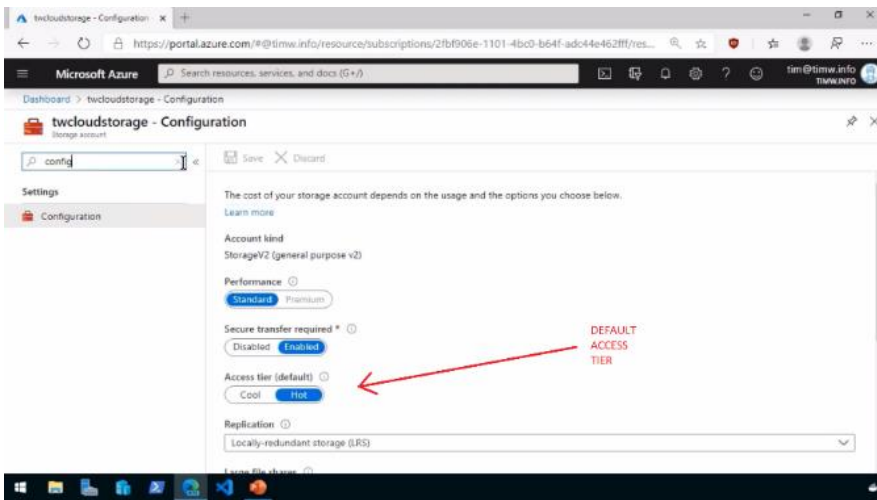
But what if we have thousands of blobs across different containers and we want to instantiate retention policies such that the access tier will change from say hot to cool if the document hasn't been modified in at least N number of days? What if you want to go from cool to archive if files reach a certain threshold? That's part of the life cycle management.
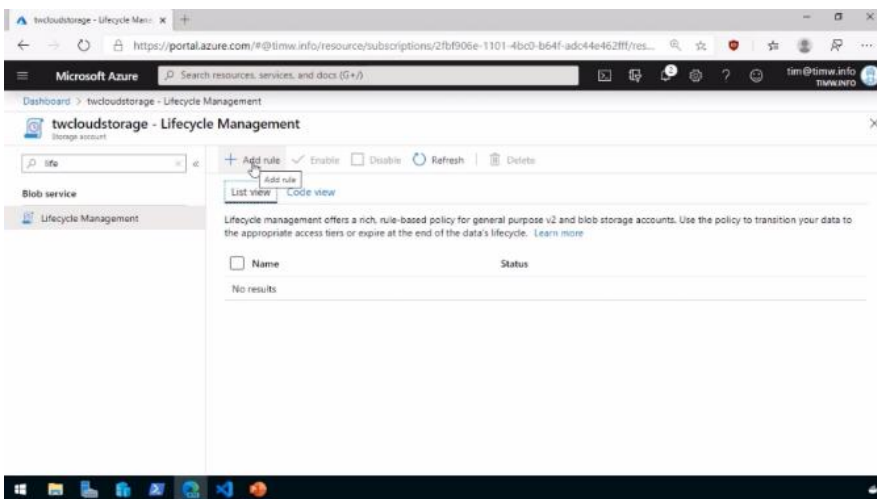
Go to Add Rule+ and set the retention amounts to create what's called the Action Set.
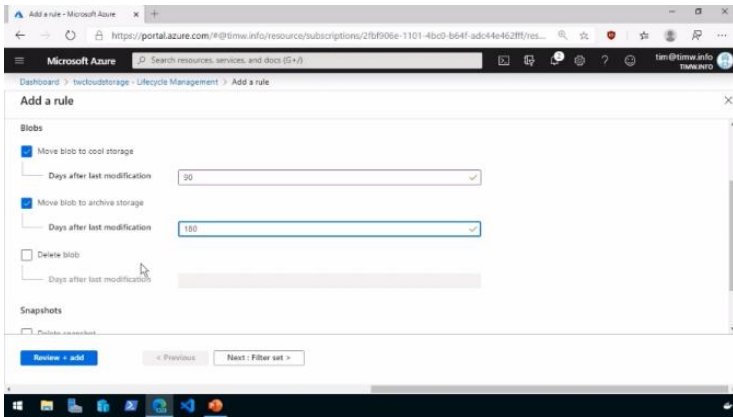
Now we can filter within the storage account which blobs we want to apply with this rule definition.
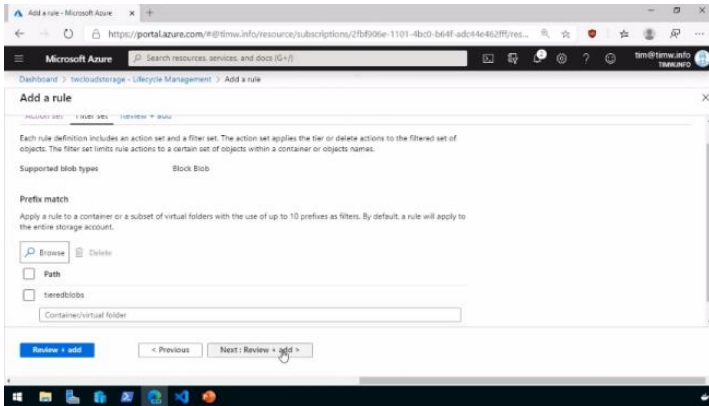
A lifecycle management policy is a collection of rules in a JSON document:

```json
{
  "rules": [
    {
      "name": "rule1",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "filters": {
          "blobTypes": [ "blockBlob" ],
          "prefixMatch": [ "container1/foo" ]
        },
        "actions": {
          "baseBlob": {
            "tierToCool": { "daysAfterModificationGreaterThan": 30 },
            "tierToArchive": { "daysAfterModificationGreaterThan": 90 },
            "delete": { "daysAfterModificationGreaterThan": 2555 }
          },
        }
      },
    },
    {
      "name": "rule2",
      "type": "Lifecycle",
      "definition": {...}
    }
}
```

```
    ]
}
```

Filters can be blobTypes, prefixMatch, blobIndexMatch. Actions can be tierToCool, enableAutoTierToHotFromCool, tierToArchive, delete. Run condition can be daysAfterModificationGreaterThan or daysAfterCreationGreaterThan.

Storage Access Policies

Storage access policies provide us with an additional level of control over SAS. The policy overrides the restrictions (time and permissions) of a SAS. SAS is not available at the container level, but access policy is.

To add stored access policies from the Azure Portal, select a storage resource and go to Settings > Access Policy
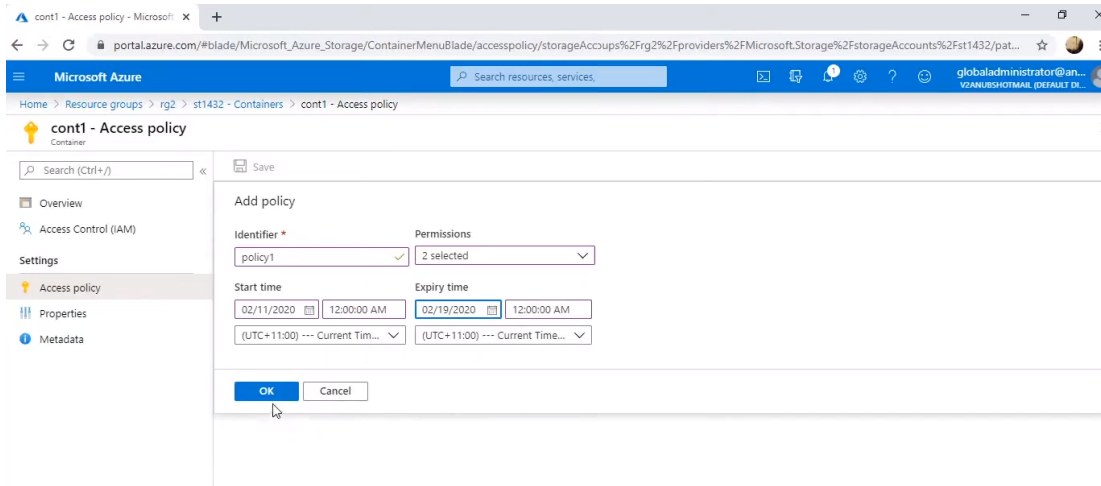
3. Implement Azure security (20-25%)

3.1 Implement user authentication and authorization

- create and implement shared access signatures
- authenticate and authorize users and apps by using Azure Active Directory
- authenticate and authorize users using Microsoft Identity Platform

https://docs.microsoft.com/en-us/azure/active-directory/develop/reference-app-manifest

Security principal can be a signed-on user, a group, a service principal (headless process), or a managed identity.

In Azure AD, an *App Manifest* is the definition of an application object within the *Microsoft Identity Platform*. The application manifest contains a definition of all the attributes of an *application object* in the Microsoft identity platform. It includes configuration for authentication and authorization. Example of an App Manifest:

```
{
    "id": "058477a1-5d5f-45e7-bc71-66c059a58eff",
    "name": "SampleSPA",
    ...
    "allowPublicClient": true,
    "groupMembershipClaims": "All",
    "oauth2AllowIdTokenImplicitFlow": true,
    "oauth2AllowImplicitFlow": true,
    "oauth2Permissions": [],
    "oauth2RequirePostResponse": false,
    ...
}
```

Common App Manifest attributes:

- **appId** unique identifier for the app that is assigned to an app by Azure AD
- **appRoles** the collection of roles that an app may declare
- **groupMembershipClaims** the groups claim issued in an access token that the app expects. Supported values:
    - *None*
    - *SecurityGroup* – security groups and Azure AD roles
    - *ApplicationGroup* – includes only groups that are assigned to the application
    - *DirectoryRole* – the Azure AD roles the user is a member of
    - *All* – all of the Azure AD groups and roles that the signed-in user is a member of
- **optionalClaims** the collection of optional claims returned by the security token service for this specific app
- **oauth2AllowImplicitFlow** whether this web app can request OAuth2.0 implicit flow access tokens. The default is false. This flag is used for browser-based apps, like JavaScript single-page apps.
- **oauth2permissions** the collection of OAuth 2.0 permission scopes that the web API (resource) app exposes to client apps. These permission scopes may be granted to client apps during consent.
- **signInUrl** the URL to the app's home page
- **signInAudience** what Microsoft accounts are supported for the current application . Supported values:
    - *AzureADMyOrg*
    - *AzureADMultipleOrgs*
    - *AzureADandPersonalMicrosoftAccount*
    - *PersonalMicrosoftAccount*

Applications can have their own roles (appRoles in the manifest).

Once roles are added to the application, you can assign them to the application's users.

Assign users to roles
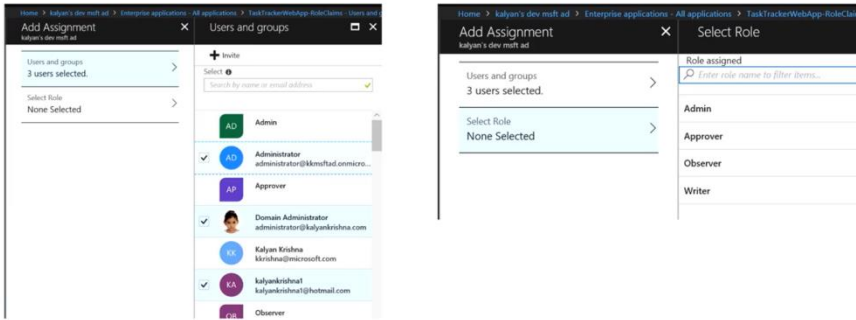
The roles will then appear in the Microsoft Identity Platform id tokens and access tokens.

Roles in token

```json
{
  "aud": "300e33f5-e62e-4581-acd2-542ece0965cc",
  "iss": "https://login.microsoftonline.com/536279f6-15cc-45f2-be2d-61e352b51eef/v2.0",
  "iat": 1563969244,
  "nbf": 1563969244,
  "exp": 1563973144,
  "aio": "AeQAG/8MAAAAYPOQy4ROQXwGbt+LpH37Q8I=",
  "groups": [
    "MSDemoUsers"
  ],
  "name": "Kalyan Krishna",
  "nonce": "6369956633167913NDUwODI0",
  "oid": "98d51ac8-a756-43ef-876f-e7e64c89b323",
  "preferred_username": "kkrishna@contosoorg.net",
  "roles": [
    "DirectoryViewers"
  ],
  "sub": "bGcfwO94xuVM7Dv-O62Bb76ZlB9RzHa0R-48jtQgKgg",
  "tid": "536279f6-15cc-45f2-be2d-61e352b51eef",
  "uti": "WQBn7mDb2UygvE7fPrIfAA",
  "ver": "2.0"
}
```

In your .Net application code, you can authorize based on these roles.

Asp.net core middleware configuration

```csharp
// Startup.cs

public void ConfigureServices(IServiceCollection services)
{
    // Other code

    // By default, the claims mapping will map claim names in the old format to accommodate older SAML application.
    // 'http://schemas.microsoft.com/ws/2008/06/identity/claims/role' instead of 'roles'
    // This flag ensures that the ClaimsIdentity claims collection will be built from the claims in the token
    JwtSecurityTokenHandler.DefaultMapInboundClaims = false;

    services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
    {
        // The claim in the Jwt token where App roles are available.
        options.TokenValidationParameters.RoleClaimType = "roles";
    });
}

// In code..(Controllers & elsewhere)
[Authorize(Roles = "DirectoryViewers")]

// or
User.IsInRole("DirectoryViewers");
```

Privileged Identity Management
Privileged Identity Management (PIM) is a feature for protecting privileged admin accounts in Azure AD.

Mutual TLS Authentication
Mutual TLS authentication is used for SSL certificates in Azure App Service. The certificate is in the X-ARR-ClientCert HTTP header. It is Base64 encoded. The mutual TLS authentication service will not validate the certificate for you, it will

simply make sure that it's present in the HTTP header. You have to write code in your application to validate the certificate.

Shared Access Signature (SAS)
Skill: create and implement shared access signatures

When working with storage accounts, you need to control who and how much time a process, person, or application can access your data.

- **Shared Key Authorization** – use one of the two access keys configured at the Azure Storage account level. Azure Storage account keys are like the root password of the Azure Storage account. The client embeds the shared key in the HTTP Authorization header of every request, and the Storage account validates the key. If the primary key is compromised, change each app to use the secondary key and regenerate the primary key. Consider this as the new secondary key.
- **Shared Access Signature** – use Shared Access Signatures (SAS) for narrowing the access to specific containers inside the Storage Account. The advantage of using SAS is that you don't need to share the Azure Storage account's access key. You can also configure a higher level of granularity when setting access to your data.

Three types of SAS

1. <u>User-delegation SAS</u> – we're not using the storage account key to sign this signature; we're using the user's account credentials. Available for blob storage use only.
2. <u>Service SAS</u> (with stored access policy) – scoped to a specific service within your storage account. Service SAS is secured with a storage account key and delegates access to a resource in only one of Azure storage services (blob, file, queue, or table). Service SAS is also linked to a Stored Access Policy.
3. <u>Account SAS</u> – scoped to one or more services within your storage account. Combines all operations available to user and service SAS plus additional operations that apply to services such as get/set service properties; secured with a storage account key and delegates access to a resource in one or more of Azure storage services (blob, file, queue, or table).

User-delegation SAS and account SAS types are *Ad Hoc* – you put in things like start time and expiration. With the service SAS type, the Stored Access Policy has the start time and expiration.

User-delegation SAS is considered to be a best practice because it uses Azure AD credentials instead of the storage account key – use whenever possible (available for Blob storage only).

You can configure each SAS with a different level of access:

Services	grant access to only the services required (blob, file, queue, or table)
Resource types	grant access to a service, container, or object
Permissions	configure the permissions the user is allowed to perform
Expiration	configure the period for which the SAS is valid
IP address	grant access to a single IP address or range of IP addresses
Protocol	HTTPS-only or HTTP and HTTPS. You cannot grant access to HTTP-only

You append the SAS token to the URL that you use for accessing your storage resource. One of the parameters of a SAS token is the signature. The Azure Storage Account service uses this signature to authorize access to the storage resources.

Accounts that store user data have two typical designs:

- Clients upload and download data through a front-end proxy service, which performs authentication. This design becomes complicated or expensive to scale for high volume transactions.
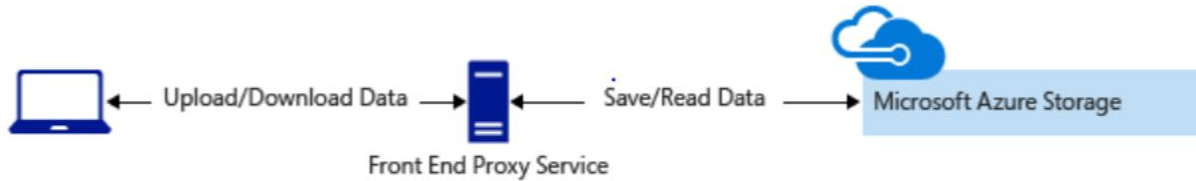
- A lightweight service authenticates the client, as needed. Next, it generates a SAS. After receiving the SAS, the client can access storage account resources directly.
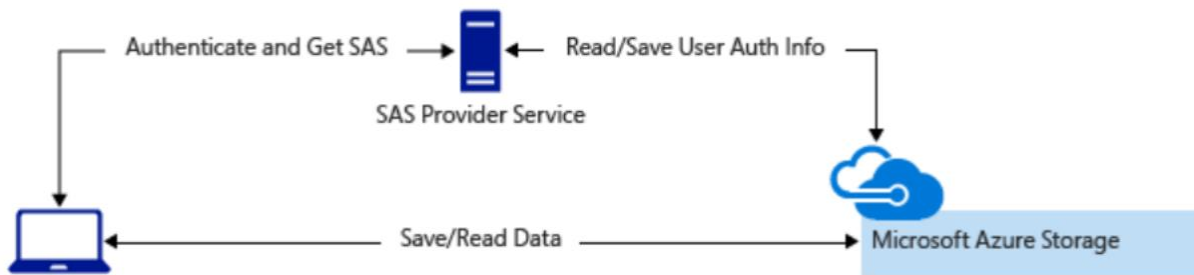
Skill: authenticate and authorize users and apps by using Azure Active Directory

An Azure subscription has a trust relationship with Azure Active Directory (Azure AD). A subscription trusts Azure AD to authenticate users, services, and devices. Multiple subscriptions can trust the same Azure AD directory. Each subscription can only trust a single directory.
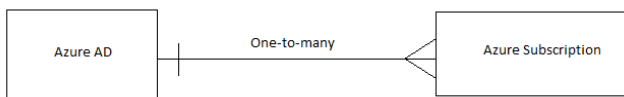
Trust relationship between Azure AD and Subscriptions

Azure AD – Register Apps

Before your application can use the Azure AD service for authenticating users, you need to register the application in your Azure AD tenant. This establishes a trust relationship between the app and Microsoft Identity Platform.

To register your app: Azure Portal > Azure Active Directory > Manage > App Registrations > follow wizard

Your application needs to use certificates and secrets to provide the application's identity when requesting a token. As with any other authorization system in C#, you need to add the [Authorized] attribute to any resource that you want to protect.

Typically, you create an appsetting for the ClientId and pass it in when your app interacts with Azure. You get the value of ClientId from the Azure Portal, Azure AD blade when you register the app.

Azure AD – Authenticate Users

You need to consider whether the users of your application would be

- **Users from your organization only** Any person that have a user account in your Azure Active Directory tenant would be able to use your application.
- **Users from any organization** You use this option when you want any user with a professional or educational Azure Active Directory account to be able to log into your application.
- **Users from any organization or Microsoft accounts** You use this option if you want your users to log into your application by using professional, educational, or any of the freely available Microsoft accounts.

Azure AD – Federated Services

Azure AD Federated Services provides simplified web single sign-on (SSO) capabilities, enabling users access resources in Azure with on-premises credentials. Identity federation gives users access to Azure applications by confirming their credentials through the on-premises directory.

RBAC – Control Access to Resources

Azure RBAC is an authorization system built on Azure Resource Manager that provides fine-grained access management of resources in Azure. You grant access by assigning the appropriate role to users, groups, and applications. The *scope* of a role assignment can be a subscription, a resource group, or a single resource. A role assigned at a parent scope also grants access to the child scopes contained within it.

In the Azure Portal, RBAC is managed for a resource by selecting the **Access Control (IAM)** pane for that resource.
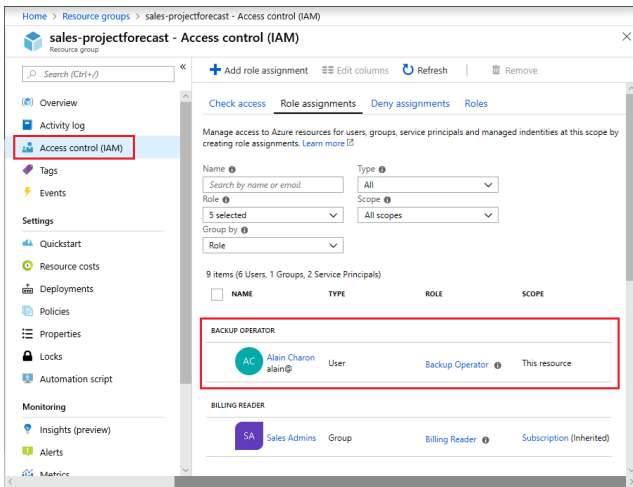
To create a role assignment, you need three elements: a security principal, a role definition, and a scope. Think of these elements as "who", "what", and "where".
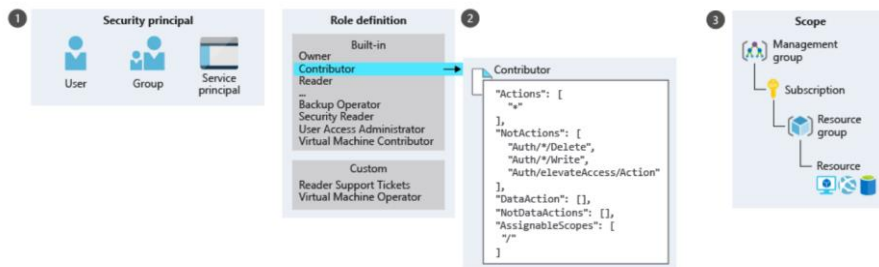
A *role assignment* is the process of binding a *role definition* to a *security principal* at a particular *scope*, for the purpose of granting access.

Four fundamental built-in roles:

- Owner – full access to all resources, including the right to grant access to others
- Contributor – create and manage all types of Azure resources, but can't grant access to others
- Reader – view existing Azure resources
- User Access Administrator – manage user access to Azure resources

Azure RBAC is an allow model. So, if one role assignment grants you read and a different role assignment grants you write, you will have read and write permissions on that resource group. NotActions are subtracted from Actions to compute the effective permissions.

Authenticate and authorize users using Microsoft Identity Platform
Skill: *authenticate and authorize users using Microsoft Identity Platform*

https://www.youtube.com/watch?v=uDU1QTSw7Ps

https://docs.microsoft.com/en-us/learn/paths/m365-identity-associate/

https://docs.microsoft.com/en-us/learn/modules/getting-started-identity/

https://docs.microsoft.com/en-us/learn/modules/getting-started-identity/2-different-token-types

https://docs.microsoft.com/en-us/azure/active-directory/develop/

https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-overview

https://docs.microsoft.com/en-us/azure/active-directory/develop/app-objects-and-service-principals

https://docs.microsoft.com/en-us/azure/storage/common/storage-auth-aad-app

The Microsoft Identity Platform is the next version of Azure AD, Microsoft's hosted identity management service. The Microsoft Identity Platform provides an authentication service, open-source libraries, and application management tools. It offers users a unified way to authenticate into apps with work, school, or personal accounts.

For developers, it's an evolution of Azure AD.

Authentication Service	Open-Source Libraries	Application Management
• Azure Active Directory • Azure AD Connect • ADFS (federated service) • So much more...	• MSAL (authentication library) • Microsoft.Identity.Web • OpenID connect	• Gallery and non-gallery apps • Single Tenant and Multi-Tenant apps • Authorization • Consent • Logs • And much more ...

Basics of modern authentication:

1. Apps request tokens from Microsoft Identity
2. Authentication happens over shared web surfaces to provide SSO
3. App passes Access Token to API
4. API validates Access Token before returning the result

The MSAL library wraps the complexity of working at the protocol level, by providing straightforward methods with behind-the-scenes capabilities such as passwordless authentication, MFA, and SSO. The vast majority of modern authentication can be performed by asking the MSAL library to sign in the user:

`MSAL.loginPopup()`   simply ask to sign in the user – implicitly supports all of the ways that a user could authenticate

`MSAL.acquireTokenSilent()`	get access tokens silently without interrupting the user's flow
`MSAL.acquireTokenPopup()`	if Microsoft Identity does need to talk to the user, then it falls back to acquiring the access tokens interactively

Modern authentication refers to protocols that scale to the size of the internet. They can provide secure access to users and devices that are outside the corporate network. There is a clear distinction between the identity provider who provides authentication services, and the service provider that needs to verify the user's identity in order to grant access to secure resources. The application is no longer concerned with managing user credentials – they rely on claims (or characteristics) of the user. To keep these claims safe, modern authentication protocols such as the Microsoft Identity Platform, have mechanisms to verify that their values haven't been tampered with. Modern authentication protocols are also stateless. The identity provider never remembers which party a given authentication request was for, and the reliant party doesn't remember which identity provider was used to perform authentication.

Use case 1: client app wants to authenticate a user to sign in and allow user to access the client app itself => **ID token**. An ID token is a security token that allows the client to verify the identity of the user.

Use case 2: client app wants to authenticate user and get user's consent to access resources from another source (resource provider) on behalf of the user => **access token**. Access tokens enable clients to securely call APIs protected by Azure AD.

Developers also need to think about user provisioning. How do your customers add their users to your application? Developers should use Azure AD with a SCIM endpoint. Microsoft Identity Platform builds on SCIM, an industry standard, so that Microsoft Identity Platform works across the identity industry.

Azure Active Directory for provisioning users

Microsoft Identity Platform also performs publisher verification so that users are more accepting of your application by providing them with confidence that they are getting the right application.

Microsoft Identity Platform gives developers:

- libraries for implementing standards-based authentication & authorization
- identity protection and conditional access
- easy access to Microsoft 365 data from Microsoft Graph
- user provisioning through Azure AD
- publisher verification

Azure AD B2B enables users in an organization to securely share files and resources with users in another organization.

Azure AD B2C enables an organization to provide customer facing apps that enable users to sign in with an identity that they already have (facebook, gmail, …) or create an account for the solution.

Microsoft Graph is a gateway to the cloud. It allows developers to build applications that can provide users with access to their data, by authenticating with Microsoft Identity.

OpenID Connect extends the OAuth 2.0 authorization protocol to be used as an authentication protocol, allowing you to implement single sign-on (SSO). It uses an **ID token** – a JSON Web Token (JWT) security token – that allows the client to verify the identity of the user, contains basic profile info about the user, and claims about the user. **Access tokens** – a JSON Web Token (JWT) security token – that enable clients to securely call APIs protected by Azure AD.
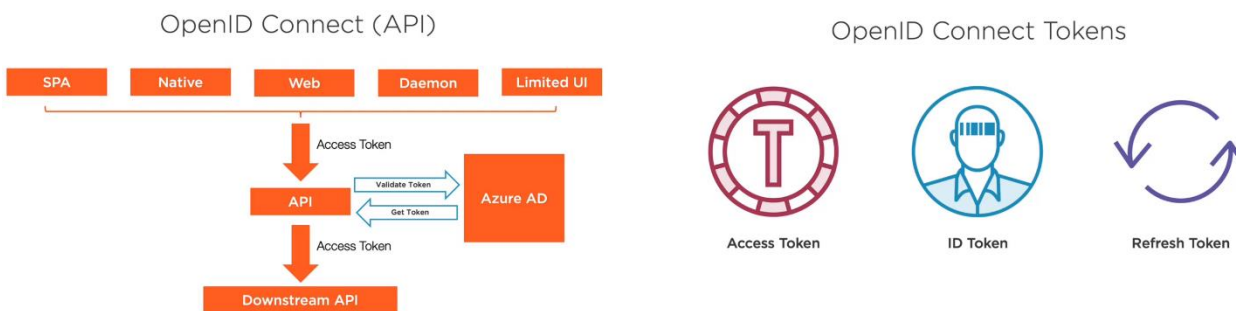
Microsoft Authentication Library (MSAL) is a well-maintained OpenID Connect library that is kept up to date by Microsoft to ensure that your implementation is as secure as possible.

Single Tenant apps: single AD directory – no one outside your organization can sign in and use the application.

Multi-tenant apps: multiple AD directories – more than just users from your organization can sign in and use the application. Multi-tenant apps can also allow users to sign in using their personal Microsoft account. Building multi-tenant apps can be a challenge because of the different policies that as set by different administrators in different tenants.

Application object defines what the developer knows about the application (auth params, secrets, certificates, permissions, roles, …). Azure AD applications are defined by exactly one application object. The application object resides within the Azure tenant where the app was registered. *The application object is the global definition of the app across all tenants.*

Service principal defines how an application will operate within your tenant. The service principal is created when an application is granted permission to use a resource in a tenant. *The service principal is the local instance of the app in a specific tenant.*

Application objects have a one-to-many relationship with service principals.

Microsoft identity platform

http://aka.ms/IdentityPlatform
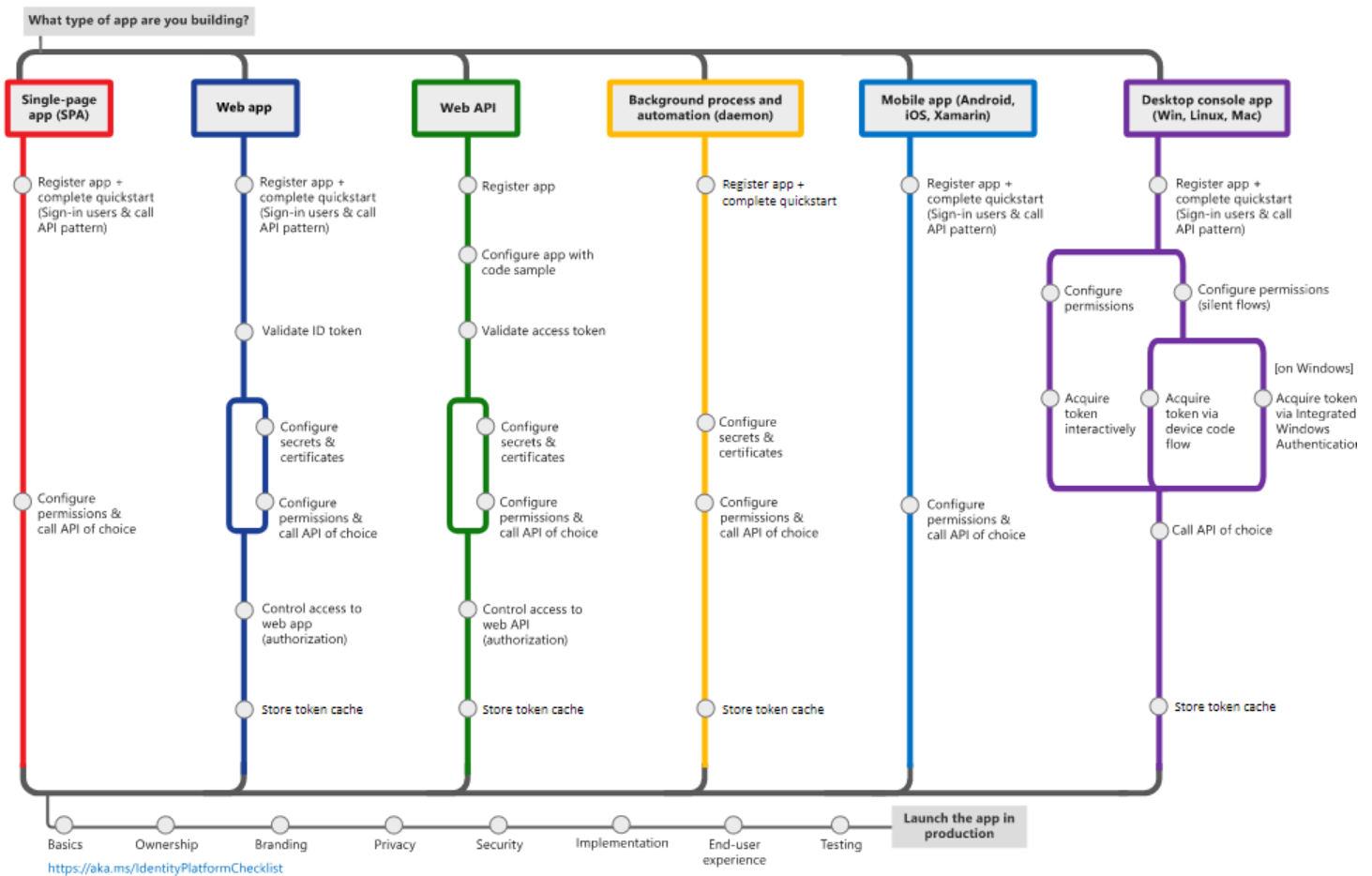
Metro map showing several application scenarios in Microsoft identity platform

Azure Function Authorization Levels

- Function
- Anonymous
- Admin

Function and Admin levels require keys.

3.2 Implement secure cloud solutions

- secure app configuration data by using the App Configuration and Azure Key Vault
- develop code that uses keys, secrets, and certificates stored in Azure Key Vault

- implement solutions that interact with Microsoft Graph

Secure App Configuration Data
Skill: secure app configuration data by using the App Configuration and Azure Key Vault

Although you can create configuration settings for your Azure App Services separately for each instance, it's easier to manage when you use a centralized configuration service.

Azure App Configuration Service
Azure App Configuration service is a centralized configuration service that allows you to share the same configuration between different Azure App Service instances. You can also use it with containerized applications or with applications running inside virtual machines. When you work with Azure App Configuration, you need to deal with two different components: The App Configuration store and the SDK.

You create key-value pairs for storing your configuration and you use the key for retrieving the value from the store. You can also add a label attribute to a key. By default, the label attribute is null. You can use the label for making values different using the same key. This is especially useful when used for deployment environments: The following three examples are different keys because the labels are different:

- Key = AppSample:DBConnection – Label = Development
- Key = AppSample:DBConnection – Label = Staging
- Key = AppSample:DBConnection – Label = Production

To use Azure App Configuration service from code, you can use a connection string for non-production environments, but you should consider using Managed Service Identity for production environments, instead of using connection strings.

Azure Key Vault
Azure Key Vault allows you to securely store keys, secrets, certificates, and any other sensitive settings that your application may need. You can use the Azure Key Vault in conjunction with Azure App Configuration by creating references in Azure App Configuration to items stored in the Azure Key Vault. You can store Key Vault information in software or hardware security module (HSM), but HSM requires the Premium sku. By default, the Key Vault endpoint is publicly available, but can be secured using VNets and NSGs. Key Vault is available under two different sku's, Standard and Premium.

Different Planes

Management plane	Data plane
Manage Key Vault itself	Manage Key Vault contents
Role-based access control	Access policy and firewall rules
Uses Azure Active Directory	Uses Azure Active Directory
Custom and built-in roles	Custom or templates in portal
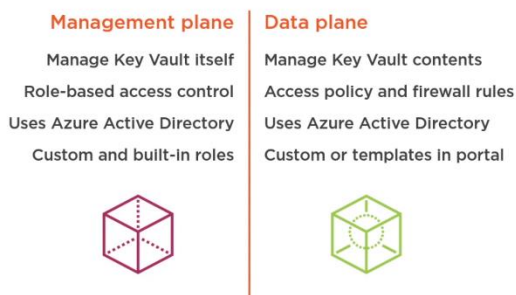
When adding a secret to Key Vault, Azure creates a URL so that the secret can be accessed programmatically.

Azure Key Vault *references* are strings which include a special prefix that we can use within our configuration settings. It will take care of configuring a managed identity to go and get this value from Azure Key Vault.

(See: https://docs.microsoft.com/en-us/azure/app-service/app-service-key-vault-references)

Azure Key Vault lets you control permissions for users, developers and administrators. You can receive logging so you always know how your secrets are accessed.

Main benefits:

- Separation of sensitive info from code
- Restricted access with access policies
- Centralized secret storage
- Logging and monitoring

Creating a key vault from the portal: All services > Key vaults > +Create > follow wizard

Creating a Key Vault with PowerShell:

```
New-AzKeyVault -VaultName 'MyVault' -ResourceGroupName 'rg-101' -Location 'East US'
```

Creating a Key Vault with CLI:

```
az keyvault create --resource-group 'rg-101' [ …optional parameters… ]
```

Soft delete key vault allows you to recover accidentally deleted secrets or even whole key vaults within the retention period (7 to 90 days). You can force a hard delete using the PURGE operation, unless Purge-Protection is enabled.
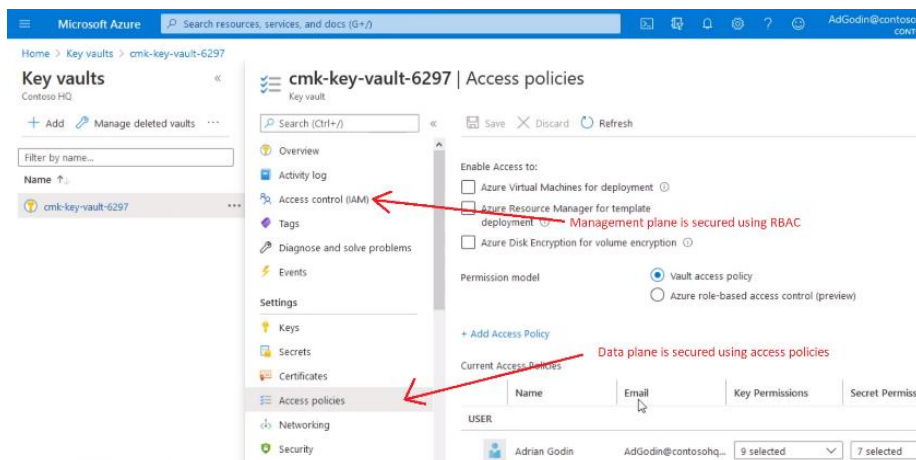
There is only one built-in RBAC role to configure access to the management plane: Key Vault Contributor. This role gives access to manage the key vault without access to the data. You can also create custom roles for the management plane.

```
# PowerShell command to assign role to a user
New-AzRoleAssignment `
    --UserPrincipalName "rob@home.com" `
    --Scope ($myKeyVault).ResourceId `
    --RoleDefinitionName "Key Vault Contributor"
```

To configure access to the key vault data plane, we use access policies. Access policies are applied to all objects in a key vault of a given type (keys, secrets, or certificates). Access policies assign permissions to an Azure AD object (application, group, or user). Permissions include things like get, set, list, create, delete, etc.

```
# PowerShell command to assign key vault access policy to a user
Set-AzKeyVaultAccessPolicy `
    --VaultName "MyVault" `
    --ResourceGroupName "MyRg" `
    --PermissionsToSecrets ("get","list","set") `
    --PermissionsToKeys ("get","list","create") `
    --UserPrincipalName "rob@here.xyz"
```

To secure a key vault's public endpoint, make a private endpoint inside a VNet, secured by NSG.

Private Endpoints
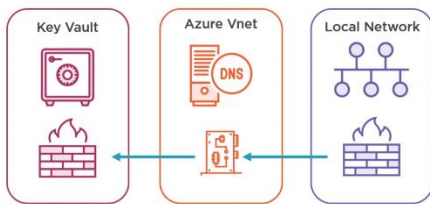
Once your key vault has been adequately secured, you can start adding keys, certificates, or secrets to your key vault using the Azure Portal, command line, or API/SDK. For example:

```
# PowerShell command to create a key in key vault
Add-AzKeyVaultKey `
    --VaultName "MyKeyVault" `
    --Name "MyKey"
```

Manage keys, secrets, and certificates by using the Key Vault API
Skill: develop code that uses keys, secrets, and certificates stored in Azure Key Vault

https://docs.microsoft.com/en-us/learn/modules/manage-secrets-with-azure-key-vault/

https://docs.microsoft.com/en-us/learn/modules/manage-secrets-with-azure-key-vault/5-access-secrets-from-web-app?pivots=csharp

Using Identity Access Management and RBAC, you can configure access policies for granting access to your Key Vault. Developers no longer need to store this sensitive information on their computers, and you can apply fine-grained access control, allowing access to specific secrets only.

Getting a token from Azure Active Directory requires a secret or certificate because anyone with a token could use the app identity to access all of the secrets in the vault. Managed Identities for Azure resources is an Azure feature that your app can use to access Key Vault and other Azure services without having to manage even a single secret outside of the vault.

The Azure Key Vault API is a REST API that handles all management and usage of keys and vaults. Each secret in a vault has a unique URL, and secret values are retrieved with HTTP GET requests.

With ASP.NET Core's AddAzureKeyVault method, you can load all the secrets from a vault into the Configuration API at startup. This technique enables you to access all of your secrets by name using the same IConfiguration interface you use for the rest of your configuration. Apps that use AddAzureKeyVault require both Get and List permissions to the vault. AddAzureKeyVault only requires the vault name as an input. It automatically handles managed identity authentication — when used in an app deployed to Azure App Service with managed identities for Azure resources enabled, it will detect the managed identities token service and use it to authenticate.

To use the Key Vault .NET API, you need **Microsoft.Azure.KeyVault** package (available from NuGet). Your application authenticates using its service principal so Azure will know who you are. You then need to get an access token for authenticating with the Azure Key Vault. You create a KeyVaultClient object that is responsible for the communication with the Azure Key Vault services.

```
// components of the secret's URL
var vaultName = "MyKeyVault";
var objectType = "secrets";
var secretName = "MySecret";
var version = "2";
var baseURL = $"https://{vaultName}.vault.azure.net";

// get authenticated key vault client
var provider = new AzureServiceTokenProvider(); // uses service principal
var tokenCB = provider.KeyVaultTokenCallback;
var authCB = new KeyVaultClient.AuthenticationCallback(tokenCB);
var client = new KeyVaultClient(authCB);

// get value of secret
var secret = await client.GetSecretAsync($"{baseURL}/{objectType}/{secretName}/{version}");
console.log($"{secret.value}");

// create new secret
secretName = "NewSecret";
await client.SetSecretAsync(baseURL, secretName, "This is a secret value.");

// delete a secret
await client.DeleteSecretAsync(baseURL, secretName);
```

The Key Vault API provides specialized methods (Get, List, Set, Delete) for each item type (secrets, keys, certificates). Most of the methods that work with items require the vault URL and the name of the item that you want to access. If you try to create a new secret, key, or certificate using the same name of an object that already exists in the vault, you are creating a new version of that object.

You can also access your key vault secrets through other types of code such as ARM templates or CLI. For example

```
az keyvault create \
    --resource-group learn-950b3a03-fe30-4c1f-815d-d259db8eff26 \
    --location centralus \
    --name ridvaultname

az keyvault secret set \
    --name SecretPassword \
    --value reindeer_flotilla \
    --vault-name ridvaultname
```

Implement solutions that interact with Microsoft Graph
Skill: implement solutions that interact with Microsoft Graph

https://docs.microsoft.com/en-us/learn/paths/m365-msgraph-associate/

https://docs.microsoft.com/en-us/learn/modules/msgraph-build-aspnetmvc-apps/

https://docs.microsoft.com/en-us/graph/overview

https://developer.microsoft.com/en-us/graph/get-started
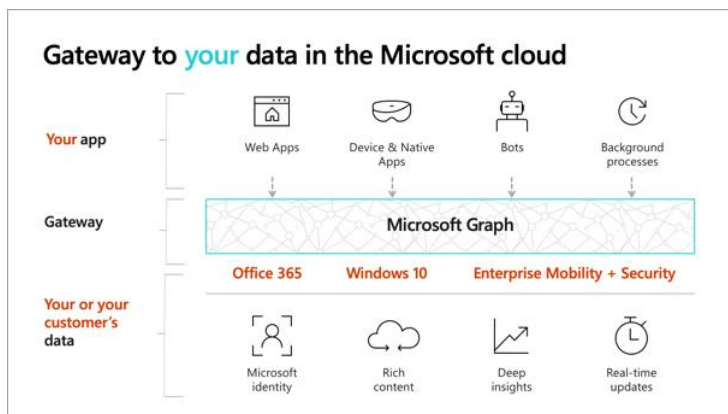
Microsoft Graph is a REST API that can be used to interact with Microsoft 365. Microsoft 365 is a term that encompasses the collection of products and services Microsoft has to offer, including Office 365. Microsoft Graph is a unified programming model to access the tremendous amount of data in the cloud. It is a proxy, which serves as the gateway to all your data in the Microsoft Cloud. Microsoft Graph enables developers to build applications that provide users access to their data. Developers and users can access their data with Microsoft Graph by authenticating and authorizing users with Microsoft identity framework. It's also an API that you can use to connect to a variety of Office 365 services as well as Azure – it provides a single end point and tools to connect to all of these services. Microsoft Graph is a restful API to expose data over 'https' so it is secure. Microsoft Graph is secured with Azure AD, which means that developers will need to authenticate with Azure AD and obtain an OAuth access token. Microsoft Graph, acting as a proxy to many of the services in the Microsoft cloud, provides access via a single top-level endpoint, https://graph.microsoft.com.

APIs centralized and consistent

https://graph.microsoft.com

Operation	Service endpoint
GET my profile	https://graph.microsoft.com/v1.0/me
GET my mail	https://graph.microsoft.com/v1.0/me/messages
GET my calendar	https://graph.microsoft.com/v1.0/me/calendar
GET my contacts	https://graph.microsoft.com/v1.0/me/contacts
GET my photo	https://graph.microsoft.com/v1.0/me/photo/$value
GET my files	https://graph.microsoft.com/v1.0/me/drive/root/children
GET my manager	https://graph.microsoft.com/v1.0/me/manager
GET last user to modify file foo.txt	https://graph.microsoft.com/v1.0/me/drive/root/children/foo.txt/lastModifiedByUser
GET users in my organization	https://graph.microsoft.com/v1.0/users
GET group conversations	https://graph.microsoft.com/v1.0/groups/<id>/conversations
GET people related to me	https://graph.microsoft.com/beta/me/people
GET my tasks	https://graph.microsoft.com/beta/me/tasks
GET my notes	https://graph.microsoft.com/beta/me/notes/notebooks
GET files trending around me	https://graph.microsoft.com/beta/me/insights/trending

Apps only need to authenticate once and obtain a single access token that can be used to retrieve data from multiple sources, greatly simplifying the development process.

Accessing the Microsoft Graph

Direct REST API – any platform, language, or framework; *Native SDKs* – utilizes different implementations, abstracts the details of REST requests over HTTP, .NET, IOS, Android, Php, etc.

Graph Explorer is a tool to query all the protected data.

"Microsoft Graph is a gateway to DATA and intelligence in Microsoft 365"

"Microsoft Graph is a Restful API to expose DATA over 'https'"

"Microsoft Graph offers Graph Explorer to query all the protected DATA"
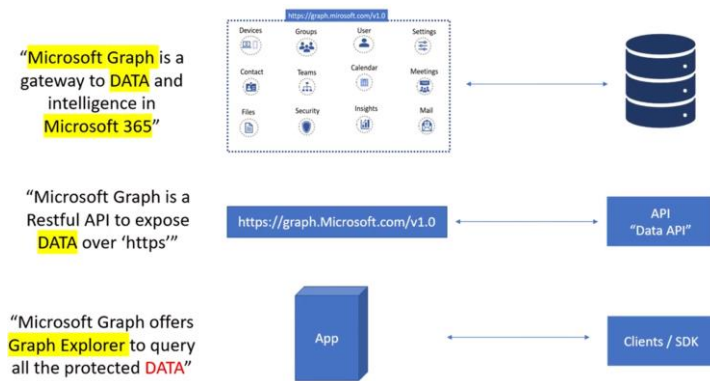
Microsoft Graph App Patterns

Scenario: A Web app calling Graph

Scenario: A Web app calling Graph

Authorization
code flow

Web App   MS Graph

```
IConfidentialClientApplication confidentialClientApplication = ConfidentialClientApplic
ationBuilder
    .Create(clientId)
    .WithRedirectUri(redirectUri)
    .WithClientSecret(clientSecret) // or .WithCertificate(certificate)
    .Build();

AuthorizationCodeProvider authenticationProvider = new AuthorizationCodeProvider(confid
entialClientApplication, scopes);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);
```

https://docs.microsoft.com/azure/active-directory/develop/scenario-web-app-call-api-overview

https://docs.microsoft.com/azure/active-directory/develop/scenario-web-app-call-api-overview

Scenario: A Desktop app calling Graph

Scenario: Desktop app calling Graph

Authorization code flow
Integrated Windows
Authentication
Username/Password

Desktop App   MS Graph

```
IPublicClientApplication publicClientApplication = PublicClientApplicationBuilder
    .Create(clientId)
    .WithTenantId(tenantID)
    .Build();

InteractiveAuthenticationProvider authenticationProvider = new InteractiveAuthenticatio
nProvider(publicClientApplication, scopes);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);

IntegratedWindowsAuthenticationProvider authenticationProvider = new IntegratedWindowsA
uthenticationProvider(publicClientApplication, scopes);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);
```

https://docs.microsoft.com/azure/active-directory/develop/scenario-desktop-overview

https://docs.microsoft.com/azure/active-directory/develop/scenario-desktop-overview

Scenario: A Web API calling Graph

Scenario: Web API calling Graph


```
IConfidentialClientApplication confidentialClientApplication = ConfidentialClientApplic
ationBuilder
    .Create(clientId)
    .WithRedirectUri(redirectUri)
    .WithClientSecret(clientSecret)
    .Build();

OnBehalfOfProvider authenticationProvider = new OnBehalfOfProvider(confidentialClientAp
plication, scopes);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);
```

https://docs.microsoft.com/azure/active-directory/develop/scenario-web-api-call-api-overview

https://docs.microsoft.com/azure/active-directory/develop/scenario-web-api-call-api-overview

Scenario: A Daemon app calling Graph

Scenario : Daemon app calling Graph
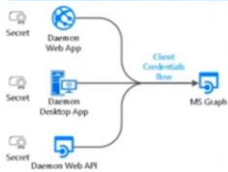

```
IConfidentialClientApplication confidentialClientApplication = ConfidentialClientApplic
ationBuilder
    .Create(clientId)
    .WithTenantId(tenantID)
    .WithClientSecret(clientSecret)
    .Build();

ClientCredentialProvider authenticationProvider = new ClientCredentialProvider(confiden
tialClientApplication);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);
```

https://docs.microsoft.com/azure/active-directory/develop/scenario-daemon-call-api?tabs=dotnet

https://docs.microsoft.com/azure/active-directory/develop/scenario-daemon-call-api?tabs=dotnet

Scenario: A Browserless app calling Graph

Scenario: Browserless app calling Graph


```
IPublicClientApplication publicClientApplication = PublicClientApplicationBuilder
    .Create(clientId)
    .Build();

DeviceCodeProvider authenticationProvider = new DeviceCodeProvider(publicClientApplicat
ion, scopes);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);
```

https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-device-code

https://docs.microsoft.com/azure/active-directory/develop/v2-oauth2-device-code

Scenario: A Mobile app calling Graph

Scenario: Mobile application calling Graph


```
IPublicClientApplication publicClientApplication = PublicClientApplicationBuilder
    .Create(clientId)
    .Build();

InteractiveAuthenticationProvider authenticationProvider = new InteractiveAuthenticatio
nProvider(publicClientApplication, scopes);
GraphServiceClient graphServiceClient = new GraphServiceClient(authenticationProvider);
```

https://docs.microsoft.com/en-us/azure/active-directory/develop/scenario-mobile-overview

https://docs.microsoft.com/azure/active-directory/develop/scenario-mobile-overview

4. Monitor, troubleshoot, and optimize solutions (15-20%)

This chapter discusses monitoring, troubleshooting, and optimizing Azure solutions.

4.1 Integrate caching and content delivery within solutions

- configure cache and expiration policies
- configure cache and expiration policies for Azure Redis Cache
- Implement secure and optimized application cache patterns including data sizing, connections, encryption, and expiration

https://docs.microsoft.com/en-us/azure/cdn/cdn-overview

https://docs.microsoft.com/en-us/learn/modules/create-cdn-static-resources-blob-storage/

https://docs.microsoft.com/en-us/learn/modules/optimize-your-web-apps-with-redis/

https://docs.microsoft.com/en-us/learn/paths/architect-modern-apps/

https://docs.microsoft.com/en-us/learn/modules/work-with-mutable-and-partial-data-in-a-redis-cache/

configure cache and expiration policies
Skill: configure cache and expiration policies

Caching is the act of storing frequently accessed data in memory. You need to control the lifetime and validity of your application's objects stored in your cache so that your users see the latest, most up-to-date content. When you replace an object on the server, you need to also replace it in cache. This is sometimes done manually through purging or file renaming.

When you cache an object, you assign it a time-to-live (TTL). Once the TTL has expired, the cache checks the file in the cache against the file on the host. If they are the same, then the TTL is reset. If they are different, then the cached file is replaced and assigned a new TTL. TTL is controlled by setting the max-age cache-control HTTP header on the web server. This can be done in different ways:

- Caching policies (rules) can be used to control how TTL values are assigned. You can create global rules for all objects within the cache, or you can use custom rules for different file types or paths within your app.
- In ASP.NET web applications you can control the TTL by setting the ClientCache property in the web.config file.
- Also, in .ASP.NET you can control the caching behavior by setting the HttpResponse.Cache property programmatically.

Web.config example:

```xml
<?xml version="1.0"?>
<configuration>
  <system.webserver>
    <staticcontent>
      <clientcache cachecontrolmode="UseMaxAge" cachecontrolmaxage="3.00:00:00" />
    </staticcontent>
  </system.webserver>
</configuration>
```

Programmatic example:

```csharp
public class HomeController: Controller
{
  public ActionResult Index()
  {
    Response.Cache.SetExpires(DateTime.Now.AddHours(1));
    Response.Cache.SetCacheability(Public);
    Response.Cache.SetLastModified(DateTime.Now);

    Return View();
  }
}
```

Azure CDN Cache Configuration

Azure CDN supports two approaches for configuring caching:

- global caching rule (overrides HTTP cache headers)
- custom caching rules (overrides global caching rule)

Azure CDN cache duration behavior:

- **Bypass Cache** – don't cache anything and ignore cache headers
- **Override** – ignore cache duration in headers, use config values
- **Set if Missing** – uses the config value if duration header is missing

Azure CDN query string handling:

- **Bypass Caching** – asset is not cached, query string always passed to origin
- **Ignore Query String** – query string is only used on initial fetch from origin (*default*)
- **Cache Every Unique URL** – URL with query string used to set cache value

configure cache and expiration policies for Redis caches
Skill: configure cache and expiration policies for Azure Redis Cache

https://docs.microsoft.com/en-us/learn/modules/optimize-your-web-apps-with-redis/

https://docs.microsoft.com/en-us/learn/modules/aspnet-session/

Redis (REmote DIctionary Server) cache is an open-source, in-memory key value pair store. It can be used as a database, cache or message broker. Azure Cache for Redis is typically used to improve the performance of systems that rely heavily on back-end data stores. Your cached data is located in-memory on an Azure server running the Redis cache as opposed to being loaded from disk by a database. Each data value is associated to a key that can be used to look up the value from the cache.

Guidelines for choosing keys:

- Avoid long keys.
- Use keys that can identify the data. For example, "sport:football;date:2008-02-02" is better than "fb:8-2-2".
- Use a convention such as "object:id", as in "sport:football".

Data is stored in Redis by using nodes and clusters. A node is a single storage location and a cluster is a set of three or more nodes.

You can create a Redis cache using the Azure portal, the Azure CLI, or Azure PowerShell. To connect to an Azure Cache for Redis instance, you'll need the host name, port, and an access key for the cache. You can retrieve this information in the Azure portal through the Settings > Access Keys page. Azure also offers a connection string which bundles this data together into a single string.

Typically, a client application will use a client library, such as StackExchange.Redis for .NET, to execute commands on a Redis cache. The main connection object is the ConnectionMultiplexer class, which abstracts the process of connecting to a Redis server. It's optimized to manage connections efficiently and intended to be kept around while you need access to the cache. Once you are done with the Redis connection, you can Dispose the ConnectionMultiplexer. This will close all connections and shutdown the communication to the server.

Configure Redis Cache
Set these parameters:

Name	The name has to be unique within Azure because it is used to generate a public-facing URL to connect and communicate with the service.
Resource Group	You can either create a new resource group, or use an existing one.
Location	Determine where the Redis cache will be physically located by selecting an Azure region. Important: Put the Redis cache as close to the data consumer as you can. Connecting to a cache in a different region can significantly increase latency and reduce reliability.
Pricing tier (sku)	Basic, Standard, Premium. You can control the amount of cache memory available on each tier by choosing a *cache level* (vm-size) from C0-C6 for Basic/Standard and P0-P4 for Premium. Microsoft recommends you always use Standard or Premium Tier for production systems. Also, use at least a C1 cache. C0 caches are meant for simple dev/test scenarios. Premium also allows you to: • configure data persistence for disaster recovery • deploy Redis cache to a virtual network for security • implement clustering to automatically split your dataset among multiple nodes
cache level (vm-size)	The size of the Azure Cache for Redis. Valid values are C0, C1, C2, C3, C4, C5, C6, P1, P2, P3, P4.

Example:

```
az redis create \
    --name "MyCache" \
    --resource-group [sandbox resource group name] \
    --location eastus \
    --sku Basic \
    --vm-size C0
```

Redis Cache Expiration Policy

Data expiration is a feature that can automatically delete a key and value in the cache after a set amount of time. Since you have limited storage with Azure Cache for Redis, you want to make sure you're only storing data that is important.

Redis console commands to implement and manage data expiration:

- EXPIRE sets the timeout of a key
- TTL returns the remaining time a key has to live
- PERSIST makes a key never expire

Alternatively, using the C# ServiceStack.Redis package:

```
public static void SetGroupChatName(string groupChatID, string chatName)
{
    using (RedisClient redisClient = new RedisClient(redisConnectionString))
    {
        //Create a key for group chat display names
        string key = groupChatID + "displayName";

        //Set the group chat display name
        redisClient.SetValue(key, chatName);

        //Set the expiration for one hour
        redisClient.Expire(key, 3600); // Note: same effect as EXPIRE command from console
    }
}
```

The default TTL for Redis Cache is infinity (never expires). If you haven't set a TTL for an item in Redis Cache, you need to manually remove that item. Use the Redis commands: KeyDelete or FlushAllDatabases.

Redis Cache Eviction Policy

Memory is the most critical resource for Azure Cache for Redis, because it's an in-memory database. Azure Cache for Redis supports max-memory eviction policies, which indicate how data should be handled when you run out of memory. There are six different max-memory eviction policies:

volatile-LRU default	evict the least recently used (LRU) keys first, among keys that have an expiry set
volatile-TTL	evict keys with a shorter TTL first, among keys that have an expiry set
volatile-random	evict keys randomly, among keys that have an expiry set
allkeys-LRU	evict the least recently used (LRU) keys first
allkeys-random	evict keys randomly
noeviction	return errors when the memory limit was reached

To set the eviction policy for Azure Cache for Redis, from Azure Portal go to the Redis and select Advanced Settings > maxmemory-policy

Store and retrieve data in Azure Redis cache

Using any of the .NET languages, you can use the *StackExchange.Redis* client for accessing your Azure Cache for Redis resource. You can also use this Redis client for accessing other Redis implementations. When reading or writing values in the Azure Cache for Redis, you need to create a *ConnectionMultiplexer* object that creates a connection to your Redis server. Creating a connection is a costly operation. For this reason, you should not create a ConnectionMultiplexer object for each read or write operation, you should store this object and reuse it across all your code.

Implement secure and optimized application cache patterns including data sizing, connections, encryption, and expiration

Skill: Implement secure and optimized application cache patterns including data sizing, connections, encryption, and expiration

https://docs.microsoft.com/en-us/learn/modules/optimize-your-web-apps-with-redis/

Cache-Aside Pattern

The application will try to read data from the cache first. If the requested data is not in the cache, the application will retrieve it from the database and then it stores the data in the cache for subsequent requests. Next time when any user requests the data, it will return it from the cache directly.
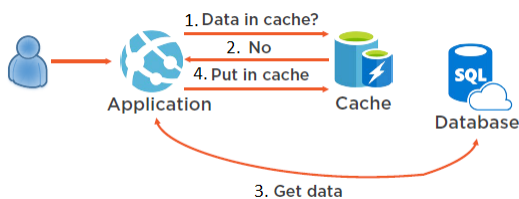
Now that your data is stored in a cache and a data store, you can run into problems when you try to make an update. Should you update both the cache and the data store? In the cache aside pattern, you invalidate the data in the cache and then make the changes to the data source directly.
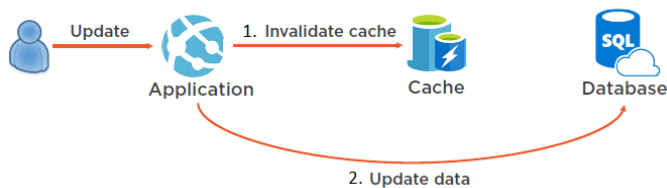
Content Cache Pattern

Cache static content (images, templates, style sheets). Content caching reduces the load on your server, thereby increasing application performance (see Redis Output Provider for ASP.NET).

User Session Cache Pattern

User sessions can be stored in Redis Cache to maintain application state between browser sessions (for example: shopping cart). Traditionally this is done with session cookies or local storage API, but these have limitations such as limited storage and slow performance. Instead, we can use a key and store the data in Redis Cache.

Job and Message Queuing Pattern

Applications often use queues to handle long-running tasks. Long running tasks can negatively affect application performance, so you can offload them onto a queue where they will run in sequence. This can improve application performance and scalability.

Distributed Transactions Pattern

A transaction against a back-end data store runs as a single atomic operation. Redis cache allows you to execute a batch of commands as a single transaction.

4.2 Instrument solutions to support monitoring and logging

- configure an app or service to use Application Insights
- analyze and troubleshoot solutions by using Azure Monitor
- implement Application Insights Web Test and Alerts

See text book page 219 Skill 4.2: Instrument solutions to support monitoring and logging

Using Application Insights
Skill: configure an app or service to use Application Insights

https://docs.microsoft.com/en-us/azure/azure-monitor/app/usage-overview

Application Insights is an extensible application performance management (APM) service. Application Insights is part of Azure Monitor. You can monitor your application while it is running by using Application Insights. Measurements, known as *telemetry*, are automatically sent to Application Insights. Using these telemetry streams, you can analyze your application's performance and create alerts and dashboards to help you better understand how your application is behaving.

Application insights can monitor the following:

- Request/Response data
- Exceptions
- Page views and page loads
- AJAX calls
- User and session counts
- Host diagnostics

- Trace logs
- Metrics and performance counters
- Custom events

You send messages from your code to Application Insights by using the *TelemetryClass* class. You can also send *log messages* to Application Insights by using the integration between System.Diagnostics and Application Insights. Any message sent to the diagnostics system using the Trace class appears in Application Insights as a Trace message. You can use the TraceException() method for sending the stack trace and the exception to Application Insights.

To configure Application Insights, first we create a new Log Analytics workspace, and then we create an Application Insights resource. The Log Analytics workspace serves as a logical storage for your logs. Next, when you create the Application Insights resource, you set a property that links it to the Log Analytics workspace that was just created.

+Create a resource > Log Analytics workspace > Create > fill in details (subscription, resource group, etc.) > Review+create
+Create a resource > Application Insights > Create > fill in details including Log Analytics workspace > Review+create

When created, Application Insights has an *Instrumentation Key* property that we use to connect from our SDK code:
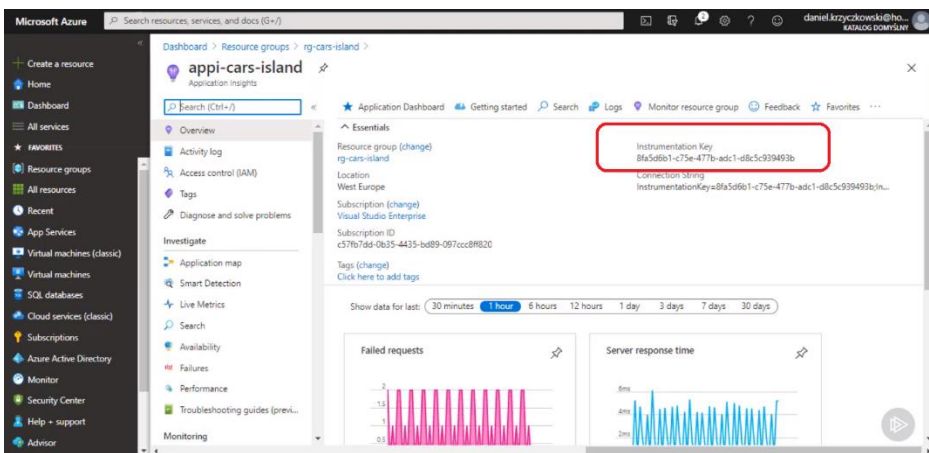
From ASP.NET Core, add NuGet package *Microsoft.ApplicationInsights.AspNetCore* . Then in the .NET Core *Startup* class, in the *ConfigureServices*() method, we add Application Insights telemetry using the *AddApplicationInsightsTelemetry()* method:
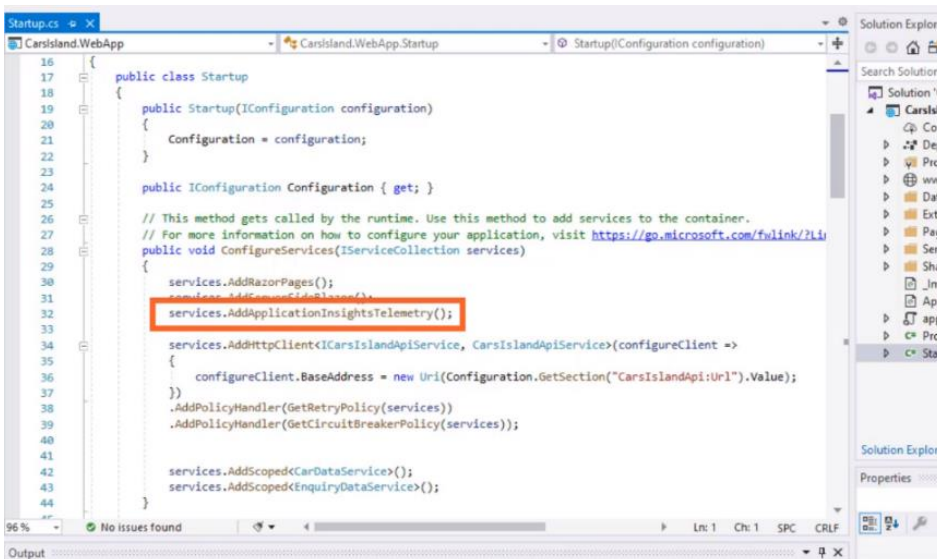
This one line of code enables collecting telemetry data in our ASP.NET Core web application. Now we provide the instrumentation key from the Application Insights workspace. We add the instrumentation key to the appSettings.json file:
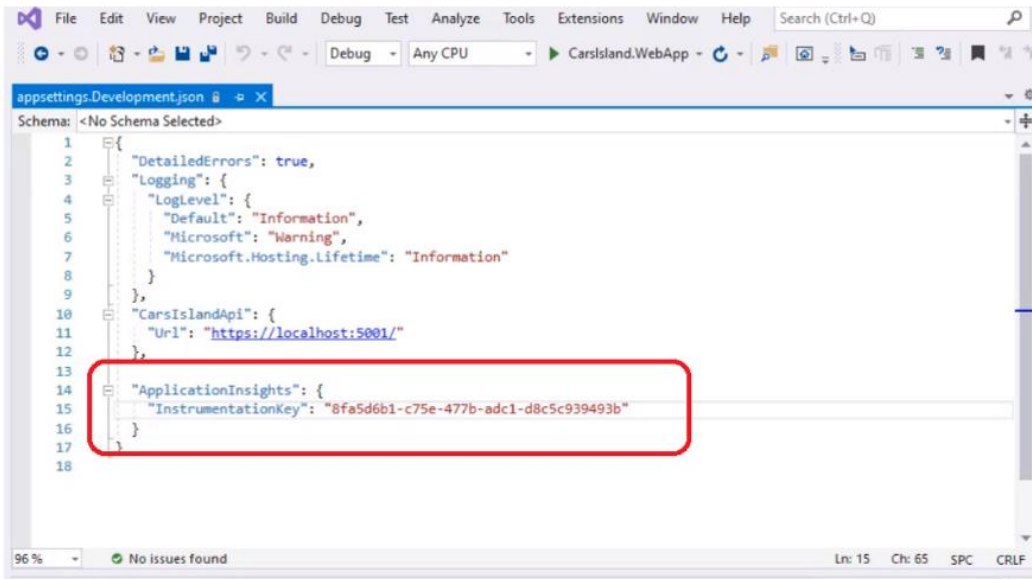
After this, every issue and exception thrown in the application will be reported automatically by the SDK (we can of course customize this from code to be more selective if we wish). To see the captured information, we go to the Portal > Application Insights > Overview > Search.
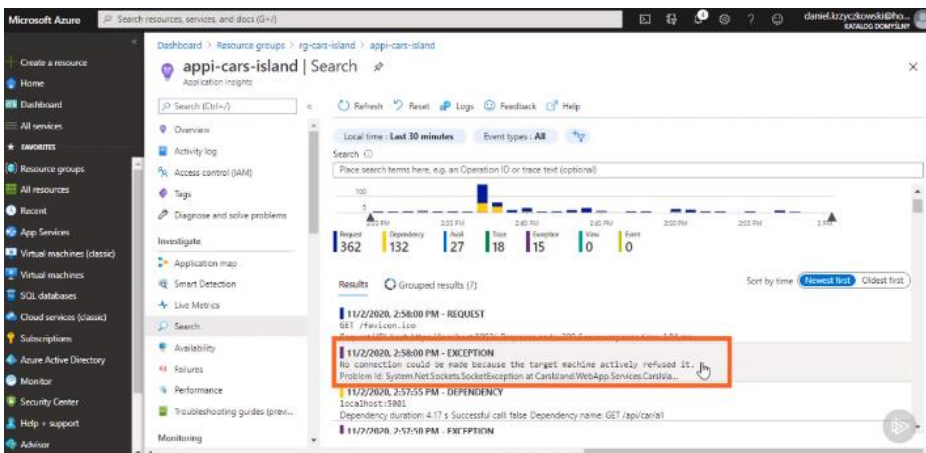
We can select the exception that was thrown in order to drill down for more details.

EXAM TIP: Remember that Application Insights is a solution for monitoring the behavior of an application on different platforms, written in different languages. There is no requirement to run your application in Azure. You only need to use Azure for deploying the Application Insights resource that are used by your application.

Application Insights Usage Demographics

Explore usage demographics to find out when people use your app, what pages they're most interested in, where your users are located, what browsers and operating systems they use.

Users   The Users feature is about how many people used your app and its features.

Funnels	The progression through a series of steps in a web application is known as a funnel.
Impact	The impact feature shows how load time and other properties of particular part of your app affect the usage for other parts of your app.
Retention	how many users return to your app, and how often they perform particular tasks or achieve goals.
User Flows	visualizes how users navigate between the pages and features of your site. How do users navigate away from a page? What do users click on a page? Are there places where users repeat the same action over and over?

Azure Monitor
Skill: analyze and troubleshoot solutions by using Azure Monitor
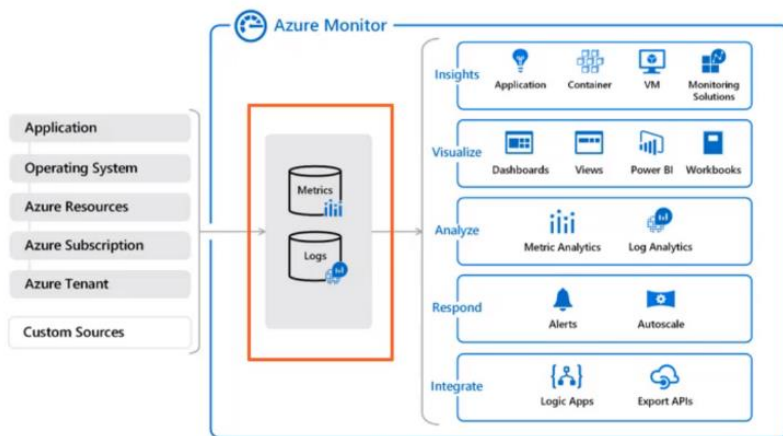
Azure Monitor is the central location for your log files, and metrics so you can run queries on them and create graphs. Depending on the information that you need to analyze, you can use Metric Analytics or Log Analytics. A metric is a numeric value at a particular point in time of your solution such as CPU usage or free memory; you can also create your own custom metrics. Because metrics are lightweight, they can monitor scenarios in near real-time. Logs are events that occurred within the system. They may be structured or free form text with a time stamp.

You use Log Analytics for analyzing the trace, logs, events, exceptions, etc. Log messages are more complex than metrics because they contain more information than a simple numeric value. You can analyze log messages by using queries to get information from the data stored in Azure Monitor.

Using Azure Monitor, you can set alerts based on the value of different metrics or logs.
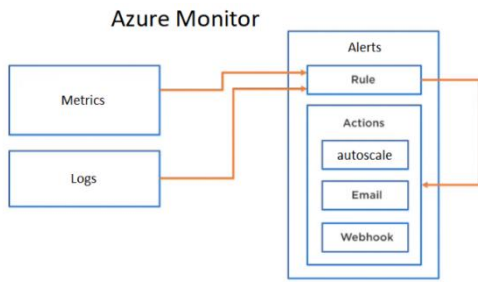
EXAM TIP: When you try to query logs from the Azure Monitor, remember that you need to enable the diagnostics logs for the Azure App Services.

Application Insights Web Tests and Alerts
Skill: implement Application Insights Web Test and Alerts

You can configure different types of tests for checking the availability of your web application.

- **URL ping test** – simple test for checking whether an endpoint URL in your application is available.
- **Multi-step web test** – A recording of a sequence of web requests, which can be played back to test more complex scenarios.
- **Custom test to track availability** You can create your own availability test in your code using the *TrackAvailability*() method from the SDK.

You can then configure Alerts from Application Insights when one of the tests fail.

Alerts use Action Groups. An Action Group is a collection of notification preferences. Azure Monitor uses action groups to notify you that an alert has been triggered. The alerts can be configured to do things like send emails. Each action has the following properties: type, name, action.

Setup availability test
Go to Portal > Application Insights > Investigate > Availability > +Add test > fill in details > Create
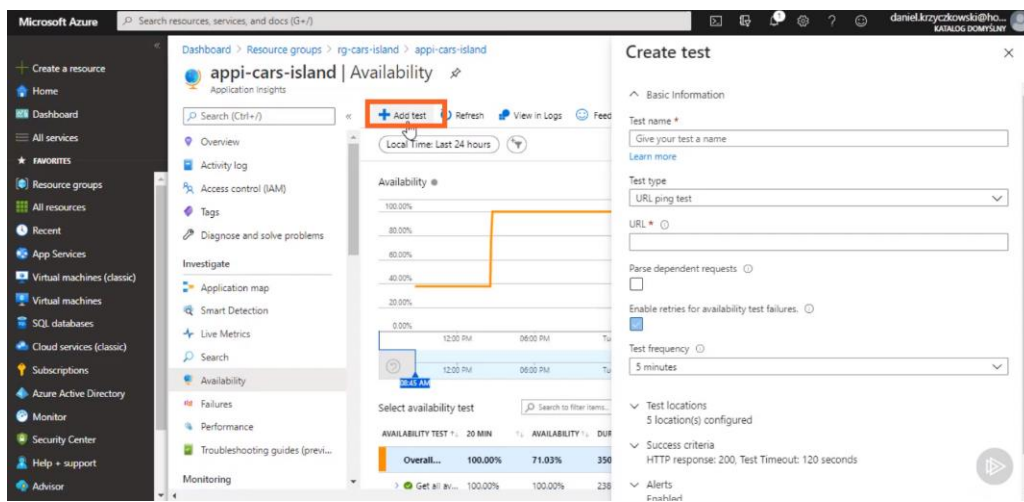
Create action group

Go to portal > Application Insights > Rules Management > +Add action group > Save changes
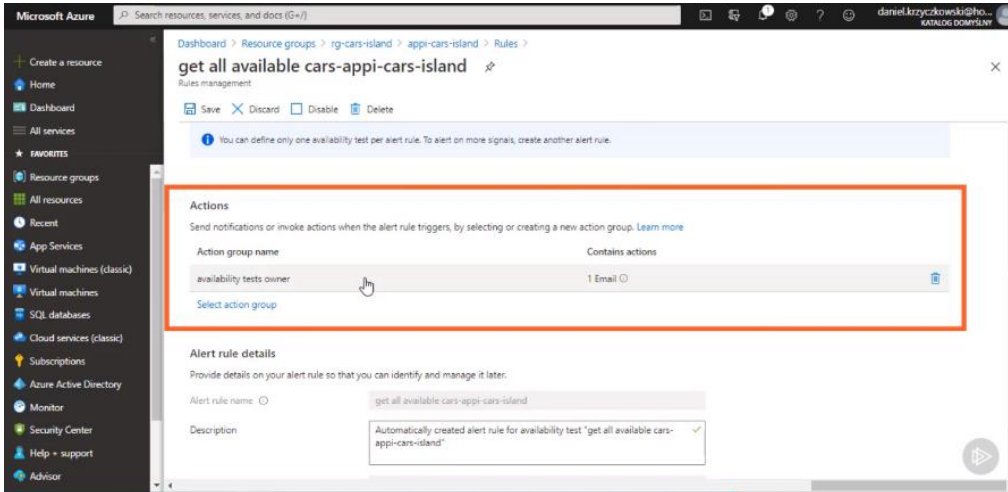
Receive and investigate alert

For email alert types, an alert email will be sent to the configured email recipient:
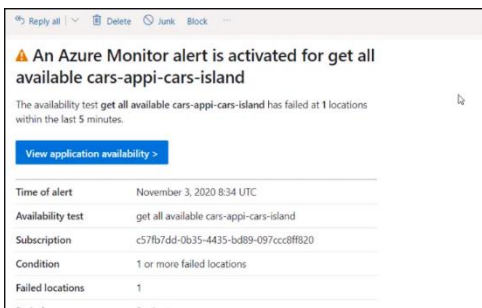
5. Azure services and third-party services (15-20%)

This chapter discusses connecting and consuming Azure services and third-party services.

5.1 Develop an App Service Logic App – *Out of scope*

5.2 Implement API Management

- create an APIM instance
- configure authentication for APIs
- define policies for APIs
- import OpenAPI definitions

https://docs.microsoft.com/en-us/azure/api-management/

APIM Definitions
https://docs.microsoft.com/en-us/learn/modules/build-serverless-api-with-functions-api-management/

https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts

API Management (APIM)
Azure API Management is a way to create consistent and modern gateways for existing back-end services. It is positioned between your APIs and the internet, and provides secure, scalable API access for your applications. It also enables you to construct an API from a set of disparate microservices. It consists of the following components:

- API gateway – accepts API calls and routes them to your backends, accessible at https://<name>.azure-api.net
- Azure portal – the administrator interface where you set up your API program
- Developer portal – web user interface for developers, accessible at https://<name>.developer.azure-api.net

API Gateway
An Azure API gateway is an instance of the Azure API Management service. It provides a single surface of communication for client web apps.
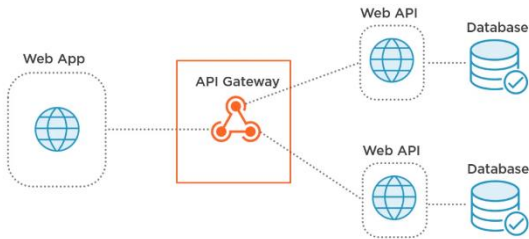
API

a way to interface with an application without understanding the complexity.

Developer Portal

where developers can read API documentation, get API keys, try out API via interactive console, and access analytics

API Products

are how APIs are surfaced to developers. A product is a collection of APIs that make up a single application that developers can gain access to. Products have one or more APIs, a title, a description, and terms of use.

API user Groups

are used to manage the visibility of API Products to developers. There are three kinds of API Groups: Administrators, developers, guests. Each developer is a member of one or more developer groups.

Create an APIM instance
Skill: create an APIM instance

https://docs.microsoft.com/en-us/azure/api-management/quickstart-arm-template

From CLI:

```
az apim create --name myapim --resource-group myResourceGroup \
  --publisher-name Contoso --publisher-email admin@contoso.com \
  --no-wait
```

From Portal:

Create a resource > Integration > API Management > fill in details > Create > Review and create

From Powershell:

```
New-AzApiManagement -Name "myapim" -ResourceGroupName "myResourceGroup" `
  -Location "West US" -Organization "Contoso" -AdminEmail admin@contoso.com
```

From VS Code:

Azure Extension > Sign in to Azure > Right-click on an Azure subscription > Create API Management in Azure.

From an ARM template:

azuredeploy.json

```json
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      "publisherEmail": {
        "type": "string",
        "minLength": 1,
        "metadata": {"description": "The email address of the owner of the service"}
      },
      "publisherName": {
        "type": "string",
        "minLength": 1,
        "metadata": {"description": "The name of the owner of the service"}
      },
      "sku": {
        "type": "string",
        "defaultValue": "Developer",
        "allowedValues": ["Developer", "Standard", "Premium"],
        "metadata": {"description": "The pricing tier of this API Management service"}
      },
      "skuCount": {
        "type": "string",
        "defaultValue": "1",
        "allowedValues": ["1", "2"],
        "metadata": {"description": "The instance size of this API Management service."}
      }
    },
    "resources": [
      {
        "type": "Microsoft.ApiManagement/service",
        "apiVersion": "2019-12-01",
        "name": "myName",
        "location": "EASTUS",
        "sku": {
          "name": "[parameters('sku')]",
          "capacity": "[parameters('skuCount')]"
        },
        "properties": {
          "publisherEmail": "[parameters('publisherEmail')]",
          "publisherName": "[parameters('publisherName')]"
        }
      }
    ]
  }
```

azuredeploy.parameters.json

```json
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      "publisherEmail": {"value": "elvis@example.com"},
      "publisherName": {"value": "Elvis"}
    }
}
```

Using the azure deployment ARM template and parameter file above, the APIM instance is created by deployment to a resource group as follows:

```powershell
# powershell example
New-AzResourceGroupDeployment `
  -Name myDeployment `
```

```
 -ResourceGroupName myResourceGroup `
 -TemplateFile "./azuredeploy.json" `
 -TemplateParameterFile "./azuredeploy.parameters.json"
```

Skill: configure authentication for APIs

API products can be Open or Protected. Protected products must be subscribed to before they can be used. Subscription approval is configured at the product level.

API's can also be protected from unauthorized web client calls. This is done through subscription keys or through certificates.

Access keys can be obtained by developers through the developer portal by going to the APIM Account details and subscriptions page:
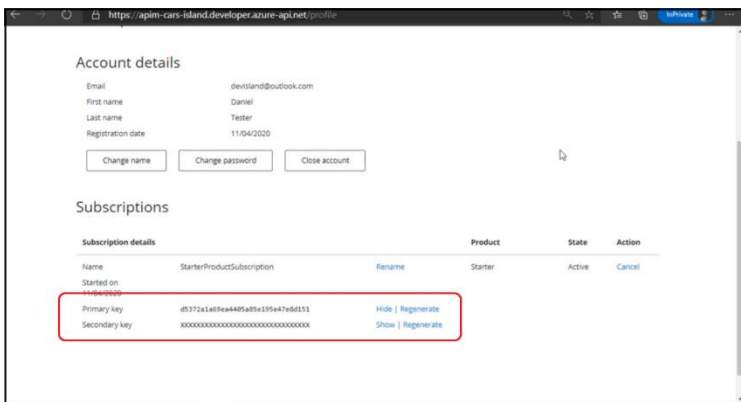
The subscription key is used from web client requests by including it with the "Ocp-Apim-Subscription-Key" HTTP header. Here is how you can make an example call with a subscription key using the Postman utility:
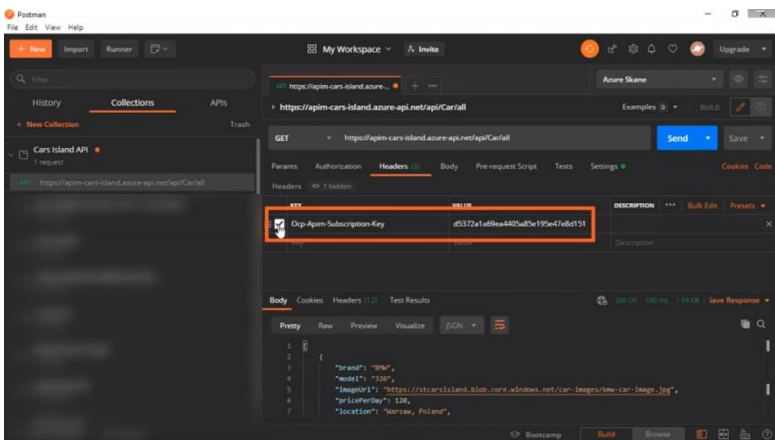
Another way to protect APIs is with certificates. Certificates can be used to provide mutual TLS authentication between the client and the API gateway. With APIM you configure the gateway to only allow requests with certificates containing a specific thumbprint. We can configure APIM to accept client certificates as follows:

Go to Azure Portal > API Management service > Security > Certificates > +Add

Under Client certificate pane, provide an id for the certificate; upload the certificate file; provide a password; click the Add button.
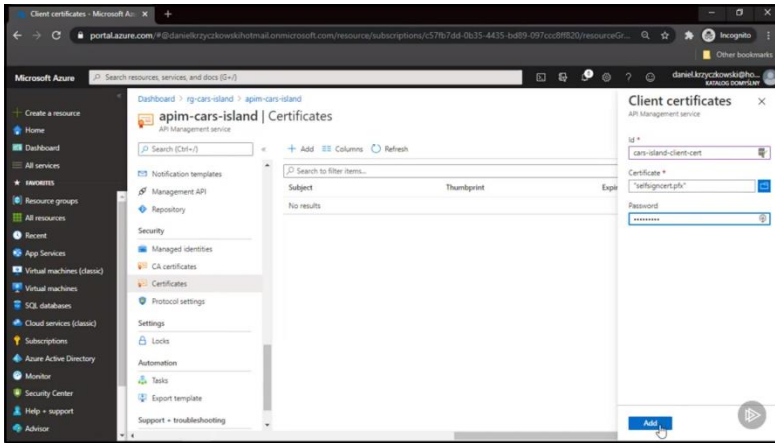
When the certificate is successfully uploaded, APIM will create a thumbprint that we need to copy and use from the client.
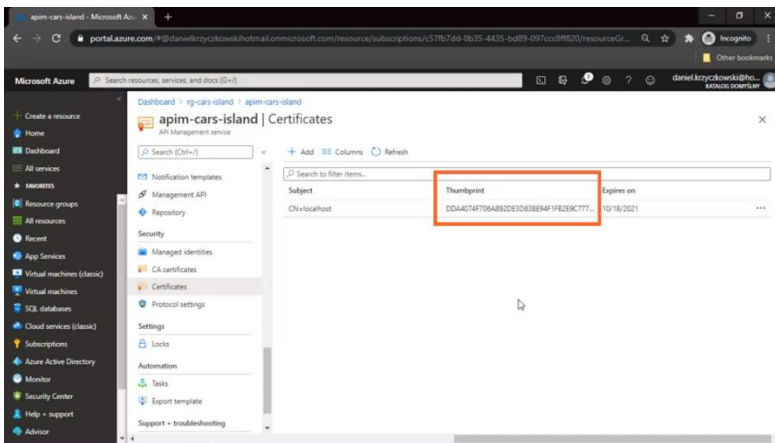
We add the thumbprint to the inbound policy:

Then in the Portal under the API policies, we paste in the code and click Save:
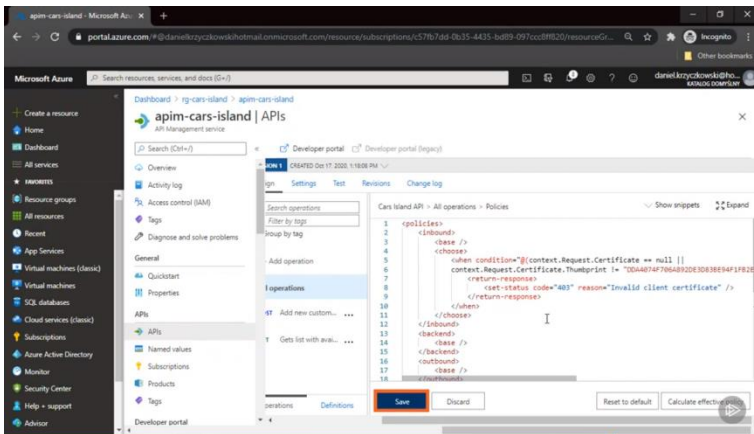
Now when the client calls the API it must provide the certificate and the password.

Define APIM Policies
Skill: define policies for APIs

https://docs.microsoft.com/en-us/learn/modules/improve-api-performance-with-apim-caching-policy/

https://docs.microsoft.com/en-us/learn/modules/protect-apis-on-api-management/

https://docs.microsoft.com/en-us/learn/paths/architect-api-integration/

https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-policies

https://docs.microsoft.com/en-us/azure/api-management/api-management-policies

APIM Policies are a capability of Azure API Management that allow changing the behavior of the API through configuration, thereby generating consistent behavior across disparate API microservices in a single API Product. Commonly used APIM policies include: access restriction, authentication and JWT handling, caching, cross-domain, data transformation

APIM policies are a collection of statements that are executed sequentially on the request or response of an API. They can perform things like: convert format from XML to JSON, restrict the number of incoming calls, enforce existence and/or value of HTTP header, cache responses according to cache control configuration.

APIM policies can be applied to different scopes:

- *Global* scope affects all APIs within the APIM instance;
- *Product* scope manages access to the product as a single entity;
- *API* scope affects only a single API;
- *Operation* scope affects only one operation within the API.

They will be applied in this order unless overridden by the `<base />` tag in the policy XML file.

There are 4 kinds of policy execution:

- *Inbound* policies execute when a request is received.
- *Backend* policies execute before a request is forwarded to a managed API.
- *Outbound* policies execute before a response is sent.
- *On-Error* policies execute when an exception is raised.

Polices are defined using XML format.

```xml
<policies>
  <inbound>
    <!-- statements to be applied to the request go here -->
  </inbound>
  <backend>
    <!-- statements to be applied before the request is forwarded to
         the backend service go here -->
  </backend>
  <outbound>
    <!-- statements to be applied to the response go here -->
  </outbound>
  <on-error>
    <!-- statements to be applied if there is an error condition go here -->
  </on-error>
</policies>
```

Example 1:

```xml
<policies>
  <inbound>
    <rate-limit calls="5" renewal-period="10" />
    <cache-lookup vary-by-developer="false" vary-by-developer-groups="false" must-revalidate="true" downstream-caching-type="none" caching-type="internal" />
    <base />
  </inbound>
  <backend>
    <base />
  </backend>
  <outbound>
    <cache-store duration="60" />
    <base />
  </outbound>
  <on-error>
    <base />
  </on-error>
</policies>
```

Example 2 – caching:

To set up a cache, you use an outbound policy named cache-store to store responses. You also use an inbound policy named cache-lookup to check if there is a cached response for the current request.

Suppose, for example, that the board games Stock Management API received a GET request to the following URL and cached the result:

```
http://<boardgames.domain>/stock/api/product?partnumber=3416&customerid=1128
```

This request is intended to check the stock levels for a product with part number 3416. The customer ID is used by a separate policy as does not alter the response. Subsequent requests for the same part number can be served from the cache, as long as the record has not expired.

Now suppose that a different customer requests the same product:

```
http://<boardgames.domain>/stock/api/product?partnumber=3416&customerid=5238
```

By default, the response can't be served from the cache, because the customer ID is different. However, requests for the same product from different customers could be returned from the cache if we vary by partnumber.

```
<policies>
    <inbound>
        <base />
        <cache-lookup>
            <vary-by-query-parameter>partnumber</vary-by-query-parameter>
        </cache-lookup>
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <cache-store duration="60" />
        <base />
    </outbound>
    </on-error>
        <base />
    </on-error>
</policies>
```

Example 3 – http headers:

A common http header found in the response from an API is x-powered-by. This header exposes information about the platform technology being used, which could be exploited by an attacker.
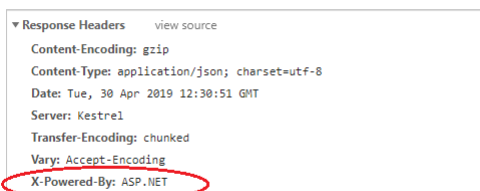
To remove this header, we use the `set-header` policy. We replace the default `<outbound>` tag in the APIM policies file with this:

```
<outbound>
    <set-header name="X-Powered-By" exists-action="delete" />
    <base />
</outbound>
```

Import OpenAPI definitions
Skill: import OpenAPI definitions

https://swagger.io/docs/specification/about/

https://docs.microsoft.com/en-us/learn/modules/improve-api-developer-experience-with-swagger/

Importing OpenAPI Definitions | SwaggerHub Documentation (smartbear.com)

https://youtu.be/pRS9LRBgjYg

https://docs.microsoft.com/en-us/azure/api-management/import-and-publish

https://soltisweb.com/blog/detail/2020-08-19-importingopenapiapiintoazureapim

https://docs.microsoft.com/en-us/azure/api-management/import-api-from-oas

OpenAPI is a description for REST APIs. It comes with a YAML (or JSON) definition file that follows a standardized format for describing the inputs and outputs, where the API is hosted (endpoints), operations, and what authorization is required to access it. The *OpenAPI Specification* defines how to describe a REST API. An OpenAPI enables you to understand and use a REST API.

OpenAPI comes with some standard tools:

- API validator – checks whether the API confirms to industry standards
- API document generator – generates API reference documentation
- API SDK generator – generates an SDK in the language of your choice

Swagger is an open-source set of tools to design, build, document, and use RESTful web services. It includes automated documentation, code generation, and test-case generation. OpenAPI is a standard way to describe RESTful API's, while Swagger is a toolset built around that standard. Swashbuckle is an open-source Swagger implementation used for generating Swagger documentation for .NET Core APIs using .NET reflection.

Import an OpenAPI Using Azure Portal
1. In Azure portal, search for and select API Management services.

2. On the API Management services page, select your API Management instance.

3. API Management instance > +Add API > OpenAPI
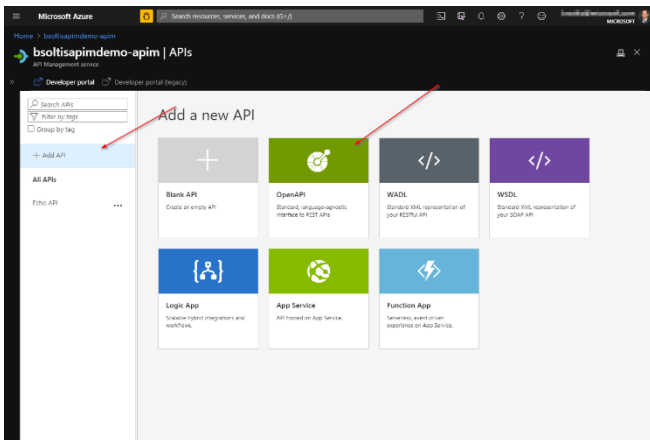
4. In the OpenAPI specification field either enter a URL or select and upload a file. The remaining fields will autofill from the specification.

5. Click the Create button.

Import an OpenAPI using PowerShell

The Azure APIM PowerShell commands allow you to import the OpenAPI via code. For example:

```
$ApiMgmtContext = New-AzApiManagementContext -ResourceGroupName "my-rg" -ServiceName "my-apim"

Import-AzApiManagementApi -Context $ApiMgmtContext `
        -SpecificationFormat "OpenApi" `
        -SpecificationPath "C:\Users\Myself\Downloads\my-swagger.yaml" `
        -Path "my-path"
```

PowerShell will return the newly created OpenAPI information.

5.3 Develop event-based solutions

- implement solutions that use Azure Event Grid
- implement solutions that use Azure Notification Hubs
- implement solutions that use Azure Event Hub

https://docs.microsoft.com/en-us/learn/paths/connect-your-services-together/

https://docs.microsoft.com/en-us/learn/modules/choose-a-messaging-model-in-azure-to-connect-your-services/

Comparing Events and Messages

Events	Messages
Lightweight notification of a state change	Application data from a source system to be consumed elsewhere
Publisher does not know (or care) how the message is handled	There is an expectation that a message will be handled by a receiver
Follows a publisher/subscriber model	Can follow either a publisher/subscriber or a producer/consumer model

Selecting an Event-based Service

1 Does your solution have an expectation of how data is handled or does it contain app data? If so, select a Messaging service.

2 Do you need a solution to send events to mobile devices as push notifications? Select Azure Notification Hub.

3 Does your solution produce discrete events, that report state changes that a system can act on? Select Azure Event Grid.

4 Does your solution report state over time for analysis by another system, such as in a data pipeline? Select Azure Event Hub.

Event Grid
Skill: implement solutions that use Azure Event Grid

https://docs.microsoft.com/en-us/learn/modules/react-to-state-changes-using-event-grid/

https://docs.microsoft.com/en-us/learn/modules/choose-a-messaging-model-in-azure-to-connect-your-services/4-choose-event-grid

Event Grid is a routing service that distributes events from different Azure sources, such as Blob storage accounts, to different Azure handlers, such as Azure Functions or Webhooks.

Event Grid is for discrete events that report state changes and are actionable. If something is listening (subscribed) for that event, it takes action when the event occurs. Subscribers can add filtering rules so only events get delivered that match the rules.
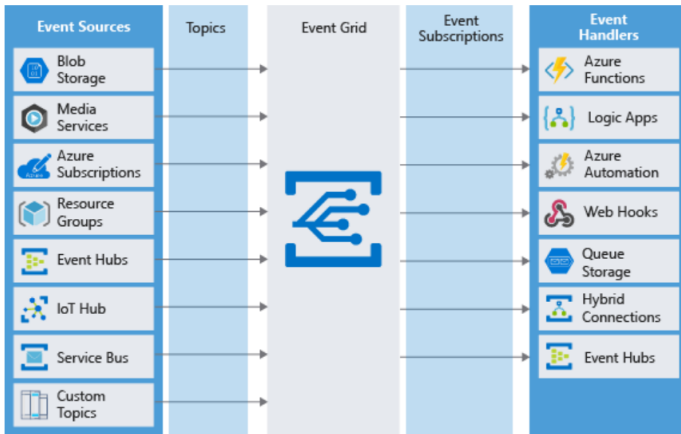
Event Grid terminology

- **Events**: What happened.
- **Event sources**: Where the event took place.
- **Topics**: The endpoint where publishers send events.
- **System topics:** built-in topics provided by Azure services, such as a blob storage service.
- **Custom topics:** are applications and third-party topics.
- **Event subscriptions**: The endpoint or built-in mechanism to route events, sometimes to multiple handlers. Subscriptions are also used by handlers to filter incoming events intelligently.
- **Event handlers**: The app or service reacting to the event.

Workflow

1. Create topic
2. Configure publisher application to send events to that topic
3. Add a subscriber application to the topic

Every event that is sent to a handling application will have a data property containing a message. The message has event specific information that was provided by the publisher.

Event Grid needs to be enabled. Instead of providing a service like an Azure Function, Event grid is enabled at the subscription level, by registering a resource provider. This way other services can send events to it. For example (CLI):

```
az provider register --namespace Microsoft.EventGrid
az provider show --namespace MicrosoftEventGrid --query "registrationState"
```

Azure Notification Hubs (ANH)
Skill: implement solutions that use Azure Notification Hubs

https://docs.microsoft.com/en-us/azure/notification-hubs/notification-hubs-push-notification-overview

Also see textbook p 287
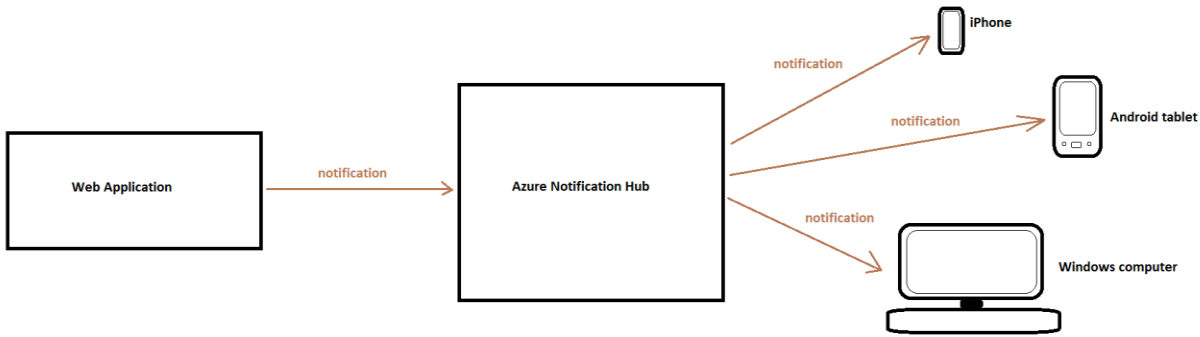
Notification Hub Fundamentals

Notification Hub is for *user-notification* type events (example: events that prompt user or their device for attention). Unlike the other 2 types of events that go from application to application, these events are meant to get the attention of the end user of the application, although there is also a *push-notification* (server initiated) that is meant only for the device the application is running on – an example of push-notification is when your phone buzzes to let you know you have new email.

Namespaces organize hubs. A Namespace is a collection of Notification Hubs. Generally, you create one namespace per application. Within a namespace you create one notification hub per environment (prod, dev, test). Credentials are supplied at the namespace level. Billing is done at the namespace level.

Azure Notification Hubs provide an easy-to-use and scaled-out push engine that enables you to send notifications to any platform (iOS, Android, Windows, etc.) from any back-end (cloud or on-premises). Azure Notification Hubs abstract the complexities of the different delivery patterns for different notification target platforms.

Workflow

1. Setup vendor specific platform notification service for each platform you want to send notifications to
2. Setup notification hub instance through Azure portal (or CLI, etc.)
3. Map credential keys obtained from platform notification service to notification hub
4. Register devices – can be done with .NET SDK through a web API backend
5. Send pushes – can be done with cross-platform .NET SDK via web API

Event Hub
Skill: implement solutions that use Azure Event Hub

Fundamentals

Event Hubs can handle Volume, Variety, and Velocity. Event Hubs is an event ingestion and processing service with time-retention buffer at the big data scale. Being highly scalable it's a good fit for Big Data scenarios such as IoT. Much like Event Grid, Event Hub decouples the producer and consumer of events. Unlike Event Grid, you can integrate Event Hubs with both Azure and non-Azure services. Event Hub can capture events directly to Azure Blob Storage or Data Lake as events come in.

Event Hub is for series-type events that report a condition and are time-ordered, and analyzable (example: telemetry).

Event Hubs is an intermediary for the publish-subscribe communication pattern. Unlike Event Grid, however, it is optimized for extremely high throughput, a large number of publishers, security, and resiliency. Event Hubs can handle events that exist outside of Azure.

You need to create an Event Hub namespace before creating the Event Hub:

```
az eventhubs namespace create --name <NAME> /
    --resource-group <NAME> /
    --location <LOCATION> /
    --sku <Basic|Standard>
```

Once the namespace is created, you can create the Event Hub:

```
az eventhubs eventhub create --name <NAME> /
    --namespace <NAME> /
    --message-retention 3 /
    --partition-count 4 /
    -g <GROUP NAME>
```

Event Hub Terminology

Namespace is a container for one or more Event Hubs. It serves as a scoping container. Any options applied to the namespace will apply to all of the Event Hubs contained within it (such as pricing). Namespaces have a pre-purchased unit of capacity, called *throughput units*, that controls the throughput of events.

Event producers are the applications that send data to the Event Hubs.

Partitions are buckets for the event messages. Event Hub event messages get delivered, in order, to a partition.

Partitions

Like a bucket for event messages

Hold events time-ordered as they arrive

Events not deleted once read

Event Hubs decides which partition events are sent to
- Can specify partition with partition key

Maximum 32 partitions

Create as many as expected concurrent subscribers

Consumer groups are nothing more than a particular view of the Event Hub. It has things like the current position where an application is. With multiple consumer groups, Event Hubs can support multiple applications reading through the events all at their own pace.

Subscribers are the consumers or the applications that read the events contained within the Event Hub.

Workflow: Send Events to Event Hub

1. Install .NET SDK for Event Hubs: Azure.Messaging.EventHubs
2. Obtain connection info – endpoint will include key used to authorize to the Event Hub. Can be obtained from
   Azure Portal > Event Hubs Namespace > Settings > Shared access policies
3. Open connection using EventHubProducerClient – good idea to cache this object
4. Prepare data – convert to binary representation – *JsonSerializer.SerializeToUtf8Bytes()*
5. Send data – single or batch of events – you may specify a partition

Workflow: Read Events from Event Hub

1. Install .NET SDK for Event Hubs: Azure.Messaging.EventHubs
2. Obtain connection info.
3. Open connection using EventHubConsumerClient or EventProcessorClient
4. Retrieve data
5. Decode data – deserialize from binary back to your class – *JsonSerializer.Deserialize<MyClass>()*
6. Finally, close the connection to the Event Hubs

5.4 Develop message-based solutions

• implement solutions that use Azure Service Bus
• implement solutions that use Azure Queue Storage queues

https://docs.microsoft.com/en-us/learn/paths/connect-your-services-together/

Messages
In general terms, a *message* is raw data produced by a service with the goal of being stored or processed elsewhere. This means that the publisher of the messages has an expectation of some other system or subscriber process the message. Because of this expectation, the subscriber needs to notify the publisher about the status of the message. The defining characteristic of a message is that the overall integrity of the application may rely on messages being received. The sender and receiver of a message are often coupled by a strict data contract.

Choosing a messaging technology:

1. Is the communication an event? If so, consider using Event Grid or Event Hubs.
2. Should a single message be delivered to more than one destination? If so, use a Service Bus topic. Otherwise, use a queue.

https://docs.microsoft.com/en-us/learn/modules/implement-message-workflows-with-service-bus/2-choose-a-messaging-platform

Events and Queues

An *event* triggers a notification that something has occurred. Events are "lighter" than messages and are most often used for broadcast communications.

- The event may be sent to multiple receivers, or to none at all
- Events are often intended to have a large number of subscribers for each publisher (fan out)
- The publisher of the event has no expectation about the action a receiving component takes

Service Bus and Queue Storage are designed to handle messages. If you want to send events, you would likely choose Event Grid or Event Hub.

A *queue* is a simple temporary storage location for messages. A sending component adds a message to the queue. A destination component picks up the message at the front of the queue.

Queues decouple the source and destination components to insulate destination components from high demand.

Benefits of message queuing: encourages application logic modularity, and enables fault tolerance between modules.

Service Bus
Skill: implement solutions that use Azure Service Bus

https://docs.microsoft.com/en-us/learn/modules/implement-message-workflows-with-service-bus/

Azure Service Bus is an enterprise-level message broker that allows different applications to communicate with each other in a reliable way. A message is raw data that an application sends asynchronously to the broker to be processed by another application connected to the broker. The message can contain JSON, XML, or text information. Service bus messages are ordered (FIFO).
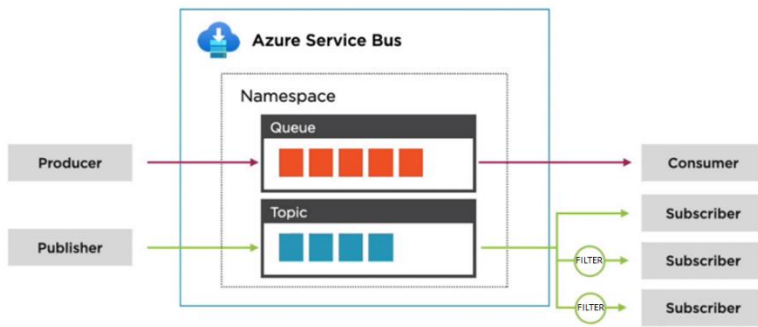
Namespace is a container for all components of the messaging. A single service bus namespace can contain multiple queues and topics. The different components of your solution connect to the topics and queues in the namespace.

Queue is a container of messages. As a new message arrives at the queue, the Service Bus service assigns a timestamp to the message. Queues are appropriate for point-to-point communication scenarios in which a single application needs to communicate with another single application.

Topic Topics are similar to queues but are used in publisher/subscriber models. The difference between queues and topics is that topics can have several applications receiving messages. When subscribed to a topic, the receiver can filter the messages.
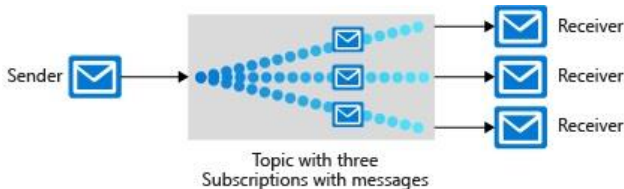
Filters Topic filters can be specified as:

- Boolean filters – all or none of the messages are selected
- SQL filters – SQL-like expression evaluated against message properties
- Correlation filters – matched against message properties and property values

Subscription This is basically a dedicated "queue" for a subscriber into a topic. It is ordered (FIFO).

Dead-letter Queue (DLQ) This enables you to capture messages that were not processed during their lifetime, and act accordingly with those messages.

To code against Service Bus, use the Microsoft.Azure.ServiceBus NuGet package. Start by creating an instance of QueueClient class both in sending and receiving components to or from the service bus.

To connect to a queue in a service bus, you need:

- • The URL of the Service Bus namespace and queue or topic, also known as an endpoint.
- • An access key.

Service Bus URL Structure
https://pluralsight.servicebus.windows.net/testqueue
namespace   queue or topic name

Interact with Service Bus using CLI

```
#create a queue
az servicebus queue create --name <NAME> --namespace-name <NAME> --resource-group <RG-NAME>

#delete a queue
az servicebus queue delete --name <NAME> --namespace-name <NAME> --resource-group <RG-NAME>

#create a topic
Az servicebus topic create --name <NAME> --namespace-name <NAME> --resource-group <RG-NAME>

#delete a topic
az servicebus topic delete --name <NAME> --namespace-name <NAME>

#create a subscription
az servicebus topic subscription create --name <NAME> --namespace-name <NAME> --topic-name <NAME>
```

Interacting using the Azure Portal

To create a Shared Access Policy choose:

Portal > Service Bus > Namespace > Settings > Shared access policy > +Add

In Add SAS Policy sidebar, enter Policy name, tick some boxes and click the Create button.
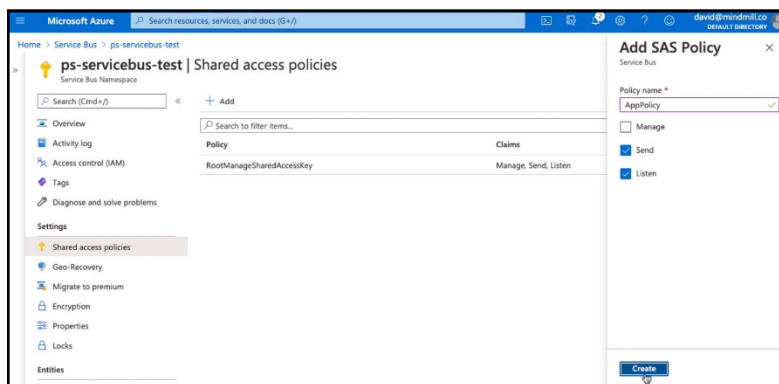
Then you will get connection strings:
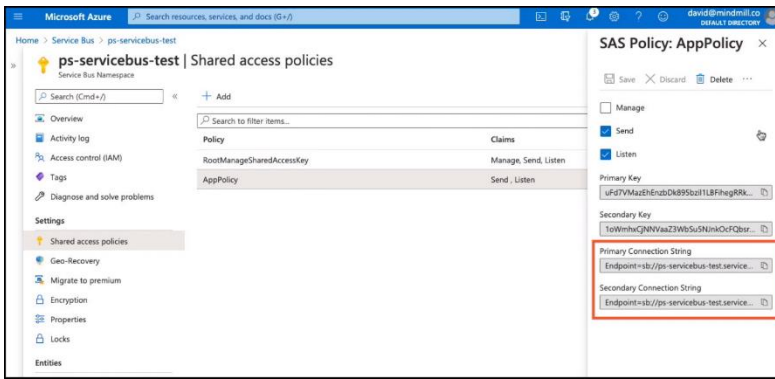
Copy the strings and use them in your code.

Interacting with .NET: To send a message to your Service Bus Queue:

```
using Microsoft.Azure.ServiceBus;

var myConnectionString = ConfigurationManager.Get("MyConnection");
var myQueueName = "MyQueue";
queueClient = new QueueClient(myConnectionString, myQueueName);

var encodedMessage = new Message(Encoding.UTF8.GetBytes("Hello, World!"));
await queueClient.SendAsync(encodedMessage);
await queueClient.CloseAsync();
```

To receive a message from your Service Bus Queue, you must first register a message handler - this is the method in your code that will be invoked when a message is available on the queue.

```
queueClient.RegisterMessageHandler(MessageHandler, messageHandlerOptions);
```

Then, within the message handler, call the QueueClient.CompleteAsync() method to remove the message from the queue.

```
public void MessageHandler()
{
        await queueClient.CompleteAsync(message.SystemProperties.LockToken);
        await queueClient.CloseAsync();
}
```

To allow multiple components to receive the same message, use an Azure Service Bus topic.

To send a message to the Service Bus Topic:

```
using Microsoft.Azure.ServiceBus;
topicClient = new TopicClient(myConnectionString, myTopicName);
var encodedMessage = new Message(Encoding.UTF8.GetBytes("Hello, World!"));
await topicClient.SendAsync(encodedMessage);
```

To receive a message from the Service Bus Topic Subscription, create a subscription object and then register a message handler

```
subscriptionClient = new SubscriptionClient(myConnectionString, myTopicName, mySubscriptionName);
subscriptionClient.RegisterMessageHandler(MessageHandler, messageHandlerOptions);
```

Then within the message handler, call CompleteAsync() to remove the message from the queue.

```
Public void MessageHandler()
{
        await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);
}
```

Azure Queue Storage queues
Skill: implement solutions that use Azure Queue Storage queues

https://docs.microsoft.com/en-us/learn/modules/communicate-between-apps-with-azure-queue-storage/

A storage queue is a high-performance message buffer that can act as a broker between the front-end components (the "producers") and the middle tier (the "consumer").
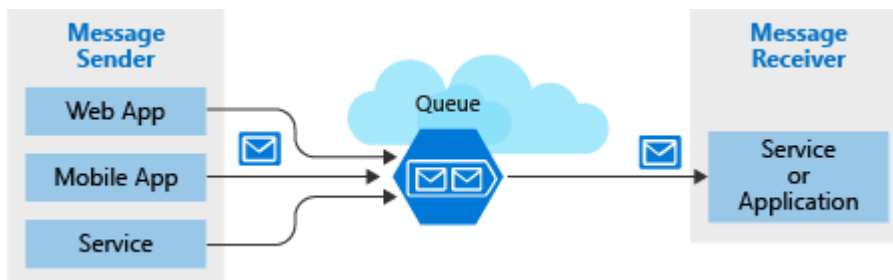
A queue increases resiliency by temporarily storing waiting messages. At times of high demand, the queue may increase in size, but messages are not lost. The destination component can catch up and empty the queue as demand returns to normal. This makes queue storage useful for critical business data that would be damaging to lose.

A message in a queue is a byte array of up to 64 KB. Message contents are not interpreted at all by any Azure component. If you want to create a structured message, you could format the message content using XML or JSON. Your code is responsible for generating and interpreting your custom format. Unlike Service Bus, Storage Queue messages <u>are not</u> ordered.

A queue is part of a storage account. When you create a storage account that will contain queues, you should consider the following settings:

- Queues are only available as part of Azure general-purpose storage accounts (v2). You cannot add them to Blob storage accounts.
- You should choose a location that is close to either the source components or destination components or (preferably) both.
- Data is always replicated to multiple servers. You have a choice of replication strategies: Locally Redundant Storage (LRS) is low-cost but vulnerable to disasters that affect an entire data center while Geo-Redundant Storage (GRS) replicates data to other Azure data centers.
- Require *secure transfer* if sensitive information may pass through the queue. This setting ensures that all connections to the queue are encrypted using Secure Sockets Layer (SSL).

To access a queue, you need three pieces of information:

- Storage account name
- Queue name
- Authorization token

This information is used by both applications that talk to the queue. The combination of your storage account name and your queue name uniquely identifies a queue. Every request to a queue must be authorized. There are several options to choose from: Azure AD, shared key, shared access signature (SAS).

Queue URL Structure

You access a queue using a REST API. To do this, you'll use a URL that combines the name you gave the storage account with the domain queue.core.windows.net and the path to the queue you want to work with. The Azure Storage Client Library for .NET is a library provided by Microsoft that formulates REST requests and parses REST responses for you. The client library uses a connection string to establish your connection. Your connection string is available in the Settings section of your Storage Account in the Azure portal, or through the Azure CLI and PowerShell. A connection string is a string that combines a storage account name and account key and must be known to the application to access the storage account.

The sender creates the queue and adds a message. The receiver retrieves a message, processes it, and then deletes the message from the queue.
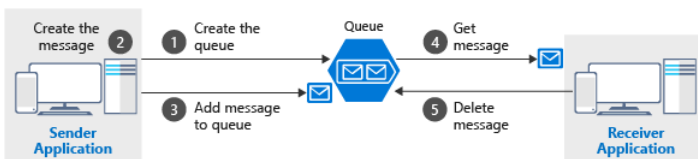
Notice that get and delete are separate operations. This arrangement handles potential failures in the receiver and implements a concept called at-least-once delivery. After the receiver gets a message, that message remains in the queue but is invisible for 30 seconds. If the receiver crashes or experiences a power failure during processing, then it will never delete the message from the queue. After 30 seconds, the message will reappear in the queue and another instance of the receiver can process it to completion.

Interacting with queue storage using CLI:

```
#create a queue
az storage queue create --name <NAME>

#delete a queue
az storage queue delete --name <NAME>

#view message (without affecting visibility)
az storage message peek --queue-name <NAME>

#get message
az storage message get --queue-name <NAME>

#delete a message
az storage message delete --queue-name <NAME> --id <MESSAGE-ID> --pop-receipt <RECEIPT-ID>

#delete all messages
az storage message clear --queue-name <NAME>
```

Interacting using .NET:

.NET packages via NuGet: Azure.Storage.Queues

To connect to a queue, you first create a CloudStorageAccount with your connection string. The resulting object can then create a CloudQueueClient, which in turn can open a CloudQueue instance.

```
CloudStorageAccount account = CloudStorageAccount.Parse(connectionString);
CloudQueueClient client = account.CreateCloudQueueClient();
CloudQueue queue = client.GetQueueReference("myqueue");
await queue.CreateIfNotExistsAsync();
var message = new CloudQueueMessage("your message here");
await queue.AddMessageAsync(message);
```

Creating a CloudQueue doesn't necessarily mean the actual storage queue exists. However, you can use this object to create, delete, and check for an existing queue.

In the receiver, you get the next message, process it, and then delete it after processing succeeds.

```
CloudQueue queue;
CloudQueueMessage message = await queue.GetMessageAsync();
if (message != null)
{
    // Process the message
    await queue.DeleteMessageAsync(message);
}
```

Queue Storage or Service Bus?

Use Case	Service Bus	Queue Storage
Individual message > 64 KB	✓	
Guarantee message ordering (*FIFO*)	✓	
topics (*one-to-many*)	✓	
Transactions	✓	
Duplicate detection	✓	
Receice messages without polling (AMQP 1.0)	✓	
at-least-once delivery	✓*	✓
At-most-once delivery	✓**	
Overall storage needs > 80 GB		✓

Service Bus Receive Modes

- *Peek Lock: at-least-once delivery – two stage receive operation
- **Receive delete: at-most-once delivery – message is immediately marked as received

Appendices

Appendix 1 – Deleted Topics (out of scope as of March 26, 2021)

A1-1.1 Implement IaaS solutions
- configure VMs for remote access
- Azure Kubernetes Services (AKS)

VMs – Remote Access Configuration
Azure VMs can have an optional public IP address assigned to them so that we can interact with the VM over the Internet. Alternatively, we can set up a VPN that connects our on-premises network to Azure.

To connect to a Linux VM via SSH, you need:

- the public IP address of the VM
- the username of the local account on the VM
- a public key configured in that account
- access to the corresponding private key
- port 22 open on the VM

Azure Kubernetes Service (AKS)
AKS and containers provide a platform for microservice based architectures in the cloud. Kubernetes also provide continuous integration and continuous delivery (CI/CD) services that help deploy and manage containerized applications.

A1-2.1 Develop solutions that use Cosmos DB storage
- interact with data using the appropriate SDK
- create Cosmos DB containers
- implement scaling (partitions, containers)
- implement server-side programming including stored procedures, triggers, and change feed notifications

Cosmos DB Scaling (Partitions/Containers)
A partitioning strategy enables you to add more partitions to your database when need them. This scaling strategy is called scale out or horizontal scaling.

Server-Side Programming – Stored procedures, Triggers, Change Feed Notifications

Stored procedures are written in JavaScript and are stored in a container on Azure Cosmos DB. Stored procedures are the only way to achieve atomic transactions within Azure Cosmos DB; the client-side SDKs do not support transactions. UDFs can be called only from inside queries.

A1-2.2 Develop solutions that use blob storage
- implement data archiving and retention
- implement hot, cool, and archive storage

A1-3.1 Implement user authentication and authorization
- implement OAuth2 authentication
- control access to resources by using role-based access controls (RBAC)

OAuth2
Token based authentication.
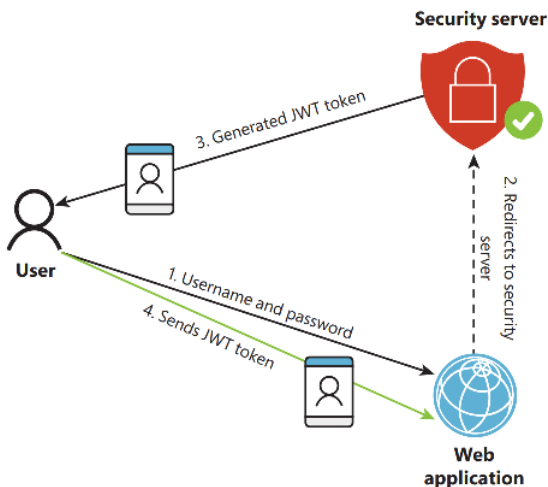
FIGURE 3-1 Basic workflow of token-based authentication
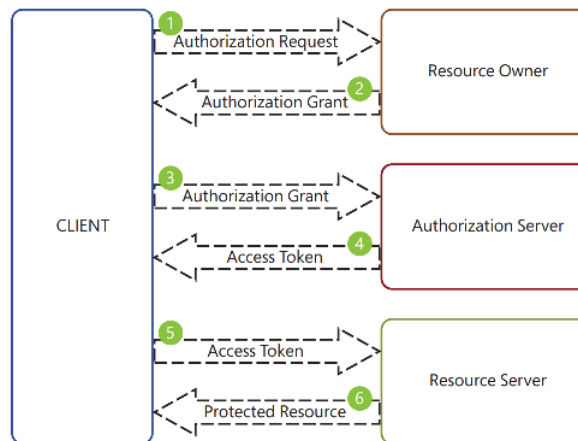
FIGURE 3-2 OAuth basic authentication flow

For example:
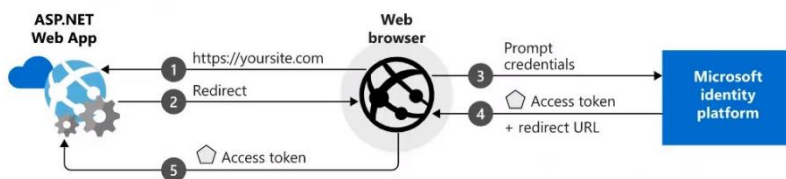
The OAuth protocol defines four roles:

- **Resource owner** Person or entity that can grant access to the resources
- **Resource server** Hosts the resources that you want to share
- **Client** Third-party application that needs to access the resource.
- **Authorization server** Server that issues the access token to the client for accessing the resources.
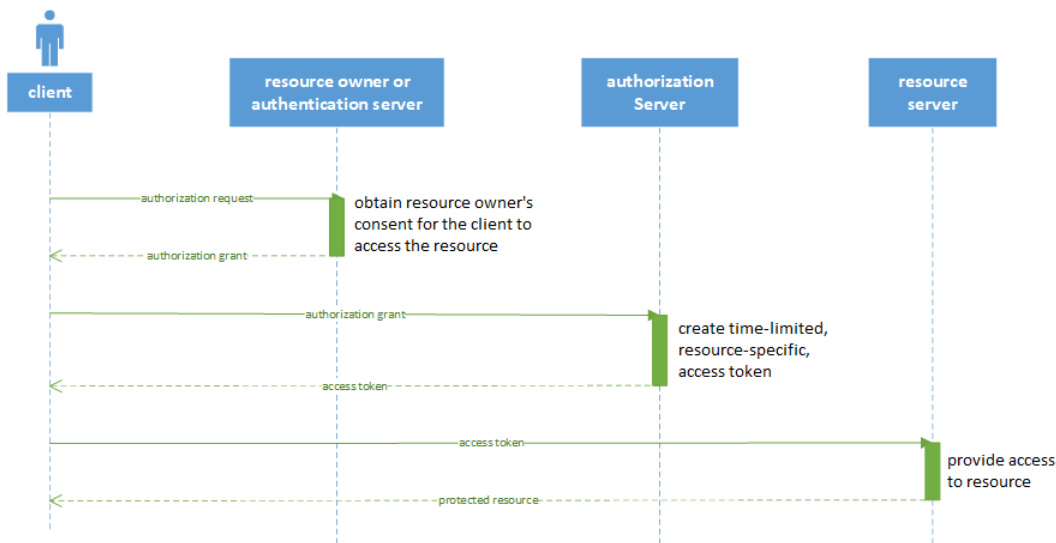
When you are working with OAuth2 authentication, remember that you don't need to store the username and password information in your system. You can delegate that task in specialized authentication servers.

A1-3.2 Implement secure cloud solutions
- implement Managed Identities for Azure resources

Implement Managed Identities for Azure resources

When you are designing your application, you identify the different services or systems on which your application depends. In all these situations, there is a common need to authenticate with the service before you can access it. The drawback is that you need to store a security credential to be able to authenticate to the service that you want to access. You can address most of these situations by using the Azure Key Vault, but your code still needs to authenticate to Azure Key Vault. Fortunately, Azure AD provides *Managed Identities for Azure resources* that removes the need to use credentials for authenticating your application to Azure services.

EXAM TIP: You can configure two different types of managed identities: system- and user-assigned. System-assigned managed identities are tied to the service instance. If you delete the service instance, the system-assigned managed identity is automatically deleted as well. You can assign the same user-assigned managed identities to several service instances. You can also assign several user assigned identities to the same service instance (permissions are additive). When you assign a user-assigned identity, it over-rides the system assigned identity.

A1-4.1 Integrate caching and content delivery within solutions
- develop code to implement CDNs in solutions
- Store and retrieve data in Azure Redis cache

Develop code to implement CDNs in solutions

To use Azure CDN with your solution, you need to configure a profile and create an endpoint, which can be done through the Azure portal. This profile contains the list of endpoints in your application that would be included in the CDN.

When you add content to a CDN cache, the system automatically assigns a Time-To-Live (TTL). If the TTL is lower than the current time, the CDN considers the content to be fresh. If the TTL expires, the CDN marks the content as stale or invalid. When the next user tries to access the invalid content, the CDN compares the cached file with the content in the web server. If the files don't match, the CDN updates it from the web server.
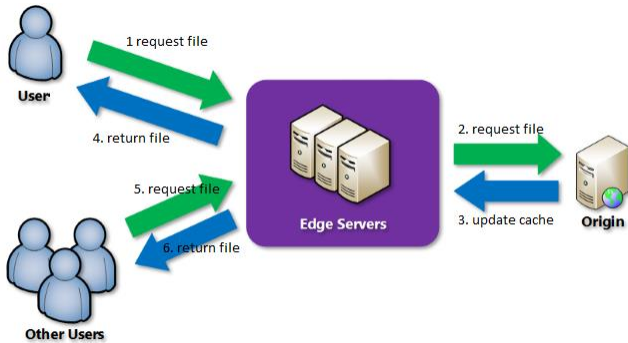
You can configure the default TTL associated with a site by using the Cache-Control HTTP Header. You set the value for this header in different ways:

- Default CDN configuration – Azure CDN automatically configures a default value of seven days.
- Caching rules – Set global or custom caching rules on the CDN
- Web config files:

```
<configuration>
 <system.webServer>
  <staticContent>
   <clientCache cacheControlMode="UseMaxAge" cacheControlMaxAge="3.00:00:00" />
  </staticContent>
 </system.webServer>
</configuration>
```

- Programmatically:

```
// Set the caching parameters.
Response.Cache.SetExpires(DateTime.Now.AddHours(5));
Response.Cache.SetCacheability(HttpCacheability.Public);
Response.Cache.SetLastModified(DateTime.Now);
```

Azure Front Door (an advanced routing and caching system) allows you to cache big files and compress data on the fly. As with Azure CDN, you can configure the cache and expiration time for the elements in the cache. The cache configuration is performed at routing rule level.

FIGURE — Configuring Azure Front Door cache

A1-4.2 Instrument solutions to support monitoring and logging
- configure instrumentation in an app or service by using Application Insights
- implement code that handles transient faults

Handling Transient Faults

Developing an application for the cloud means your application depends on resilient cloud resources, thereby making your application more resilient. But there can still be situations that temporarily affect your application, such as automatic failovers or load balancing operations. These situations are referred to as transient faults. A transient fault is a temporary one-time error that comes and goes.

Strategies to handle transient faults: retry/back-off logic; avoid tightly couples operations between apps and servers – perhaps use queues or other messaging systems. In general, expect that API calls will not always work, and handle errors gracefully.

Detect and classify faults Not all the faults that may happen during the application execution are transient. Your application needs to identify whether the fault is transient, long-lasting, or a terminal failure.

Retry the operation when appropriate Once your application determines that it's dealing with a transient fault, the application needs to retry the operation. It also needs to keep track of the number of retries of the faulting operation.

Implement an appropriate retry strategy Indefinitely retrying the operation could lead to other problems. Your application needs to set a retry strategy that defines the number of retries, sets the delay between each retry, and sets the actions that your application should take after a failed attempt. Setting the correct number of retries and the delay between them depends on factors such as the type of resources, the operating conditions, and the application itself.

Use existing built-in retry mechanism When working with SDKs for specific services, the SDK usually provides a built-in retry mechanism. Before thinking of implementing your retry mechanism, you should review the SDK and use the built-in retry mechanism.

Determine whether the operation is suitable for retrying When an error is raised, it usually indicates the nature of the error. Once you determine your application is dealing with a transient fault, you need to determine whether retrying the operation can succeed. You should implement operation retries if the following conditions are met:

- You can determine the full effect of the operation.
- You fully understand the conditions of the retry.
- You can validate these conditions.

Use the appropriate retry count and interval Setting the wrong retry count could lead your application to fail or could lock resources that can affect the health of the application. If you set the retry count too low, your application may not have enough time to recover and will fail. If you set the retry count too high or too short, you can lock resources, such as threads, connections, or memory. When choosing the appropriate retry count and interval, you need to consider the type of operation that suffered the transient fault. Common retry strategies:

Exponential back-off use a short time interval for the first retry, and then exponentially increase it for subsequent retries. For example, you set the initial interval to 3 seconds and then 9, 27, 81 for subsequent retries.

Incremental intervals use a short time interval for the first retry, then incrementally increase it for the subsequent retries. For example, you set the initial interval to 3 seconds and then 5, 8, 13, 21 for subsequent retries.

Regular intervals use the same time interval for each retry. This strategy is not appropriate in most cases.

Immediate retry You should not use this type of retry more than once. If the immediate retry doesn't recover from the transient fault, you should switch to another retry strategy.

Randomization If your application executes several retries in parallel, use random starting retry interval values with any of the previous strategies. This allows you to minimize the probability that two different application threads start the retry mechanism at the same time.

Avoid anti-patterns When implementing your retry mechanism, there are some patterns you should avoid:

- Avoid implementing duplicated layers of retries. If your operation is made of several requests to several services, you should avoid implementing retries on every stage of the operation.
- Never implement endless retry mechanisms. This can cause resource exhaustion or connection throttling. You should use the circuit breaker pattern or a finite number of retries.
- Never use immediate retry more than once.

Test the retry strategy and implementation Because of the difficulties when selecting the correct retry count and interval values, you should thoroughly test your retry strategy and implementation. You should pay special attention to heavy load and high concurrency scenarios. You should test this by injecting transient and non-transient faults into your application.

Manage retry policy configuration When you are implementing your retry mechanism, you should not hardcode retry count and interval values. Instead, you can define a retry policy that contains the retry count and interval as well as the mechanism that determines whether a fault is transient or non-transient. You should store this retry policy in configuration files so that you can fine-tune the policy.

Log transient and non-transient faults A single transient fault doesn't indicate an error in your application. If the number of transient faults is increasing, this can be an indicator of a more significant potential failure or that you should increase the resources assigned to the faulting service. You should log transient faults as Warning messages instead of Errors. Using the Error log level could lead to triggering false alerts in your monitoring system.

Circuit Breaker Pattern One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached. The basic idea behind the circuit breaker pattern is this: wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error or with some alternative service or default message, without the protected call being made at all. This will make sure system is responsive and threads are not waiting for an unresponsive call.
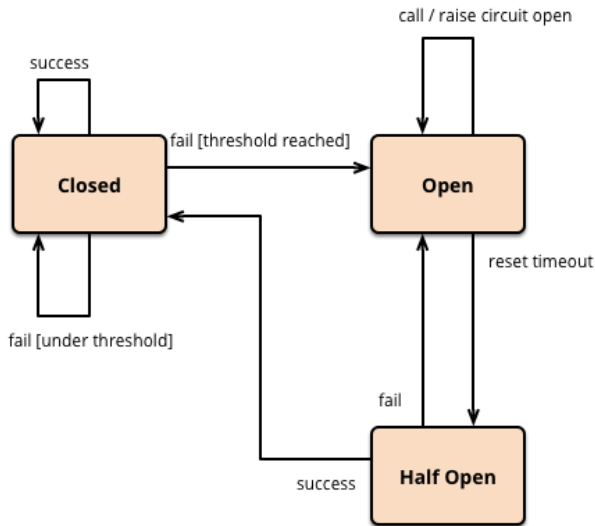
The circuit breaker has three distinct states:

- **Closed** When everything is normal, all calls pass through to the services. When the number of failures exceeds a threshold the breaker trips, and it goes into the Open state.

- **Open** The circuit breaker returns an error for calls without executing the function.

- **Half-Open** After a timeout period, the circuit switches to a half-open state to test if the underlying problem still exists. If a single call fails in this half-open state, the breaker is once again tripped. If it succeeds, the circuit breaker resets back to the normal, closed state.

A1-5.1 Develop an App Service Logic App
- create a Logic App
- create a custom connector for Logic Apps
- create a custom template for Logic Apps

Logic App Definition
A Logic App is a connector service that connects Azure services to each other and to third-party services. Logic Apps manage work flows.

Create a Logic App
Skill: create a Logic App

https://docs.microsoft.com/en-us/learn/paths/build-workflows-with-logic-apps/

Azure Logic Apps automate the execution of business processes using a graphical design tool to arrange pre-made components into the sequence you need.
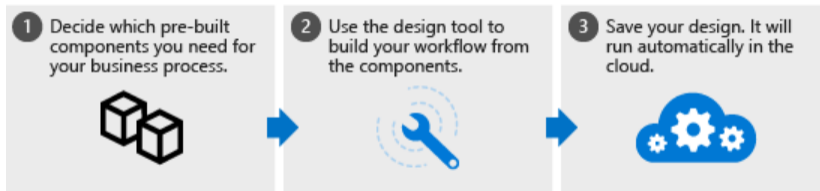
Logic Apps Custom Connector
Skill: create a custom connector for Logic Apps

A connector is a component that provides an interface to an external service, using the external service's REST or SOAP API. You can write custom connectors to access services that don't have pre-built connectors.

Logic Apps Custom Template
Skill: create a custom template for Logic Apps

https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-create-azure-resource-manager-templates

https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-azure-resource-manager-templates-overview

ARM templates can be used to create and deploy logic apps.

Appendix 2 – Command-Line commands

Azure CLI, PowerShell, Docker, Func, …

A2-1.1 Implement IaaS solutions

Provision VMs
Skill: provision VMs

Create resource group.

```
#CLI create resource group example
az group create \
    --name "test-rg" \
    --location "centralus"
```

```
# powershell create resource group example
New-AzResourceGroup `
  -Name "test-rg" `
  -Location "{location}"
```

Create vm. Open port. Get public IP. Azure CLI example:

```
#create vm
az vm create \                          az vm create \
    --resource-group "test-rg" \            --resource-group "test-rg" \
    --name "win2016-vm" \                   --name "linux-vm" \
    --image "win2016datacenter" \           --image "UbuntuLTS" \
    --admin-username "robi" \               --admin-username "robi" \
    --admin-password "myPasswordHere"       --authentication-type "ssh" \
                                            --ssh-key-value ~/.ssh/id_rsa.pub

#open port
az vm open-port \                       az vm open-port \
    --resource-group "test-rg" \            --resource-group "test-rg" \
    --name "win2016-vm" \                   --name "linux-vm" \
    --port "3389"                           --port "22"

#get public ip
az vm list-ip-addresses \
    --resource-group "test-rg" \
    --name "win2016-vm"
```

PowerShell example:

```
New-AzVm `
    -ResourceGroupName "test-rg" `
    -Name "win2016-vm" `
    -Image "Win2019Datacenter" `
    -Location "East US" `
```

```
    -VirtualNetworkName "test-wp1-eus-network" `
    -SubnetName "default" `
    -SecurityGroupName "test-wp1-eus-nsg" `
    -PublicIpAddressName "test-wp1-eus-pubip" `
    -OpenPorts 80,3389

Get-AzPublicIpAddress `
    -ResourceGroupName "test-rg" `
    -Name "win2016-vm"
```

VMs – ARM Templates

Skill: configure, validate, and deploy ARM templates

To deploy an ARM template, use either the Azure CLI command `az deployment group create` or the Azure PowerShell command `New-AzResourceGroupDeployment`.

```
# CLI example
az deployment group create \
  --name blanktemplate \
  --resource-group myResourceGroup \
  --template-file "{provide-the-path-to-the-template-file}"
```

```
# powershell example
New- AzResourceGroupDeployment `
  -Name {name of your resource group} `
  -Location "{location}"
```

Create Docker Images

Skill: configure container images for solutions

Building a Container Image

Example Dockerfile:

FROM defines a base container image used for subsequent instructions in the docker file.

RUN executes a command inside the container. In the example, it will create a directory named app inside the container image.

WORKDIR sets the working directory for any subsequent instructions in this docker file.

COPY is used to copy application binaries into the container.

RUN executes scripts inside the container.

EXPOSE tells the container's runtime which port the application in the container is listening on.

ENTRYPOINT is used to define which script or binary to start when the container is started from this container image. Inside the [] is the command to run along with its parameters. Sometimes CMD is used instead of ENTRYPOINT.

Build container image from docker file in current directory. Name and tag the container image:

```
docker build -t webappimage:v1 .
```

To run an instance of this container image on the local host as a container:

```
docker run –name webapp –publish 8080:80 –detach webappimage:v1
```

-*publish* maps the local machine port 8080 to the container port 80 that the application running inside the container is listening on.

With the container running, we can use curl to connect to the published port 8080.

```
curl http://localhost:8080
```

We can stop and remove the running webapp (the container):

```
docker stop webapp
docker rm webapp
```

Publish Image to Azure Container Registry (ACR)
Skill: publish an image to the Azure Container Registry

First, create an ACR:

```
ACR_NAME='demo-acr'

az acr create \
    --resource-group 'test-rg' \
    --name $ACR_NAME \
    --sku Standard
```

To push your image to your ACR registry:

```
# 1. Log in to your Azure subscription
az login

# 2. Log in to your registry
az acr login --name $ACR_NAME

# 3. Tag the image that you want to upload to the registry - for example webappimage:v1
docker tag webappimage:v1 $ACR_NAME.azurecr.io/webappimage:v1

# 4. Push the image for example webappimage:v1
docker push $ACR_NAME.azurecr.io/webappimage:v1
```

Alternatively, you can push the image using ACR tasks:

```
az acr build –image "webappimage:v1-acr-task" –registry $ACR_NAME .
```

Azure Container Instances – Running Containers
Skill: run containers by using Azure Container Instance

```
#deploy container into ACI
SERVER=$(az acr show -query loginServer)
az container create \
    --resource-group test-rg \
    --name "my-app" \
    --ports 80 \
    --image $SERVER/webappimage:v1
```

Once deployed, the app in the container will be running in ACI. Get the URL as follows:

```
URL=$(az container show --name 'my-app' --query ipAddress.fqdn)
echo $URL
```

A2-1.2 Create Azure App Service Web Apps

Create an Azure App Service Web App
Skill: create an Azure App Service Web App

From Azure CLI
You can create a web app using Azure CLI.
```
as group create --name my-resource-group --location westus2

az appservice plan create --name my-app-service-plan --resource-group my-resource-group \
    --sku s1 --is-linux

az webapp create --group my-resource-group --plan my-app-service-plan \
    --name my-web-app --runtime "node"
```

From PowerShell
You can create a web app using PowerShell.

```
New-AzResourceGroup -Name 'my-rg' -Location 'westus'

New-AzAppServicePlan -Name 'my-plan' -Location 'westus'`
    -ResourceGroupName 'my-rg' -Tier S1

New-AzWebApp -Name 'my-web-app' -Location 'westus' `
    -AppServicePlan 'my-plan' -ResourceGroupName 'my-rg'
```

Deploy Code to a Web App
Skill: deploy code to a web app

From Azure CLI
```
az webapp up
```

A2-1.3 Implement Azure functions
Create and deploy Azure Function apps
Skill: create and deploy Azure Function apps

```
# create function app
az functionapp create -g MyResourceGrp -p MyAppServicePlan -n MyFuncAppName -s MyStorageAcct
```

After the Function App has been created, you can publish Azure Functions to it.

To use the Func command, first install Azure func core tools:

```
npm i azure-functions-core-tools
```

Then make a directory and go there:

```
md azfunc
cd azfunc
```

Initialize the function app:

```
# as a dotnet app
func init dotnet
```

```
# or as a node app
func init node
```

```
# or as a python app
func init python
```

Then create the function:

```
# example: cosmos dB triggered
func new CosmosDBTrigger
```

A2-2.1 Develop solutions that use Cosmos DB storage
Example: create a Cosmos DB account using Azure CLI.

```
# create a cosmos db account (defaults to sql api)
az cosmosdb create --name pluralsight --resource-group pluralsight

# create a sql database
az cosmosdb sql database create --account-name pluralsight
    --name sampledb

# create a sql database container
az cosmosdb sql container create --resource-group pluralsight
    --account-name pluralsight --database-name sampledb
    --name samplecontainer --partition-key-path "/employeeid"
```

A2-2.2 Develop solutions that use Blob storage

Move Blob storage items between storage accounts or containers
Skill: move items in Blob storage between storage accounts or containers

To copy blobs with the Azure CLI, use the `az storage blob copy` command.

```
#copy a single blob                                  #copy container with all its blobs
az storage blob copy start                           az storage blob copy start-batch
    --source-account-name   acct1                         --source-account-name   acct1
    --source-account-key    00000000                      --source-account-key    00000000
    --source-container      images                        --source-container      images
    --source-blob           pic.jpg
    --account-name          acct2                          --account-name          acct2
    --account-key           00000000                       --account-key           00000000
    --destination-container pics                           --destination-container pictures
    --destination-blob      pic.jpg
```

The AzCopy command line utility has a copy command.

General form:

```
azcopy copy "<source-path>" "<target-path>" [--recursive=true]
```

Examples:

```
# upload local folder
azcopy copy "C:\Documents" "https://mystorageurl.com/mycontainer?[SAS]" -recursive=true


# copy single blob between storage accounts
azcopy copy "https://url1.com/mycontainer/myfile.txt?[SAS]" "https://url2.com/mycontainer/myfile.txt?[SAS]"


# copy all blobs from one container to another
azcopy copy "https://url1.com/mycontainer1?[SAS]" "https://url2.com/mycontainer2?[SAS]" -recursive=true


#copy all containers from one storage account to another
azcopy copy "https://url1.com?[SAS]" "https://url2.com?[SAS]" -recursive=true
```

You can change the blob storage tier from Azure CLI:

```
az storage blob set-tier
    --account-key 12345
    --account-name myacct
    --container mycontainer
    --name mypic.jpg
    --tier Cool
```

A2-3.2 Implement secure cloud solutions

Secure App Configuration Data
Skill: secure app configuration data by using the App Configuration and Azure Key Vault

Creating a Key Vault:

```
# CLI
az keyvault create --resource-group 'rg-101' [ …optional parameters… ]
```

```
# PowerShell
New-AzKeyVault -VaultName 'MyVault' -ResourceGroupName 'rg-101' -Location 'East US'
```

Manage keys, secrets, and certificates by using the Key Vault API
Skill: develop code that uses keys, secrets, and certificates stored in Azure Key Vault

```
az keyvault create \
    --resource-group learn-950b3a03-fe30-4c1f-815d-d259db8eff26 \
    --location centralus \
    --name ridvaultname

az keyvault secret set \
    --name SecretPassword \
    --value reindeer_flotilla \
    --vault-name ridvaultname
```

A2-4.1 Integrate caching and content delivery within solutions
Skill: configure cache and expiration policies for Azure Redis Cache

```
az redis create \
    --name "MY_NAME" \
    --resource-group "MY_RG" \
    --location eastus \
    --vm-size C0 \
    --sku Basic
```

A2-4.2 Instrument solutions to support monitoring and logging
Skill: configure an app or service to use Application Insights

```
# Configure web server logging to the file system (Windows or Linux)
az webapp log config \
  --name myApp \
  --resource-group myRG \
  --web-server-logging filesystem
```

```
# Configure app logging to Azure Blob Storage (Windows only)
az webapp log config \
  --name myApp \
  --resource-group myRG \
  --application-logging azureblobstorage
```

```
# Configure container logging to the file system (Linux only)
az webapp log config \
  --name myApp \
  --resource-group myRG \
  --docker-container-logging filesystem
```

A2-5.2 Implement API Management
Create an APIM instance
Skill: create an APIM instance

```
# CLI
```

```
az apim create --name myapim --resource-group myResourceGroup \
  --publisher-name Contoso --publisher-email admin@contoso.com \
  --no-wait



# PowerShell
New-AzApiManagement -Name "myapim" -ResourceGroupName "myResourceGroup" `
  -Location "West US" -Organization "Contoso" -AdminEmail admin@contoso.com
```

A2-5.3 Develop event-based solutions

Event Grid
Skill: implement solutions that use Azure Event Grid

Instead of creating a service, Event grid is enabled at the subscription level, by registering a resource provider. This way other services can send events to it:

```
az provider register --namespace Microsoft.EventGrid
az provider show --namespace MicrosoftEventGrid --query "registrationState"
```

Event Hub
Skill: implement solutions that use Azure Event Hub

You need to create an Event Hub namespace before creating the Event Hub:

```
az eventhubs namespace create --name <NAME> /
    --resource-group <NAME> /
    --location <LOCATION> /
    --sku <Basic|Standard>
```

Once the namespace is created, you can create the Event Hub:

```
az eventhubs eventhub create --name <NAME> /
    --namespace <NAME> /
    --message-retention 3 /
    --partition-count 4 /
    -g <GROUP NAME>
```

Import OpenAPI definitions
Skill: import OpenAPI definitions

Import the OpenAPI via PowerShell:

```
$ApiMgmtContext = New-AzApiManagementContext -ResourceGroupName "my-rg" -ServiceName "my-apim"

Import-AzApiManagementApi -Context $ApiMgmtContext -SpecificationFormat "OpenApi" -SpecificationPath
"C:\Users\XXXXXX\Downloads\myAPI-0.1-swagger.yaml" -Path "myapi"
```

A2-5.4 Develop message-based solutions

Service Bus
Skill: implement solutions that use Azure Service Bus

Interact with Service Bus using CLI:

```
#create a queue
az servicebus queue create --name <NAME> --namespace-name <NAME> --resource-group <RG-NAME>

#delete a queue
az servicebus queue delete --name <NAME> --namespace-name <NAME> --resource-group <RG-NAME>

#create a topic
Az servicebus topic create --name <NAME> --namespace-name <NAME> --resource-group <RG-NAME>

#delete a topic
az servicebus topic delete --name <NAME> --namespace-name <NAME>

#create a subscription
az servicebus topic subscription create --name <NAME> --namespace-name <NAME> --topic-name <NAME>
```

Azure Queue Storage queues
Skill: implement solutions that use Azure Queue Storage queues

Interacting with queue storage using CLI:

```
#create a queue
az storage queue create --name <NAME>

#delete a queue
az storage queue delete --name <NAME>

#view message (without affecting visibility)
az storage message peek --queue-name <NAME>

#get message
az storage message get --queue-name <NAME>

#delete a message
az storage message delete --queue-name <NAME> --id <MESSAGE-ID> --pop-receipt <RECEIPT-ID>

#delete all messages
az storage message clear --queue-name <NAME>
```

With PowerShell:

```
New-AzResourceGroup -ResourceGroupName <RG-NAME>
New-AzStorageAccount -Name <NAME> -ResourceGroupName <RG-NAME>
$MyKey = Get-AzStorageAccountKey -AccountName <NAME>
New-AzStorageQueue -Name <Q-NAME> -StorageAccountName <NAME> -StorageAccountKey $MyKey
Get-AzStorageQueue -StorageAccountName <NAME> -StorageAccountKey $MyKey | Select Name
```

A2-Miscellaneous – Managing Azure AD Tenants

http://vcloud-lab.com/entries/microsoft-azure/how-to-switch-to-other-azure-ad-tenant-using-powershell-and-azure-cli

Between Azure CLI and PowerShell, you can manage the currently active tenant.

```
# Get all tenants associated with the currently logged in user.
Get-AzTenant
```

```
# Get currently active tenant
az account show
```

```
# login to a different tenant - note: tenant must be linked to a subscription
az logout
az login --tenant dtekorg.onmicrosoft.com
```

Appendix 3 – [ARM templates](#) (Infrastructure as Code)

https://docs.microsoft.com/en-gb/learn/paths/az-204-implement-iaas-solutions/

https://docs.microsoft.com/en-us/learn/paths/deploy-manage-resource-manager-templates/

https://docs.microsoft.com/en-us/learn/modules/create-azure-resource-manager-template-vs-code/

https://docs.microsoft.com/en-us/learn/modules/create-azure-resource-manager-template-vs-code/2-explore-template-structure

https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/template-user-defined-functions

A3-1 ARM Template File Structure

schema	required	the location of the JSON schema file that describes the structure of JSON data
contentVersion	required	the version of your template – to document significant changes and to ensure you're deploying the right template
resources	required	the actual items you want to deploy or update in a resource group or a subscription
apiProfile	opt.	defines a collection of API versions for resource types
parameters	opt.	define values that are provided during deployment – can be provided by a parameter file, command-line parameters, or in Azure portal
variables	opt.	define values that are used to simplify template language expressions
functions	opt.	user-defined functions that are available within the template – simplify your template when complicated expressions are used repeatedly
output	opt.	the values that will be returned at the end of the deployment

In its simplest structure, a template has the following elements:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "",
  "apiProfile": "",
  "parameters": {  },
  "variables": {  },
  "functions": [  ],
  "resources": [  ],
  "outputs": {  }
}
```

Example:

```json
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.1",
    "apiProfile": "",
    "parameters": {},
    "variables": {},
    "functions": [],
    "resources": [
        {
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2019-06-01",
            "name": "learntemplatestorage123",
            "location": "westus",
            "sku": {
                "name": "Standard_LRS"
            },
            "kind": "StorageV2",
            "properties": {
                "supportsHttpsTrafficOnly": true
            }
        }
    ],
    "outputs": {}
}
```

A3-2 ARM Template Parameters

ARM template *parameters* section has the following properties:

```
"parameters": {
  "<parameter-name>" : {
    "type" : "<type-of-parameter-value>",
    "defaultValue": "<default-value-of-parameter>",
    "allowedValues": [ "<array-of-allowed-values>" ],
    "minValue": <minimum-value-for-int>,
    "maxValue": <maximum-value-for-int>,
    "minLength": <minimum-length-for-string-or-array>,
    "maxLength": <maximum-length-for-string-or-array-parameters>,
    "metadata": {
      "description": "<description-of-the parameter>"
    }
  }
}
```

allowed parameters types:

- string
- secureString
- integers
- boolean
- object
- secureObject
- array

Example parameter section:

```
"parameters": {
```

```
    "storageAccountType": {
        "type": "string",
        "defaultValue": "Standard_GRS",
        "metadata": {
            "description": "The type of the new storage account created to store the VM disks."
        }
    }
}
```

Example parameter usage:

```
"resources": [
    {
        "type": "Microsoft.Storage/storageAccounts",
        "apiVersion": "2019-04-01",
        "name": "learntemplatestorage123",
        "location": "{Location}",
        "sku": {
            "name": "[parameters('storageAccountType')]"
        },
        "kind": "StorageV2",
        "properties": {
            "supportsHttpsTrafficOnly": true
        }
    }
]
}
```

Example call with a parameter:

```
az deployment group create \
  testdeployment1 \
  --group my-rg \
  --template-file $templateFile \
  --parameters storageAccountType=Standard_LRS
```

A3-3 ARM Template Outputs

ARM template *outputs* section specifies values that will be returned after a successful deployment.

```
"outputs": {
    "<output-name>": {
        "condition": "<boolean-value-whether-to-output-value>",
        "type": "<type-of-output-value>",
        "value": "<output-value-expression>",
        "copy": {
            "count": <number-of-iterations>,
            "input": <values-for-the-variable>
        }
    }
}
```

Example output section:

```
"outputs": {
        "storageEndpoint": {
            "type": "object",
            "value": "[reference('learntemplatestorage123').primaryEndpoints]"
        }
    }
```

A3-4 ARM Template Functions

ARM template *functions* allow you to construct values you need dynamically. They appear in expressions. Expressions are values that are evaluated when the template is deployed. Expressions start and end with brackets, [and], and can return a string, integer, Boolean, array, or object.

Example ARM template function:

```
"parameters": {
  "location": {
    "type": "string",
    "defaultValue": "[resourceGroup().location]"
  }
},
```

There are many kinds of ARM template functions:

- **Array functions** for working with arrays. For example, first and last.
- **Comparison functions** for making comparisons in your templates. For example, equals and greater.
- **Date functions** for working with dates. For example, utcNow and dateTimeAdd.
- **Deployment value functions** for values related to the deployment. For example, environment and parameters.
- **Logical functions** for working with logical conditions. For example, if and not.
- **Numeric functions** for working with integers. For example, max and mod.
- **Object functions** for working with objects. For example, contains and length.
- **Resource functions** for getting resource values. For example, resourceGroup and subscription.
- **String functions** for working with strings. For example, length and startsWith.

A3-5 ARM Template Variables

An ARM template *variable* is used when a value needs to be specified in several places in a template. Wherever the variable is used in the template, Resource Manager replaces it with the resolved value. Using ARM template variables, maintenance of expressions is in one place, and the template is more readable. ARM template variables are defined in the variables: {} section of a template.

Example ARM template variable definition:

```
"variables": {
  "storageName": "[concat(parameters('storageNamePrefix'), uniqueString(resourceGroup().id))]"
},
```

Example ARM template variable usage:

```
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "name": "[variables('storageName')]",
  }
]
```

A3-6 Resource Tags

When deploying resources with an ARM template, you can use *resource tags* to add values that help you identify the resource's use. The tag values will be displayed on the overview page for the Azure resource and in cost reports. Resource tags appear in the resource section of the ARM template. For example:

```
"resources": [{
        "name": "[variables('uniqueStorageName')]",
        "type": "Microsoft.Storage/storageAccounts",
        "apiVersion": "2019-06-01",
        "tags": {
                "value": {
                    "Environment": "Dev",
                    "Project": "Learn"
                }
            }
        },
    }],
```

A3-7 ARM Template Parameter Files

ARM template *parameter files* are JSON files that hold parameter values. For example:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      "storagePrefix": {
        "value": "storage"
      },
      "storageSKU": {
        "value": "Standard_LRS"
      },
      "resourceTags": {
        "value": {
            "Environment": "Dev",
            "Project": "Learn"
        }
      }
    }
}
```

A3-8 ARM Template Deployment

To deploy an ARM template, use either the Azure CLI command `az deployment group create` or the Azure PowerShell command `New-AzResourceGroupDeployment`. To use a parameter file, you specify the path to the parameter file in the deployment command.

```
# CLI example
az deployment group create \
  --name blanktemplate \
  --resource-group myResourceGroup \
  --template-file "{provide-the-path-to-the-template-file}", \
  --parameters "{path-to-the-template-file}"
```

```
# powershell example
New-AzResourceGroupDeployment `
  -Name blanktemplate `
  -ResourceGroupName myResourceGroup `
  -TemplateFile "{provide-the-path-to-the-template-file}" `
  -TemplateParameterFile "{path-to-the-template-file}"
```

A3-9 ARM Management Policies

ARM management policies are useful for controlling ARM deployment. They are used to restrict the creation and management of Azure resources. For example

```
{
    "if": {
        "not": {
            "field": "tags",
            "containsKey": "ProjectID"
        }
    },
    "then": {
        "effect": "deny"
    }
},
{
    "if": {
        "not": {
            "field": "location",
            "in": ["CanadaEast", "CanadaCentral"]
        }
    },
    "then": {
        "effect": "deny"
    }
}
```

ARM management policies can be deployed:

- REST API
    - HTTPS PUT commands
- Azure PowerShell
    - New-AzureRmPolicyDefinition
    - New-AzureRmPolicyAssignment
    - Remove-AzureRmPolicyAssignment
- Azure CLI
    - az policy definition create
    - az policy assignment create
    - az policy assignment delete

The steps are similar to working with RBAC role assignments. You create a policy definition, and then assign it to a scope.

Appendix 4 – Policies

Policies are collections of statements that serve to configure and change the behavior of the various Azure services.

A4-2.2 Develop solutions that use blob storage

storage policies (45)

stored access policies (46)

Blob Storage Lifecycle Management Polices (46-47)

Storage access policies (48)

A4-3.2 Implement secure cloud solutions

access policies (58)

A4-4.1 Cache and Expiration Policies, Cache and Expiration Policies for Azure Redis Cache

configure cache and expiration policies

Caching policies are special rules that you can set up, either from the Azure Portal, or the SDK, to control how TTL values are assigned. You can create global rules for all objects within the cache, or you can use custom rules for different file types or paths within your app. It's even possible to exclude certain areas within your app so the data there isn't cached at all. The web server can be configured to send a cache control header as part of the HTTP response.

configure cache and expiration policies for Azure Redis Cache

Redis cache expiration policy

By default, a key-value pair in Redis cache won't expire. Redis console commands to implement and manage data expiration:

- EXPIRE sets the timeout of a key (TTL)
- TTL returns the remaining time a key has to live
- PERSIST makes a key never expire

Alternatively, using the C# ServiceStack.Redis package:

```
redisClient.Expire(key, 3600); // Note: same effect as EXPIRE command from console
```

Redis cache eviction policies

Redis Cache eviction policies (aka maxmemory policy) indicate how data should be handled when you run out of memory:

volatile-LRU (*default*)	evict the least recently used (LRU) keys first, but only among keys that have an expire set
volatile-TTL	evict keys with a shorter TTL first, but only among keys that have an expire set
volatile-random	evict keys randomly, but only among keys that have an expire set
allkeys-LRU	evict keys by trying to remove the least recently used (LRU) keys first
allkeys-random	evict keys randomly
noeviction	return errors when the memory limit was reached

A4-5.2 Implement API Management

define policies for APIs

APIM Policies are a capability of Azure API Management that allow changing the behavior of the API through configuration.

Polices are defined using XML format.

```
<policies>
  <inbound>
    <!-- statements to be applied to the request go here -->
  </inbound>
  <backend>
    <!-- statements to be applied before the request is forwarded to
         the backend service go here -->
  </backend>
  <outbound>
    <!-- statements to be applied to the response go here -->
  </outbound>
  <on-error>
    <!-- statements to be applied if there is an error condition go here -->
  </on-error>
</policies>
```

A4-5.4 Develop message-based solutions

Create a shared access policy

Appendix 5 – Pricing Tiers (sku)

https://docs.microsoft.com/en-us/learn/modules/configure-app-service-plans/3-determine-plan-pricing

A5-1.2 Create Azure App Service Web Apps

Pricing tiers: Free (F), Shared (D), Basic (B), Standard (S), Premium (P), Isolated (I).

- **Shared compute**: **Free (F)** and **Shared (D)**, the two base tiers, runs an app on the same Azure VM as 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 (B)**, **Standard (S)**, **Premium (P)** tiers run apps on dedicated Azure VMs. The higher the tier, the more VM instances are available to you for scale-out. SSL certificates, "Always On", and manual scaling are only supported on Basic (B) tier or higher. Autoscaling and Deployment Slots are only supported on Standard (S) tier or higher. Standard (S) tier or higher is recommended for production workloads.

- **Isolated (I)**: This tier runs dedicated Azure VMs on dedicated Azure Virtual Networks. It provides network isolation on top of compute isolation to your apps. It provides the maximum scale-out capabilities. Isolated (I) tier is needed for App Service Environment (ASE) (fully isolated and dedicated environment for securely running App Service apps at high scale)

Examples: F1 is free tier; D2 is shared tier; B1 is basic tier.

A5-1.3 Implement Azure Functions

https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale

Plan	Benefits
Consumption plan	Scale automatically and only pay for compute resources when your functions are running. Instances of the Functions host are dynamically added and removed based on the number of incoming events. - Default hosting plan. - Pay only when your functions are running. - Scales automatically, even during periods of high load.
Premium plan	Automatically scales based on demand using pre-warmed workers which run applications with no delay after being idle, runs on more powerful instances, and connects to virtual networks.
Dedicated plan	Run your functions within an App Service plan at regular App Service

Plan	Benefits
	plan rates.

A5-2.2 Develop solutions that use blob storage

There are three kinds of standard performance storage accounts:
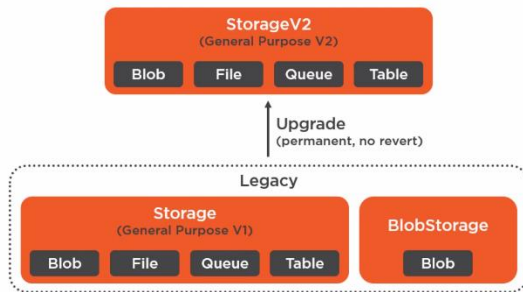
Omitting the legacy storage account kinds, we have the following standard and premium storage account kinds:
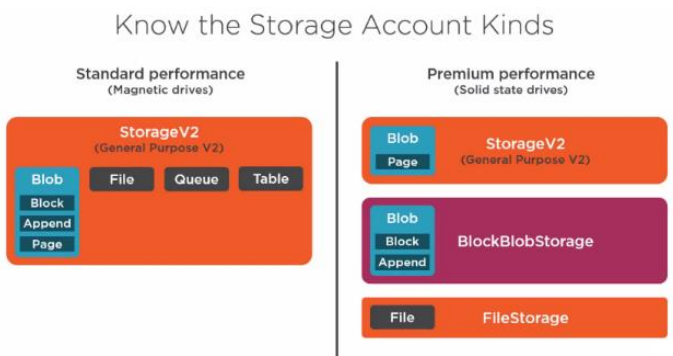
A5-3.2 Implement secure cloud solutions

Key Vault is available under two different sku's, Standard and Premium. You can store Key Vault information in software or hardware security module (HSM), but HSM requires the Premium sku.

A5-4.1 Cache and Expiration Policies, Cache and Expiration Policies for Azure Redis Cache

Pricing tiers: Basic, Standard, Premium. Also, Enterprise and Enterprise Flash. You can control the amount of cache memory available on each tier by choosing a *cache level* (vm-size) from C0-C6 for Basic/Standard and P0-P4 for Premium. Microsoft recommends you always use Standard or Premium Tier for production systems. Also, use at least a C1 cache. C0 caches are meant for simple dev/test scenarios. Premium also allows you to:

- configure data persistence for disaster recovery

- deploy Redis cache to a virtual network for security
- use up to 53 GB of memory in a single instance
- implement clustering to automatically split your dataset among multiple nodes (aka. shards) – Redis Cluster supports up to 10 shards to create 530 GB of memory

A5-5.2 Implement API management

APIM tiers: Developer, Basic, Standard, Premium, Isolated, Consumption.

Developer tier has no SLA. Standard adds Azure AD integration. Premium adds multi-region deployments. Isolated adds complete compute isolation. Consumption tier runs on shared hardware, is serverless (billed per call rather than per hour), scales automatically to meet demand – calls to the consumption tier require a short warm up time, so they are a bit slower to start.

A5-5.4 Develop Message Based Solutions

Service Bus tiers: chosen at the namespace level: Basic, Standard, Premium. Basic does not include queues. Standard has a max 256 KB message size. Premium has a max 1 MB message size.

Appendix 6 – Code and C# Classes

A6-1.1 Implement IaaS Solutions

Creating a VM:

```
static void Main(string[] args)
{
    var azure = Azure
            .Configure()
            .Authenticate(<credentials>)
            .WithDefaultSubscription();

    var rg = azure.ResourceGroups
            .Define(rgName)
            .Create();

    var vnet = azure.Networks
            .Define(vNetName)
            .WithExistingResourceGroup(rg)
            .Create();

    var nic = azure.NetworkInterfaces
            .Define(nicName)
            .WithExistingResourceGroup(rg)
            .WithExistingPrimaryNetwork(vnet)
            .Create();

    var vm = azure.VirtualMachines
            .Define(vmName)
            .WithExistingResourceGroup(rg)
            .WithExistingPrimaryNetworkInterface(nic)
            .WithLatestWindowsImage("2012-R2-Datacenter")
            .WithSize(Standard)
            .Create();
}
```

A6-1.2 Create Azure App Service Web Apps

A6-1.3 Implement Azure Functions

configure input and output bindings using parameter decorators:

```
public static class BlobTriggerCSharp
{
        [FunctionName("BlobTriggerCSharp")]
        public static Task Run(
                [EventGridTrigger]EventGridEvent ev,
                [Blob("{data.url}", FileAccess.Read)] Stream myBlob, // in implied
                [CosmosDB(databaseName: "GIS")] out dynamic doc,
                [SignalR(HubName = "myhub")]IAsyncCollector<MyClass> sigR // in implied
                )
                {
                        doc = new { Description = ev.Topic, id = Guid.NewGuid() };
                        return sigR.AddAsync( new SignalRMessage{Target="newMessage"} );
                }
}
```

A6-2.1 Cosmos DB Storage

Cosmos DB DocumentDB API examples. Old way:

```
using Microsoft.Azure.Documents.Client;

DocumentClient dc = new DocumentClient(uri, key);
var result = dc.CreateDocumentCollectionUri(uri, partitionkeyValue)
        .GetAwaiter()
        .GetResult();

var query = dc.CreateDocumentQuery<MyModel>(uri, myOptions)
        .Where(e => e.LastName == "Jack");

foreach (var emp in query)
{
        Console.WriteLine(emp);
}
```

Version 3 of .NET SDK, add `Microsoft.Azure.Cosmos` package from command line:

```
dotnet add package Microsoft.Azure.Cosmos
```

Then using C# you can do this:

```
CosmosClient client = new CosmosClient(endpoint, key);

// An object containing relevant information about the response
DatabaseResponse databaseResponse = await client.CreateDatabaseIfNotExistsAsync(databaseId, 10000);

// A client side reference object that allows additional operations like ReadAsync
Database database = databaseResponse;
DatabaseResponse readResponse = await database.ReadAsync();

// Set throughput to the minimum value of 400 RU/s
ContainerResponse simpleContainer = await database.CreateContainerIfNotExistsAsync(
    id: containerId,
    partitionKeyPath: partitionKey,
    throughput: 400);

Container container = database.GetContainer(containerId);
ContainerProperties containerProperties = await container.ReadContainerAsync();
ItemResponse<SalesOrder> response = await container.CreateItemAsync(salesOrder, new
PartitionKey(salesOrder.AccountNumber));

string id = "[id]";
string accountNumber = "[partition-key]";
ItemResponse<SalesOrder> response = await container.ReadItemAsync(id, new PartitionKey(accountNumber));

// Create a query for items under a container using a parameterized SQL statement. It returns a FeedIterator.
QueryDefinition query = new QueryDefinition(
    "select * from sales s where s.AccountNumber = @AccountInput ")
    .WithParameter("@AccountInput", "Account1");

FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
    query,
    requestOptions: new QueryRequestOptions()
    {
        PartitionKey = new PartitionKey("Account1"),
        MaxItemCount = 1
    });

// Clean up.
await database.GetContainer(containerId).DeleteContainerAsync();
await database.DeleteAsync();
```

A6-2.2 Blob Storage

https://docs.microsoft.com/en-gb/learn/modules/work-azure-blob-storage/4-manage-container-properties-metadata-dotnet

First add Azure.Storage.Blobs package to your project.

```
dotnet add package Azure.Storage.Blobs
```

create a new container and get a list of some system properties

```
//Create a CloudBlobClient for working with the Storage Account
CloudBlobClient blobClient = Common
        .CreateBlobClientStorageFromSAS(appSettings.SASToken, appSettings.AccountName);

//Get a container reference for the new container.
CloudBlobContainer container = blobClient
        .GetContainerReference(appSettings.ContainerName);
```

```
//Create the container if not already exists
container.CreateIfNotExists();

//You need to fetch the container properties before getting their values
container.FetchAttributes();
Console.WriteLine($"{container.StorageUri.PrimaryUri.ToString()}");
Console.WriteLine($"{container.Properties.ETag}");
Console.WriteLine($"{container.Properties.LastModified.ToString()}");
Console.WriteLine($"{container.Properties.LeaseStatus.ToString()}");
```

set and retrieve properties and metadata

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;

BlobServiceClient blobServiceClient = new BlobServiceClient(_connectionString);
BlobContainerClient blobContainerClient = blobServiceClient
        .GetBlobContainerClient(_blobContainerName);
BlobClient blobClient = blobContainerClient
        .GetBlobClient(_blobName);

BlobProperties blobProperties = await blobClient.GetPropertiesAsync();
await blobClient.SetHttpHeadersAsync(blobHttpHeaders);
await blobClient.SetMetadataAsync(metadata);
```

A6-3.1 Authentication and Authorization

https://docs.microsoft.com/en-us/azure/active-directory/develop/tutorial-v2-asp-webapp

Authentication with OpenID Connect

```
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.OpenIdConnect;

/// <summary>
/// Send an OpenID Connect sign-in request.
/// Alternatively, you can just decorate the SignIn method with the [Authorize] attribute
/// </summary>
public void SignIn()
{
    if (!Request.IsAuthenticated)
    {
        HttpContext.GetOwinContext().Authentication.Challenge(
            new AuthenticationProperties{ RedirectUri = "/" },
            OpenIdConnectAuthenticationDefaults.AuthenticationType);
    }
}

/// <summary>
/// Send an OpenID Connect sign-out request.
/// </summary>
public void SignOut()
{
    HttpContext.GetOwinContext().Authentication.SignOut(
            OpenIdConnectAuthenticationDefaults.AuthenticationType,
            CookieAuthenticationDefaults.AuthenticationType);
}
```

A6-3.2 Secure Cloud Solutions

First add the package Microsoft.Azure.KeyVault

```
dotnet add package Microsoft.Azure.KeyVault
```

Use the .NET Key Vault API

```
// uses service principal credentials to get tokens
var tokenCB = new AzureServiceTokenProvider()
                .KeyVaultTokenCallback;

var authCB = new KeyVaultClient
                .AuthenticationCallback(tokenCB);

// get authenticated key vault client
var client = new KeyVaultClient(authCB);

// get value of secret
var secret = await client.GetSecretAsync($"{baseURL}/{objectType}/{secretName}/{version}");
console.log($"{secret.value}";

// create new secret
secretName = "NewSecret";
await client.SetSecretAsync(baseURL, secretName, "This is a secret value.");

// delete a secret
await client.DeleteSecretAsync(baseURL, secretName);
```

A6-4.1 Caching and Content Delivery

Azure CDN caching example

```
public class HomeController: Controller
{
  public ActionResult Index()
  {
    Response.Cache.SetExpires(DateTime.Now.AddHours(1));
    Response.Cache.SetCacheability(Public);
    Response.Cache.SetLastModified(DateTime.Now);

    Return View();
  }
}
```

 redis cache expiration policy example using the C# ServiceStack.Redis package

```
public static void SetGroupChatName(string groupChatID, string chatName)
{
    using (RedisClient redisClient = new RedisClient(redisConnectionString))
    {
        //Create a key for group chat display names
        string key = groupChatID + "displayName";

        //Set the group chat display name
        redisClient.SetValue(key, chatName);

        //Set the expiration for one hour
```

```
        redisClient.Expire(key, 3600); // Note: same effect as EXPIRE command from console
    }
}
```

A6-4.2 Instrument Monitoring and Logging

Application Insights has an Instrumentation Key property that we use to connect from our SDK code. From ASP.NET Core, add NuGet package *Microsoft.ApplicationInsights.AspNetCore* . Then in the .NET Core *Startup* class, in the *ConfigureServices*() method, we add Application Insights telemetry using the *AddApplicationInsightsTelemetry()* method

```
public class startup
{
        public void ConfigureServices(IServiceCollection services)
        {
                services.AddApplicationInsightsTelemetry();
        }
}
```

This one line of code enables collecting telemetry data.

A6-5.1 Out of scope

A6-5.2 API Management

A6-5.3 Event-based Solutions

Publish to Event Grid topic

```
using Microsoft.Azure.EventGrid;
using Microsoft.Azure.EventGrid.Models;
using Microsoft.Extensions.Configuration;
using System.Collections.Generic;

string topicHostname = new Uri(configuration["EventGridTopicEndpoint"]).Host;
EventGridClient client = new EventGridClient(
            new TopicCredentials(configuration["EventGridAccessKey"]));

List<EventGridEvent> events = new List<EventGridEvent>();
```

```csharp
events.Add(new EventGridEvent()
{
        Id = Guid.NewGuid().ToString(),
        EventType = "MyCompany.Items.NewItemCreated",
        Data = new NewItemCreatedEvent() { itemName = "Item 1" },
        EventTime = DateTime.Now,
});
client.PublishEventsAsync(topicHostname, events).GetAwaiter().GetResult();
```

To send and receive events from Event Hub, install Azure.Messaging.EventHubs and
Azure.Messaging.EventHubs.Processor. Use EventHubProducerClient to send, and EventHubConsumerClient to receive.

example Event Hub send:

```csharp
using System;
using System.Text;
using System.Threading.Tasks;
using Azure.Messaging.EventHubs;
using Azure.Messaging.EventHubs.Producer;

static async Task Main()
    {
        // Create a client that you can use to send events to an event hub
        await using (var client = new EventHubProducerClient(conecstr, hubname))
        {
            // Create a batch of events
            using EventDataBatch eventBatch = await client.CreateBatchAsync();

            // Add events to the batch.
            eventBatch.TryAdd(new EventData(Encoding.UTF8.GetBytes("First event")));
            eventBatch.TryAdd(new EventData(Encoding.UTF8.GetBytes("Second event")));
            eventBatch.TryAdd(new EventData(Encoding.UTF8.GetBytes("Third event")));

            // Use the producer client to send the batch of events to the event hub
            await client.SendAsync(eventBatch);
        }
    }
```

example Event Hub receive:

```csharp
using Azure.Messaging.EventHubs;
using Azure.Messaging.EventHubs.Consumer;
using Azure.Messaging.EventHubs.Processor;

static async Task Main()
{
        // Create an event processor client to process events in the event hub
        EventProcessorClient processor = new EventProcessorClient(
                storageClient,
                consumerGroup,
                ehubNamespaceConnectionString,
                eventHubName);

        // Register handlers for processing events and handling errors
        processor.ProcessEventAsync += ProcessEventHandler;
        processor.ProcessErrorAsync += ProcessErrorHandler;

        // Start the processing
        await processor.StartProcessingAsync();

        // Wait for 30 seconds for the events to be processed
        await Task.Delay(TimeSpan.FromSeconds(30));

        // Stop the processing
        await processor.StopProcessingAsync();
}
```

```
static async Task ProcessEventHandler(ProcessEventArgs eventArgs)
{
        // Write the body of the event to the console window
        Console.WriteLine($"{Encoding.UTF8.GetString(eventArgs.Data.Body.ToArray())}");

        // Update checkpoint so that we only receive new events next time
        await eventArgs.UpdateCheckpointAsync(eventArgs.CancellationToken);
}

static Task ProcessErrorHandler(ProcessErrorEventArgs eventArgs)
{
        // Write details about the error to the console window
        Console.WriteLine(eventArgs.Exception.Message);
        return Task.CompletedTask;
}
```

A6-5.4 Message-based Solutions

send message to Service Bus queue

```
using Microsoft.Azure.ServiceBus;

var myConnectionString = ConfigurationManager.Get("MyConnection");
var myQueueName = "MyQueue";
queueClient = new QueueClient(myConnectionString, myQueueName);

var encodedMessage = new Message(Encoding.UTF8.GetBytes("Hello, World!"));
await queueClient.SendAsync(encodedMessage);
await queueClient.CloseAsync();
```

receive a message from Service Bus queue

```
queueClient.RegisterMessageHandler(MessageHandler, messageHandlerOptions);

public void MessageHandler()
{
        await queueClient.CompleteAsync(message.SystemProperties.LockToken);
        await queueClient.CloseAsync();
}
```

To interact with Azure Storage queues, add package Azure.Storage.Queues

send a message to Azure Storage Queue

```
CloudStorageAccount account = CloudStorageAccount.Parse(connectionString);
CloudQueueClient client = account.CreateCloudQueueClient();
CloudQueue queue = client.GetQueueReference("myqueue");
await queue.CreateIfNotExistsAsync();
var message = new CloudQueueMessage("your message here");
await queue.AddMessageAsync(message);
```

receive a message from Azure Storage Queue

```
CloudQueue queue;
CloudQueueMessage message = await queue.GetMessageAsync();
if (message != null)
{
    // Process the message
    await queue.DeleteMessageAsync(message);
}
```

Appendix 7 – Authentication

Azure AD B2C

https://docs.microsoft.com/en-us/azure/active-directory-b2c/

https://docs.microsoft.com/en-gb/azure/active-directory-b2c/overview

https://docs.microsoft.com/en-us/azure/active-directory-b2c/technical-overview

https://docs.microsoft.com/en-gb/azure/active-directory-b2c/configure-authentication-sample-web-app

https://docs.microsoft.com/en-gb/azure/active-directory-b2c/add-sign-up-and-sign-in-policy

https://docs.microsoft.com/en-us/learn/modules/enable-external-access-with-b2c/

https://www.youtube.com/watch?v=RNfJW8Fv10U/

https://docs.microsoft.com/en-gb/azure/active-directory-b2c/enable-authentication-web-application

https://docs.microsoft.com/en-us/learn/paths/m365-identity-associate/

https://docs.microsoft.com/en-us/learn/modules/identity-secure-custom-api/

User flows are pre-defined and easily configurable policies. They define the steps of the user experience as well as the interactions between Azure AD B2C and various components. Your application can call one or more policies depending on whether the user would like to sign-up, sign-in, reset their password or do something else. An application registered with an Azure AD B2C tenant triggers a user flow by using a standard open ID connect (OIDC) authentication request that includes a policy parameter in the URL. The policy engine (PE) looks at the policy to determine what steps need to be executed and in what order. B2C has a core store to store user objects. The B2C authentication services component uses the core store to validate credentials.

To add Microsoft Authentication Library (MSAL) to an MVC application, add Microsoft.Identity.Web and Microsoft.Identity.Web.UI NuGet packages.

In `Startup/ConfigureServices()` you wire up the communication for open id connect with:

```
services
  .AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
  .AddMicrosoftIdentityWebApp(Configuration.GetSection("AzureAd"));
```

Where "`AzureAd`" is the key for the relevant configuration section. And in `Startup/Configure()` you add:

```
app.UseAuthentication();
app.UseAuthorization();
```

Then set up your application settings with:

```
"AzureAd": {
  "Instance": "https://robdasb2c.b2clogin.com",
  "ClientId": "22222222-2222-2222-2222-222222222222",
  "Domain": "robdasb2c.onmicrosoft.com",
  "SignUpSignInPolicyId": "B2C_1_SignUpAndSignIn",
  "CallbackPath": "/signin-oidc",
},
```

In this case you are referencing a user flow policy called `SignUpSignInPolicyId` which you have to set up in the Azure AD B2C tenant. The client ID is a GUID that references the application that is registered to your Azure AD B2C tenant.

The call back path is the URI path that comes at the end of the redirect URI for your application registration in your Azure AD B2C tenant. For example: `https://localhost:5001/signin-oidc`

Microsoft Graph

https://github.com/Azure-Samples/active-directory-aspnetcore-webapp-openidconnect-v2/tree/master/2-WebApp-graph-user/2-1-Call-MSGraph

https://github.com/AzureAD/microsoft-identity-web

https://www.youtube.com/watch?v=EBbnpFdB92A

The ASP.NET Core client web app uses the Microsoft.Identity.Web to sign a user in, and obtain a JWT access Tokens from Azure AD. The access token is used by the client app as a bearer token to call Microsoft Graph.
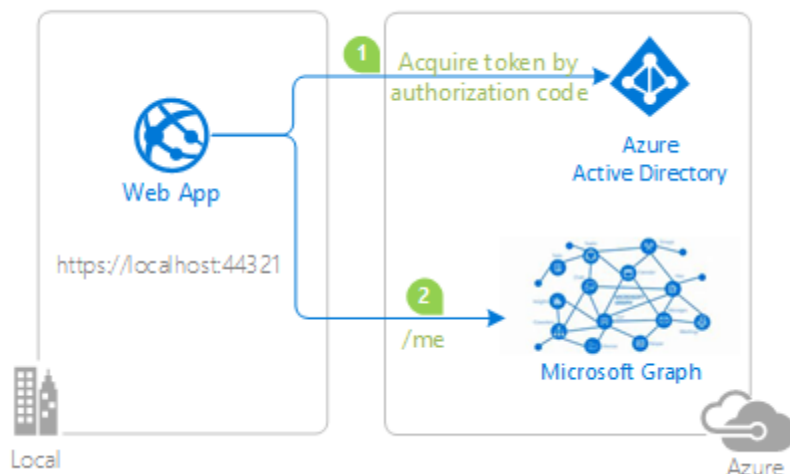
Permissions

Microsoft identity platform supports two types of permissions:

- *Delegated permissions* are used by apps that have a signed-in user present.
- *Application permissions* are used by apps that run without a signed-in user present.

Delegated permissions are provided to the application by the user so the app can perform actions on their behalf. This doesn't give permissions to the app, instead the user is simply allowing the app to act on their behalf using their permissions. For these permissions, the effective permissions of your app are the intersection of the delegated permissions the app has been granted and the privileges of the currently signed-in user. In other words, the app can never have more privileges than the signed-in user.

Best practices for requesting permissions:

- Only ask for the permissions required for app functionality that has already been implemented.
- When requesting permissions for app functionality, you should request the least-privileged access.
- Apps should gracefully handle scenarios where the user doesn't grant consent to the app when permissions are requested.

There are two ways to define the permissions an app needs that drives what permissions are listed in the consent dialog:

- *Static consent*: the permissions are defined in the app's registration within the Azure AD admin center
- *Dynamic consent*: when the app requests an access token, it specifies the permissions it needs in the scope property. If the user hasn't already consented to those permissions, they'll go through the consent experience for the additional permissions. This enables developers to only ask users for the permissions the app needs at that time.

Client credentials grant flow

Microsoft identity platform
https://login.microsoftonline.com/common

/oauth2/v2.0/token   /admindconsent

Web Server   Web API

Recommended: Sign the user into the application

Redirects admin user to grant permissions

Admin reviews & grants permissions

Returns response w/tenantID

Request token, providing client_secret as credentials

Returns token

Calls Web API with token in Authorization header

Validates token

Returns secure data to web server app

On behalf of grant flow

Application   Microsoft identity platform   Web API A   Web API B

1

User is authenticated. Application sends access token A to Web API A

2 Request access token for Web API B, providing token A, client ID and secret

3 Request access token B

Call Web API B with access token B in authorization header

4

5 Return data from secured resource

Role Based Access Control (RBAC)
https://docs.microsoft.com/en-us/learn/modules/identity-users-groups-approles/

https://docs.microsoft.com/en-us/learn/paths/m365-identity-associate/

TrainingContent/Identity/05 Users Groups Roles at master · OfficeDev/TrainingContent

The administrator assigns roles to different users to control who can access to what content and functionality. Using RBAC with Application Roles and Role Claims, developers can securely enforce authorization in their apps. Users are managed from the Azure AD admin center (https://aad.portal.azure.com) including users within your organization and those added as guests in business-to-business (B2B) scenarios. When you register an app in Azure AD, you specify what types of accounts it will support. While administrators can control which users have access to an app, developers can account for specific users within the app using RBAC. Developers first create a *role definition* within the app's registration in the Azure AD admin center. The code within the app can check if the user has access to certain resources based on the roles assigned to the user.

Appendix 8 – RESTful API

https://www.youtube.com/watch?v=0pcM6teVdKk

RESTful Services: architectural pattern for creating an API using HTTP as its underlying communication method. REST specifies a set of constraints that the system should adhere to. The REST constraints are:

- Client Server – separation of concerns between client and server that separates the evolution of the client's logic from the server-side logic
- Stateless – the communication between the client and server must be stateless between requests so that nothing is stored on the server relating to the client, the request from the client should contain all information required by the server to process the request. This ensures that each request can be treated independently by the server
- Cacheable – when the data does not change that often, the server lets the client know how long the data is good for so that the client does not have to go back to the server for the same data over and over again
- Uniform interface – defines resources in terms of data entities where each resource has a specific URI, and defines how the HTTP verbs *post*, *get*, *put*, and *delete* tell the server what to do with the resource; these verbs generally correspond to the CRUD actions that can be performed and a database table row: *create*, *read, update*, and *delete*.
- Layered system
- Code on demand (optional)

A RESTful API request contains a verb, request headers (additional information about the request), request body (the data to be sent to the server). The corresponding server response contains a response body (data sent as a response from the server), and a response status code.

Appendix 9 – Study Notes for 2023 Renewal

After achieving Azure developer certification, you must renew it every year by taking a renewal exam.

This appendix contains my 2023 renewal exam study notes.

Skills measured in renewal assessment:

- Explore the Microsoft identity platform
- Implement authentication by using the Microsoft Authentication Library
- Explore Microsoft Graph
- Implement Azure Key Vault
- Implement Azure App Configuration
- Monitor app performance
- Explore Azure Event Grid
- Explore Azure Event Hubs

Microsoft Learn references for skills measured:

- [Renewal for Microsoft Certified: Azure Developer Associate](#)
- [1 Explore the Microsoft identity platform](#)
- [2 Implement authentication by using the Microsoft Authentication Library](#)
- [3 Explore Microsoft Graph](#)
- [4 Implement Azure Key Vault](#)
- [5 Implement Azure App Configuration](#)
- [6 Monitor app performance](#)
- [7 Explore Azure Event Grid](#)
- [8 Explore Azure Event Hubs](#)

Additional related Microsoft Learn references that are relevant to the 2023 renewal exam:

- Microsoft identity platform authentication libraries (MSAL)
  https://learn.microsoft.com/en-us/azure/active-directory/develop/reference-v2-libraries
- Microsoft OData query options
  https://learn.microsoft.com/en-us/odata/concepts/queryoptions-overview
- Microsoft identity platform permissions and consent
  https://learn.microsoft.com/en-us/azure/active-directory/develop/permissions-consent-overview
- Microsoft Graph paging
  https://learn.microsoft.com/en-us/graph/paging
- Azure Identity client library for .NET
  https://learn.microsoft.com/en-us/dotnet/api/overview/azure/identity-readme?view=azure-dotnet
- Authentication flow in MSAL
  https://learn.microsoft.com/en-us/azure/active-directory/develop/msal-authentication-flows
- Requesting permissions through consent
  https://learn.microsoft.com/en-us/azure/active-directory/develop/consent-types-developer

When registering an app with AAD tenant, an application object, a globally unique app ID, and a service principal object are automatically created. The application object resides in the AAD tenant where the app was registered.

Security principal – defines the access policy and permissions in the AAD tenant.

Application object – a template or blueprint to create one or more service principal objects. Has:

- 1:1 relationship with the software application
- 1:many relationship with its corresponding service principal objects

Service principal object – to access resources that are secured by an AAD tenant. Has 3 types:

- Application service principal – local representation of a global application object in a tenant
- Managed identity service principal – provide an identity for applications to use when connecting to resources that support AAD authentication
- Legacy service principal – an app created before app registrations were introduced

OAuth 2.0 Scopes are permissions needed by app to perform their functions – represented by application ID URI and permission value. Example: Microsoft Graph application with permission to read users calendar is represented by `https://graph.microsoft.com/Calendars.Read`

Permission types:

- Delegated permissions – used by apps that have user or admin consent and have a signed-in user present. The app is delegated to act as the user when making calls to the target resource.
- Application permission – used by apps that run without a signed-in user present. Only an administrator can consent to application permissions.

Consent types:

- Static user consent – specify all the permissions in the app's configuration. Static permissions also enable administrators to consent on behalf of all users in the organization.
- Incremental and dynamic user consent – ask for a minimum set of permissions upfront and request more over time as the customer uses additional app features. If the user hasn't yet consented to new scopes added to the request, they'll be prompted to consent only to the new permissions.
- Admin consent – ensures that administrators have additional controls before authorizing access to highly privileged data. Requires the static permissions registered for the app.

In an authorization request, an app can request permissions using the scope query parameter, for example:

```
GET https://login.microsoftonline.com/authorize?scope=https://graph.microsoft.com/calendars.read
```

After the user enters their credentials, Microsoft identity checks for a matching record of consent. If the user hasn't consented, Microsoft identity asks the user to grant the requested permissions.

Conditional Access Policy – a tool that AAD uses to allow (or deny) access to resources based on identity signals. These signals include who the user is, where the user is, and what device the user is requesting access from. During sign-in, Conditional Access collects signals from the user, makes decisions based on those signals, and then enforces that decision by allowing or denying the access request or challenging for a multifactor authentication response.

Based on these signals, the decision might be to allow full access if the user is signing in from their usual location. If the user is signing in from an unusual location or a location that's marked as high risk, then access might be blocked entirely or possibly granted after the user provides a second form of authentication.

Conditional Access enables you to protect services in a multitude of ways including:

- Multifactor authentication
- Allowing only Intune enrolled devices to access specific services
- Restricting user locations and IP ranges

Only in certain cases when an app indirectly or silently requests a token for a service does an app require code changes to handle Conditional Access challenges. Apps performing the on-behalf-of flow require code to handle Conditional Access challenges. For example, you are building an app that uses a middle tier service to access a downstream API. An admin applies a policy to the downstream API. When an end user signs in, the app requests access to the middle tier and sends the token. The middle tier performs on-behalf-of flow to request access to the downstream API. At this point, a claims "challenge" is presented to the middle tier. Code change is needed for the middle tier to send the challenge back to the app.

A9-2 Implement authentication by using the Microsoft Authentication Library (MSAL)

MSAL can be used to provide secure access to Microsoft APIs, third-party web APIs, or your own web API. Using MSAL, a token can be acquired from a number of application types: web applications, web APIs, single-page apps (JavaScript), mobile and native applications, and daemons and server-side applications.

MSAL provides many authentication flows including:

Flow	Description
Authorization code	Enables a client application to obtain authorized access to protected resources like web APIs.
Client credentials	Service applications run without user interaction
On-behalf-of	The application calls a service/web API, which in turns calls downstream service such as Microsoft Graph. The intent is to pass a user's identity and permissions through the request chain.
Implicit	Used in browser-based applications
Device code	Enables sign-in to a device by using another device that has a browser
Integrated Windows	Windows computers silently acquire an access token when they are domain joined
Interactive	Mobile and desktops applications call Microsoft Graph in the name of a user
Username/password	The application signs in a user by using their username and password

Application categories for acquiring security tokens:

- **Public client applications**: run on devices or desktop computers or in a web browser. They're not trusted to safely keep application secrets, so they only access APIs on behalf of the user (public client flows). Public clients can't hold configuration-time secrets, so they don't have client secrets.
- **Confidential client applications**: run on servers (web apps, web API apps, or even service/daemon apps). They're difficult to access, so they're capable of keeping an application secret. Confidential clients can hold configuration-time secrets. Each instance of the client has a distinct configuration (including client ID and client secret).

With MSAL.NET, the way to instantiate an application is by using the application builders: *PublicClientApplicationBuilder* and *ConfidentialClientApplicationBuilder*.

You first need to register your app so that it can be integrated with the Microsoft identity platform. After registering, you may need:

- The client ID
- The identity provider URL (the instance) and the sign-in audience for your application. These two parameters are collectively known as the authority.
- The tenant ID if you are writing a line of business application solely for your organization (also named single-tenant application).
- The application secret (client secret) or certificate, if it's a confidential client app.
- For web apps, and sometimes for public client apps (in particular when your app needs to use a broker), you'll have also set the redirectUri where the identity provider will contact back your application with the security tokens.

Initializing *public* client applications:

```
IPublicClientApplication app = PublicClientApplicationBuilder.Create(clientId).Build();
```

Initializing *confidential* client applications:

```
string redirectUri = "https://myapp.azurewebsites.net";
IConfidentialClientApplication app = ConfidentialClientApplicationBuilder.Create(clientId)
    .WithClientSecret(clientSecret)  // builder modifier – set client secret to identify the app
    .WithRedirectUri(redirectUri )   // builder modifier - Overrides default redirect URI
    .Build();
```

Builder modifiers – ".with" methods, are applied as modifiers to the application builder.

For public or confidential clients:

```
.WithAuthority()
.WithTenantId(string tenantId)
.WithClientId(string)
.WithRedirectUri(string redirectUri)
.WithComponent(string)
.WithDebugLoggingCallback()
.WithLogging()
.WithTelemetry(TelemetryCallback telemetryCallback)
```

Specifically for confidential clients:

```
.WithCertificate(X509Certificate2 certificate)
.WithClientSecret(string clientSecret)
```

Details:

https://learn.microsoft.com/en-us/training/modules/implement-authentication-by-using-microsoft-authentication-library/3-initialize-client-applications

A9-3 Explore Microsoft Graph

Microsoft Graph is the gateway to the data and intelligence in Microsoft 365. Microsoft Graph is a RESTful web API that enables you to access Microsoft Cloud service resources. Three main components facilitate the access and flow of data:

- The Microsoft Graph API offers a *single endpoint, https://graph.microsoft.com*. You can use REST APIs or SDKs to access the endpoint.
- *Microsoft Graph connectors* work in the incoming direction, delivering data external to the Microsoft cloud into Microsoft Graph
- *Microsoft Graph Data Connect* is a set of tools for delivery of Microsoft Graph data to Azure data stores.

Query Microsoft Graph by using REST

Microsoft Graph has a RESTful web API that enables you to access Microsoft Cloud service resources. The Microsoft Graph API defines most of its resources, methods, and enumerations in the OData namespace, microsoft.graph . To read from or write to a resource such as a user or an email message, you construct a request that looks like the following:

`{HTTP method} https://graph.microsoft.com/{version}/{resource}?{query-parameters}`
where:

`{HTTP method}` – The HTTP method used on the request to Microsoft Graph.

`{version}` – The version of the Microsoft Graph API your application is using.

`{resource}` – The resource in Microsoft Graph that you're referencing.

`{query-parameters}` – options or REST method parameters to customize the response.

HTTP methods
- GET – Read data from a resource.
- POST – Create a new resource, or perform an action.
- PATCH – Update a resource with new values.
- PUT – Replace a resource with a new one.
- DELETE – Remove a resource.

Versions
- v1.0 includes generally available APIs for all production apps.
- beta includes APIs that are currently in preview (do not use beta APIs in your production apps).

Resource

A resource can be an entity or complex type. Often, top-level resources also include relationships, which you can use to access other resources, like me/messages or me/drive. You can also interact with resources using methods; for example, to send an email, use me/sendMail.

Query parameters

Query parameters can be OData system query options, or other strings that a method accepts to customize its response. For example:

`GET https://graph.microsoft.com/v1.0/me/messages?filter=emailAddress eq 'jon@contoso.com'`

System query options are $filter, $select, $orderby, $count, $top, $skip and $expand

https://learn.microsoft.com/en-us/odata/concepts/queryoptions-overview

After you make a request, a response is returned that includes:

- HTTP status code that indicates success or failure.
- Response message – The result of the operation (can be empty).
- nextLink – If your request returns a lot of data, you can page through it with the URL in `@odata.nextLink`.

Query Microsoft Graph by using SDKs

The Microsoft Graph SDKs are designed to simplify building apps that access Microsoft Graph. The SDKs include two components: a service library and a core library. The service library contains models and request builders when working with the many datasets available in Microsoft Graph. The core library provides a set of features that enhance working with all the Microsoft Graph services.

Installing Microsoft Graph .NET SDK uses NuGet packages:

- Microsoft.Graph – for accessing the *v1.0* endpoint (dependency on Microsoft.Graph.Core)
- Microsoft.Graph.Beta – for accessing the *beta* endpoint (dependency on Microsoft.Graph.Core)
- Microsoft.Graph.Core – core library for making calls to Microsoft Graph.

Create a Microsoft Graph client:

```
// Build a client application.
IPublicClientApplication publicClientApplication = PublicClientApplicationBuilder
        .Create("INSERT-CLIENT-APP-ID")
        .Build();
// Create an authentication provider by passing in a client application and graph scopes.
DeviceCodeProvider authProvider = new DeviceCodeProvider(publicClientApplication, graphScopes);
// Create a new instance of GraphServiceClient with the authentication provider.
GraphServiceClient graphClient = new GraphServiceClient(authProvider);
```

Read information from Microsoft Graph:

```
// GET https://graph.microsoft.com/v1.0/me
var user = await graphClient.Me
    .Request()
    .GetAsync();
```

Retrieve a list of entities:

```
// GET https://graph.microsoft.com/v1.0/me/messages?$select=subject,sender&$filter=<some condition>&orderBy=receivedDateTime
var messages = await graphClient.Me.Messages
    .Request()
    .Select(m => new {
        m.Subject,
        m.Sender
    })
    .Filter("<filter condition>")
    .OrderBy("receivedDateTime")
    .GetAsync();
```

Delete an entity:

```
// DELETE https://graph.microsoft.com/v1.0/me/messages/{message-id}
string messageId = "AQMkAGUy...";
var message = await graphClient.Me.Messages[messageId]
    .Request()
    .DeleteAsync();
```

Create a new entity:

```
// POST https://graph.microsoft.com/v1.0/me/calendars
var calendar = new Calendar
{
    Name = "Volunteer"
};
var newCalendar = await graphClient.Me.Calendars
    .Request()
    .AddAsync(calendar);
```

Microsoft Graph Best Practices

Authentication – to access data, present an OAuth 2.0 access token to Microsoft Graph in either of the following ways:

- The HTTP *Authorization* request header, as a *Bearer* token
- The graph client constructor, when using a Microsoft Graph client library

Use the Microsoft Authentication Library API, MSAL to acquire the access token to Microsoft Graph.

Consent and authorization best practices:

- Principle of least privilege
- Use the correct permission type based on the scenario: for apps with signed in users, use *delegated* permissions; without signed in users, use *application* permission.
- User experience – configure apps appropriately for who will be consenting, users or administrators. Use static, dynamic, or incremental consent correctly.
- Multi-tenant applications – tenant administrators can disable the ability of end users to consent, block users from accessing other users' profiles, or limit creation of self-service groups, which can result in HTTP 403 (forbidden) errors.

Handling responses reliably and predictably for your end users:

- Pagination – always handle the possibility that the responses are paged in nature, and use the @odata.nextLink property to obtain the next paged set of results, until all pages of the result set have been read.
- Evolvable enumerations – mechanism that Microsoft Graph API uses to add new enum-members to existing enumerations without causing a breaking change for applications. If you design your application to handle unknown enum-members as well, you can opt-in to receive those members by using an HTTP `Prefer` request header.

Storing data locally – make calls to Microsoft Graph to retrieve data in real time as necessary. Avoid unnecessarily caching data locally unless needed for a specific scenario, and in that case implement proper retention and deletion policies.

A9-4 Implement Azure Key Vault

Azure Key Vault is for:

- Secret management – tokens, passwords, certificates, API keys, and other secrets
- Key management – create and control the encryption keys used to encrypt your data
- Certificate management – public and private SSL/TLS certificates

Service tiers:

- Standard, which encrypts with a software key
- Premium tier, which includes hardware security module (HSM) protected keys

Benefits:

- Centralized application secrets – securely access information by using URIs
- Securely store secrets and keys – Azure RBAC (management plane) or Key Vault access policy (data plane)
- Monitor access and use logs – enable logging for your vault; you can secure and delete logs; configure your key vault to archive logs to storage accounts, stream logs to an event hub, or send logs to Azure Monitor logs
- Simplified administration of application secrets – security information must be secured, it must follow a life cycle, and it must be highly available

Authentication:

- System assigned, managed identities for Azure resources – **best practice** – Azure internally manages the application's service principal and automatically authenticates the application
- Register the application with your Azure AD tenant
- Authentication to Key Vault in application code using Key Vault SDK
- Authentication to Key Vault with REST. Access tokens are sent using the HTTP Authorization header. For example:

```
PUT /keys/MYKEY?api-version=<api_version>  HTTP/1.1
Authorization: Bearer <access_token>
```

  When an access token is not accepted or not supplied, an HTTP 401 (unauthorized) error will be returned to the client, and will include the WWW-Authenticate header containing authorization and resource parameters that may be used to obtain a valid access token

- Service principal and certificate – **not recommended** – the app owner must rotate the certificate
- Service principal and secret – **not recommended** – it's hard to automatically rotate secrets

Encryption of data in transit:

Azure Key Vault enforces Transport Layer Security (TLS) protocol to protect data when it's traveling between Azure Key Vault and clients. Perfect Forward Secrecy (PFS) protects connections between client systems and Microsoft cloud services by using unique, 2048-bit encryption keys.

Best practices:

- Use separate key vaults per application per environment
- Control access to your key vaults by allowing only authorized applications and users
- Create regular back ups of your vault on update/delete/create of objects within a Vault

- Turn on logging and alerts
- Recovery options: turn on soft-delete and purge protection to guard against force deletion of secrets

Azure CLI code examples

For convenience, initialize some Bash variables:

```
myKeyVault="CharlesTheFifth"
myResourceGroup="ElizabethTheEighth"
myLocation="canadacentral"
```

Create an Azure Key Vault:

```
az keyvault create --name $myKeyVault --resource-group $myResourceGroup --location $myLocation
```

Create a secret:

```
az keyvault secret set --vault-name $myKeyVault --name "ExamplePassword" --value "hVFkk965BuUv"
```

Retrieve a secret:

```
az keyvault secret show --name "ExamplePassword" --vault-name $myKeyVault
```

A9-5 Implement Azure App Configuration

Azure App Configuration provides a service to centrally manage application settings and feature flags. App Configuration complements Azure Key Vault, which is used to store application secrets. The easiest way to add an App Configuration store to your application is through a client library that Microsoft provides.

Benefits of Azure App Configuration

- fully managed service
- flexible key representations and mappings
- tagging with labels
- feature flag management
- security through Azure managed identities
- encryption
- integration with popular frameworks

Azure App Configuration stores configuration data as *key-value* pairs.

Keys – Namespaces can be created using hierarchical keys separated by ":" or "/". Keys can be tagged with different labels which can be queried:

```
Key = AppName:DbEndpoint & Label = Test
```

Keys stored in App Configuration are case sensitive, but some frameworks may handle keys case-insensitively (beware).

Keys can be made up of any unicode characters except the three reserved characters asterisk, comma, backslash (*,\), however reserved characters can be escaped with a backslash

Common uses for labels include using the same keys for different environments, and key value versioning. Labels can be null.

Values – You can use all unicode characters for values. There's an optional user-defined content type (for example encoding scheme) that is associated with each value. The content type can be used to help your application process the key properly.

Azure App Configuration – Feature Management

Feature flag – a variable with a binary state of on or off and an associated code block.

Feature manager – an application package that handles the lifecycle of feature flags

Filter – a rule for evaluating the state of a feature flag (examples: browser type, geographic location, time window).

Feature flags can also be stored in appsettings.json. For example:

```
"FeatureManagement": {
    "FeatureA": true, // Feature flag set to on
    "FeatureB": false, // Feature flag set to off
    "FeatureC": {
        "FilterEnabledFor": [
            {
                "Name": "Percentage",
                "Parameters": {
                    "Value": 50
                }
            }
        ]
    }
}
```

Securing app configuration data

Encrypting configuration data with customer-managed keys – when customer-managed key capability is enabled, the App Configuration instance uses a managed identity to authenticate with Azure Active Directory. It then wraps and puts the encryption key in Azure Key Vault, and caches the unwrapped key for an hour. The key is refreshed hourly.

Components needed for customer-managed keys:

- Standard tier Azure App Configuration instance
- Azure Key Vault with soft-delete and purge-protection features enabled
- An RSA or RSA-HSM key within the Key Vault: The key must not be expired, it must be enabled, and it must have both wrap and unwrap capabilities enabled

To use Azure Key Vault, Azure App Configuration does two steps:

1. Assign a managed identity to the Azure App Configuration instance
2. Grant the identity GET, WRAP, and UNWRAP permissions in the target Key Vault's access policy.

Use private endpoints for Azure App Configuration – to allow clients on a virtual network (VNet) to securely access data over a private link. Network traffic uses this private link on the Microsoft backbone network, eliminating exposure to the public internet. You need separate private endpoints for each App Configuration store. The connection string for the private endpoint is found in the Azure Portal at App Configuration store > Settings > Access Keys. Azure uses a DNS CNAME to alias a subdomain with the prefix `privatelink`, and DNS A records for the IP address of the private endpoint. From within the VNet, the endpoint URL resolves to the private endpoint of the App Configuration store. When resolved from outside the VNet, the endpoint URL resolves to the public endpoint. When you create a private endpoint, the public endpoint is disabled.

Managed identities – A managed identity from AAD allows Azure App Configuration to easily access other AAD-protected resources, such as Azure Key Vault.

- A *system-assigned identity* is tied to your configuration store. It's deleted if your configuration store is deleted. A configuration store can only have one system-assigned identity.
- A *user-assigned identity* is a standalone Azure resource that can be assigned to your configuration store. You can have multiple user-assigned identities for a configuration store.

A system-assigned identity for an App Configuration store can be created in one step:

```
// create system-assigned identity for "myAppConfigStore1"
az appconfig identity assign --name myAppConfigStore1 --resource-group myResourceGroup
```

A user-assigned identity for an App Configuration store requires two steps. First create the identity:

```
// create user-assigned identity
az identity create --resource-group myResourceGroup --name myUserAssignedIdentity
```

Then assign its resource identifier to the App Configuration store:

```
// assign the resource identifier of the identity to "myAppConfigStore2"
az appconfig identity assign \
    --name myAppConfigStore2 \
    --resource-group myResourceGroup \
    --identities /subscriptions/[subscription id]/resourcegroups \
                 /myResourceGroup/providers/Microsoft.ManagedIdentity \
                 /userAssignedIdentities/myUserAssignedIdentity
```

A9-6 Monitor app performance

Use the tools in Azure Monitor to enhance the performance and stability of your applications.

Azure Monitor – the center of monitoring in Azure

- obtains telemetry data from various data sources
- stores the data in the data platform
- acts on the data by performing actions such as analysis, alerting, and integration by streaming to external systems
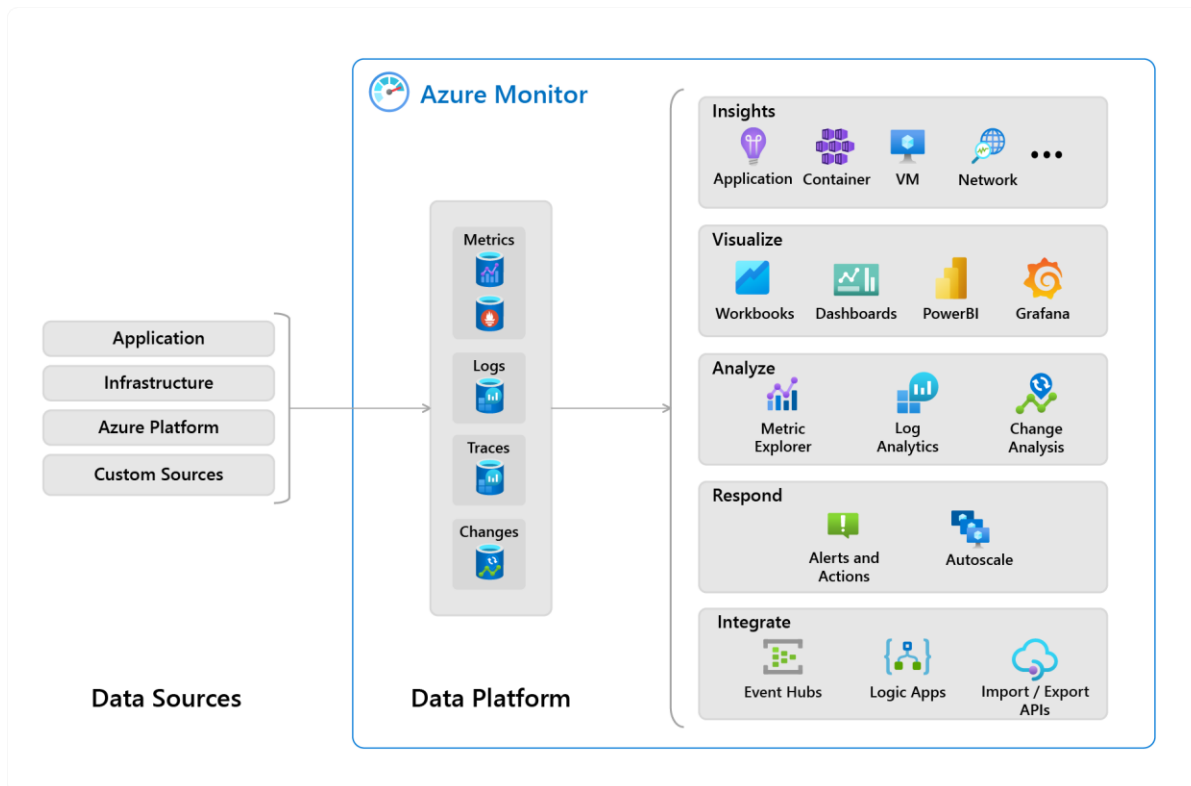
The Data Platform stores the pillars of observability:

Metrics	numerical values that describe some aspect of a system at a point in time, that are identified with a timestamp, a name, a value, and one or more labels
Logs	contain different kinds of data and may be structured or free-form text with a timestamp
Traces	series of related events that follow a user request through a distributed system
Changes	change analysis is a subscription-level observability tool that's built on the power of Azure Resource Graph

Insights provide a custom monitoring experience:

- Application Insights monitors the availability, performance, and usage of your web applications whether they're hosted in the cloud or on-premises, enabling you to diagnose errors without waiting for a user to report them
- Container Insights monitors the performance of container workloads that are hosted on Azure Kubernetes Service (AKS) and Azure Container Instances, and are available in Kubernetes through the Metrics API
- VM Insights monitors your Azure VMs at scale, analysing their performance and health and identifying dependencies on external processes.

How Application Insights works and how it collects events

Application Insights is an extension of Azure Monitor that provides Application Performance Monitoring (APM). Features:

Live Metrics	Observe activity from your deployed app in real time with no effect on the host environment.
Availability	Probe your apps external endpoint(s) to test the overall availability and responsiveness over time.
GitHub or Azure DevOps integration	Create GitHub or Azure DevOps work items in context of Application Insights data.
Usage	Reveals which features are popular and how users interact and use your app.
Smart Detection	Provides automatic failure and anomaly detection through proactive telemetry analysis
Application Map	high level top-down view of the app architecture and visual references to component health and responsiveness
Distributed Tracing	Search and visualize an end-to-end flow of a given execution or transaction

Application Insights collects Metrics and application Telemetry data, which describe application activities and health, as well as trace logging data:

- Request rates vs response times and failure rates, can be used to diagnose resourcing problems
- Dependency rates vs response times and failure rates – are external services slowing you down?
- Page views and load performance

- AJAX calls from web pages - rates, response times, and failure rates
- User and session counts
- Performance counters from server machines, such as CPU, memory, and network usage
- Host diagnostics from Docker or Azure
- Diagnostic trace logs from your app – correlate trace events with requests
- Custom events and metrics that you write yourself in the client or server code, to track business events such as items sold or games won

Log-based Metrics

Application Insights log-based metrics:

- Log-based metrics – more dimensions, better for data analysis and ad-hoc diagnostics
- Standard metrics – stored as pre-aggregated time series, better for performance

Log-based metrics (traditional metrics) developers can use the SDK to emit these events from code, or they can rely on the automatic collection of events from auto-instrumentation. When the volume of events is too high, Application Insights implements telemetry volume reduction techniques, which lowers the accuracy of the metrics.

Pre-aggregated metrics (standard metrics) are not stored as individual events but as pre-aggregated time series, so retrieving data happens much faster and requires less compute power. The collection endpoint pre-aggregates events before ingestion sampling, which means that ingestion sampling will never impact the accuracy of pre-aggregated metrics.

Instrument an app for monitoring

Application Insights is enabled through either Auto-Instrumentation (agent) or by adding the Application Insights SDK to your application code.

Auto-instrumentation is the preferred method. It requires no developer investment and eliminates future overhead related to updating the SDK. It's also the only way to instrument an application in which you don't have access to the source code. In essence, all you have to do is enable and - in some cases - configure the agent, which will collect the telemetry automatically.

Application Insights SDK is only needed in the following circumstances:

- You require custom events and metrics
- You require control over the flow of telemetry
- Auto-Instrumentation isn't available (language or platform limitations)

When using an SDK, the app and its components don't have to be hosted in Azure. The instrumentation monitors your app and directs the telemetry to an Application Insights resource, using a unique token.

In addition to the Application Insights SDKs, Application Insights also supports distributed tracing through an open source, vendor-agnostic, single distribution of libraries called OpenCensus. OpenCensus also enables the open-source community to enable distributed tracing with popular technologies like Redis, Memcached, or MongoDB.

Perform Availability Tests

After you've deployed your web app or website, you can set up Application Insights availability tests, recurring tests that run at regular intervals from points around the world. You can set up availability tests for any HTTP or HTTPS endpoint that's accessible from the public internet. You don't have to own the sight, so you can test the availability of a REST API that your service depends on.

There are 3 types of availability test:

- URL ping test – validate whether an endpoint is responding and measure performance associated with that response
- Standard test – includes SSL certificate validity, proactive lifetime check, HTTP request verb (GET, HEAD, POST, …), custom headers, and custom data associated with your HTTP request
- Custom TrackAvailability test – create custom code to run availability tests and use the TrackAvailability() method to send the results to Application Insights.

The URL ping test relies on the DNS infrastructure of the public internet to resolve the domain names of the tested endpoints. If you're using private DNS, you must ensure that the public domain name servers can resolve every domain name of your test. When that's not possible, you can use custom TrackAvailability tests instead.

Application Map – monitor performance and troubleshoot issues

Application Map helps you spot performance bottlenecks or failure hotspots across all components of your distributed application. Each node on the map represents an independently deployable part of your distributed/microservices application (component) or its dependencies; and has health KPI and alerts status. This allows you to visualize complex topologies with hundreds of components.
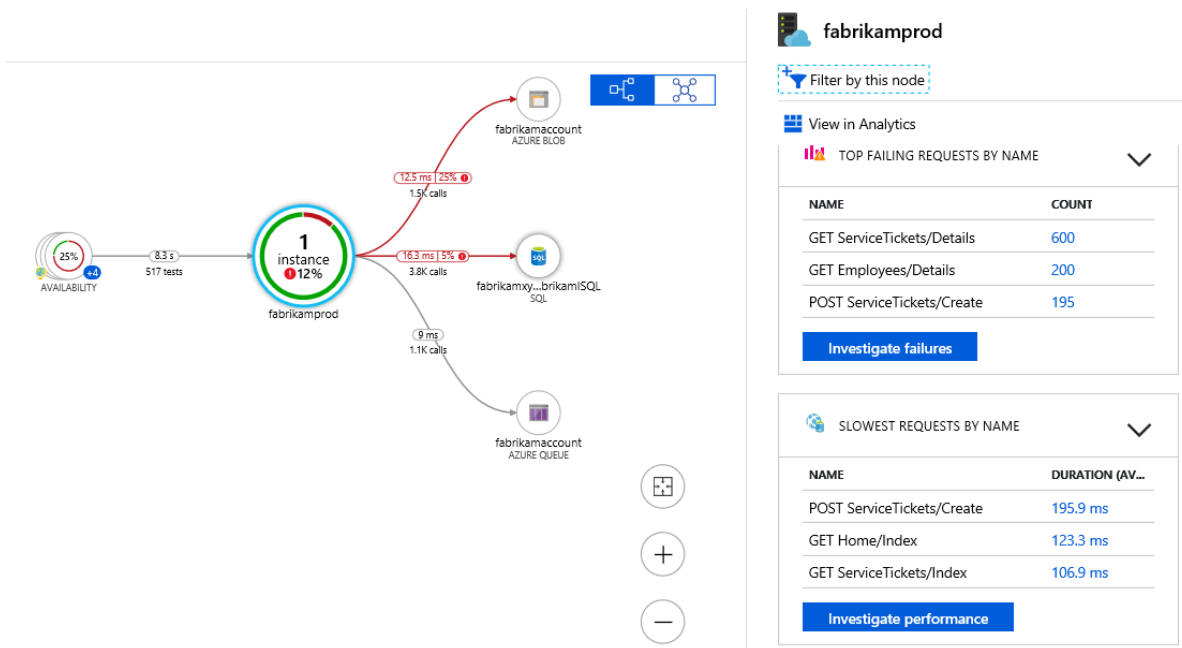
Application Map uses the cloud role name property to identify the components on the map. You can manually set or override the cloud role name and change what gets displayed on the Application Map.

A9-7 Explore Azure Event Grid

Azure Event Grid enables event-driven, reactive programming using the publish-subscribe model.
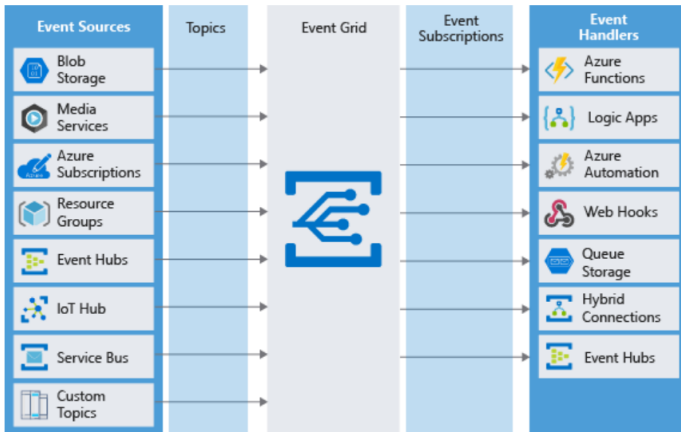
Publishers emit events, but have no expectation about how the events are handled. Subscribers decide on which events they want to handle. Event Grid has built-in support for events coming from Azure services, like storage blobs and resource groups. Event Grid also has support for your own events, using custom topics. Event Grid doesn't guarantee order for event delivery, so subscribers may receive them out of order.

Event Grid terminology:

Events – What happened. An event is the smallest amount of information that fully describes something that happened in the system.

Event sources – Where the event took place.

Topics – The endpoint where publishers send events. To respond to certain types of events, subscribers decide which topics to subscribe to:

- *System topics* are built-in topics provided by Azure services. You don't see system topics in your Azure subscription because the publishing resource owns the topics, but you can subscribe to them. As long as you have access to the resource, you can subscribe to its events.
- *Custom topics* are applications and third-party topics. When you create or are assigned access to a custom topic, you see that custom topic in your subscription.

Event subscriptions – The endpoint or built-in mechanism to route events, sometimes to more than one handler. Subscriptions are also used by handlers to intelligently filter incoming events.

Event handlers – The app or service reacting to the event. You can use a supported Azure service or your own webhook as the handler.

Event schema

Event sources send events to Event Grid in an array, which can have several event objects. Event Grid sends the events to subscribers in an array that has a single event. The following example shows the properties that are used by all event publishers:

```
[
  {
    "topic": string,
    "subject": string,
    "id": string,
    "eventType": string,
    "eventTime": string,
    "data":{
      object-unique-to-each-publisher
    },
    "dataVersion": string,
    "metadataVersion": string
  }
]
```

All events have the same following top-level data:

Property	Required	Description
topic	No	Full resource path to the event source. This field isn't writeable. Event Grid provides this value.
subject	Yes	Publisher-defined path to the event subject.
id	Yes	Unique identifier for the event.
eventType	Yes	One of the registered event types for this event source.
eventTime	Yes	The time the event is generated based on the provider's UTC time.
data	No	Event data specific to the resource provider.
dataVersion	No	The schema version of the data object. The publisher defines the schema version.
metaDataVersion	No	The schema version of the event metadata. Event Grid defines the schema of the top-level properties. Event Grid provides this value.

When publishing events to custom topics, create subjects for your events that make it easy for subscribers to know whether they're interested in the event. Subscribers use the subject to filter and route events. Consider providing the path for where the event happened, so subscribers can filter by segments of that path.

In addition to its default event schema, Azure Event Grid natively supports events in the JSON implementation of CloudEvents v1.0 and HTTP protocol binding. CloudEvents is an open specification for describing event data.

Event Grid provides durable delivery. It tries to deliver each event at least once for each matching subscription immediately.

If an error returned by the subscribed endpoint is a configuration-related error that can't be fixed with retries (for example, if the endpoint is deleted), EventGrid will either perform dead-lettering on the event or drop the event if dead-letter isn't configured.

A send-to-subscriber error doesn't cause a retry when:

Endpoint	Error
Azure Resources	400 Bad Request, 413 Request Entity Too Large, 403 Forbidden
Webhook	400 Bad Request, 413 Request Entity Too Large, 403 Forbidden, 404 Not Found, 401 Unauthorized
Note: if Dead-Letter isn't configured, events will be dropped for the above errors	

Otherwise, a retry occurs after 30 seconds, followed by exponential backoff retries.

If the endpoint responds within 3 minutes, Event Grid will attempt to remove the event from the retry queue on a best effort basis but duplicates may still be received. Event Grid adds a small randomization to all retry steps and may opportunistically skip certain retries if an endpoint is consistently unhealthy, down for a long period, or appears to be overwhelmed. The functional purpose of delayed delivery is to protect unhealthy endpoints and the Event Grid system.

You can customize the retry policy by using the following two configurations:

- Maximum number of attempts
- Event time-to-live (TTL)

Example:

```
az eventgrid event-subscription create \
  -g gridResourceGroup \
  --topic-name <topic_name> \
  --name <event_subscription_name> \
  --endpoint <endpoint_URL> \
  --max-delivery-attempts 18
```

You can configure Event Grid to batch events for delivery for improved performance in high-throughput scenarios using the following settings:

- *Max events per batch* – Event Grid doesn't delay events to create a batch if fewer events are available
- *Preferred batch size in kilobytes* – the batch size may be smaller if more events aren't available, or larger if a single event is larger than the preferred size

To enable dead-lettering, you must specify a storage account to hold undelivered events when creating the event subscription.

You can set custom HTTP headers on the events that are delivered to the following destinations:

- Webhooks
- Azure Service Bus topics and queues
- Azure Event Hubs
- Relay Hybrid Connections

This capability allows you to set custom headers that are required by a destination.

Roles

Azure Event Grid allows you to control the level of access given to different users with Azure RBAC. Event Grid provides the following built-in roles:

- *Event Grid Subscription Reader* – read Event Grid event subscriptions
- *Event Grid Subscription Contributor* – manage Event Grid event subscription operations
- *Event Grid Contributor* – create and manage Event Grid resources
- *Event Grid Data Sender* – lets you send events to Event Grid topics

The Event Grid Subscription Reader and Event Grid Subscription Contributor roles are for managing event subscriptions.

Permissions

Event handler – If you're using an event handler (other than a WebHook) you need write access to that resource. This permissions check prevents an unauthorized user from sending events to your resource.

Event source – You must have the `Microsoft.EventGrid/EventSubscriptions/Write` permission on the resource that is the event source. You need this permission because you're writing a new subscription at the scope of the resource. The resource requiring permission differs based on whether you're subscribing to a system topic or custom topic:

- System topics – need permission at the scope of the resource publishing the event. `/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/{resource-provider}/{resource-type}/{resource-name}`
- Custom topics – need permission at the scope of the event grid topic. `/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/Microsoft.EventGrid/topics/{topic-name}`

For *system topics*, if you are not the owner or contributor of the source resource, you need permission to write a new event subscription at the scope of the resource publishing the event. For example, to subscribe to an event on a storage account named myacct, you need the Microsoft.EventGrid/EventSubscriptions/Write permission on: /subscriptions/####/resourceGroups/testrg/providers/Microsoft.Storage/storageAccounts/myacct

For *custom topics*, you need permission to write a new event subscription at the scope of the event grid topic. For example, to subscribe to a custom topic named mytopic, you need the Microsoft.EventGrid/EventSubscriptions/Write permission on: /subscriptions/####/resourceGroups/testrg/providers/Microsoft.EventGrid/topics/mytopic

Using webhooks to receive events

Event Grid requires you to validate, i.e. prove ownership, of your Webhook endpoint before it starts delivering events to that endpoint. This requirement prevents a malicious user from flooding your endpoint with events.

When you use any of these Azure services:

- Azure Logic Apps with Event Grid Connector
- Azure Automation via webhook
- Azure Functions with Event Grid Trigger

the Azure infrastructure automatically handles the validation. For any other type of webhook endpoint, your endpoint code needs to participate in a validation handshake with Event Grid. There are two kinds of handshake:

Synchronous handshake: At the time of event subscription creation, Event Grid sends a subscription validation event to your endpoint. The data portion of this event includes a validationCode property. Your application verifies the validation event by returning the validation code in the response synchronously along with an HTTP 200 status code.

Asynchronous handshake: In certain cases, you can't return the ValidationCode synchronously. For example, if you use a third-party service, you can't programmatically respond. In this case, Event Grid supports a manual validation handshake. Event Grid sends a validationUrl property in the data portion of the subscription validation event. To complete the handshake, find that URL in the event data and do a GET request to it. The provided URL is valid for 5 minutes. If you don't complete the manual validation within 5 minutes, you will have to create the event subscription again.

If the endpoint returns 200 but doesn't return back a validation response synchronously, the mode is transitioned to the manual validation mode. If there's a GET on the validation URL within 5 minutes, the validation handshake is considered to be successful.

Filtering

When creating a subscription, you can add a filter. You can filter on:

- Event types
- Subject begins with or ends with
- Advanced fields and operators

The JSON syntax for filtering *by event type* is:

```
"filter": {
  "includedEventTypes": [
    "Microsoft.Resources.ResourceWriteFailure",
    "Microsoft.Resources.ResourceWriteSuccess"
  ]
}
```

When publishing events to custom topics, create *subjects* for your events that make it easy for subscribers to know whether they're interested in the event. Subscribers use the subject property to filter and route events. The JSON syntax for filtering *by subject* is:

```
"filter": {
```

```
  "subjectBeginsWith": "/blobServices/default/containers/mycontainer/log",
  "subjectEndsWith": ".jpg"
}
```

To filter by values in the data fields and specify the comparison operator, use the *advanced filtering* option. In advanced filtering, you specify the:

- operator type - The type of comparison.
- key - The field that you're using for filtering. It can be a number, boolean, or string.
- value or values - The value or values to compare to the key.

The JSON syntax for using *advanced filters* is:

```
"filter": {
  "advancedFilters": [
    {
      "operatorType": "NumberGreaterThanOrEquals",
      "key": "Data.Key1",
      "value": 5
    },
    {
      "operatorType": "StringContains",
      "key": "Subject",
      "values": ["container1", "container2"]
    }
  ]
}
```

A9-8 Explore Azure Event Hubs

Azure Event Hubs is an event ingestor – a component or service that sits between event publishers and event consumers to decouple the production of an event stream from the consumption of those events.

Event Hub Terminology

- **Event Hubs client** is the primary interface for developers interacting with the Event Hubs client library.
- **Event Hubs producer** is a type of client that serves as a source of data.
- **Event Hubs consumer** is a type of client which reads information from the Event Hub.
- **Partition** is an ordered sequence of events that is held in an Event Hubs. Azure Event Hubs provides message streaming through a partitioned consumer pattern.
- **Consumer group** is a view of an entire Event Hubs that enable multiple consuming applications to each have a separate view of the event stream, and to read the stream independently at their own pace and from their own position within the stream.
- **Event receivers** are entities that read event data from an Event Hubs.
- **Throughput units** or **Processing units** are pre-purchased units of throughput capacity.
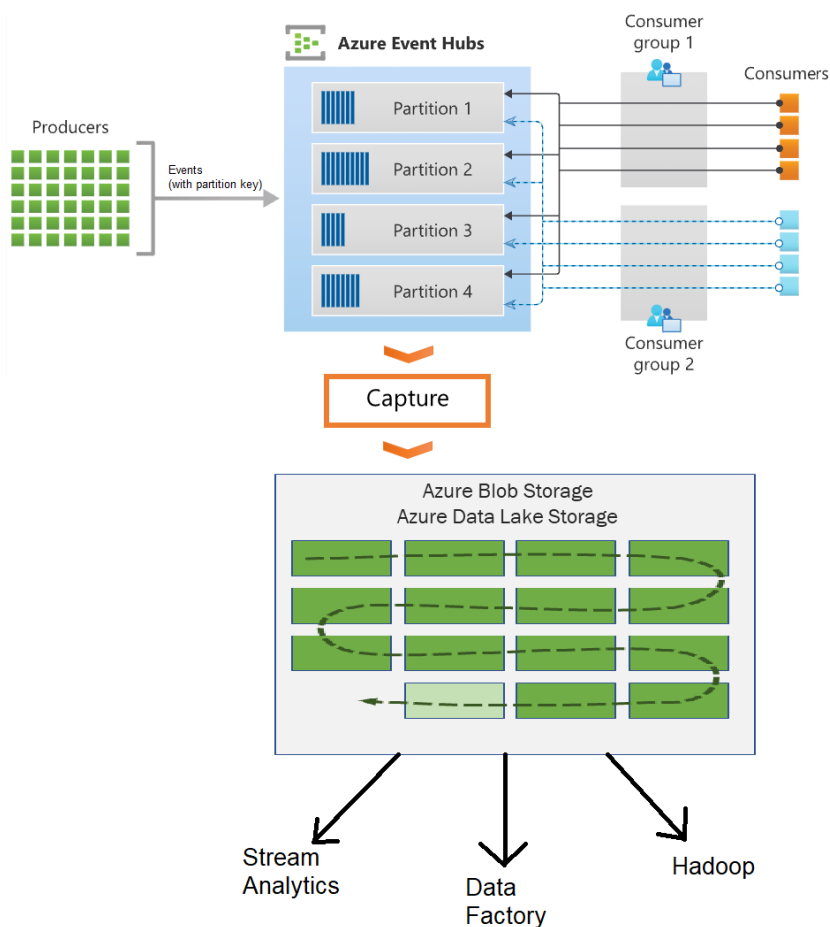
Event Hubs Capture

Event Hubs enables you to automatically capture the streaming data in Event Hubs in an Azure Blob storage or Azure Data Lake Storage account of your choice. Event Hubs Capture enables you to specify your own Azure Blob storage

account and container, or Azure Data Lake Store account, which are used to store the captured data. Event Hubs Capture enables you to set up a window to control capturing. Each partition captures independently and writes a completed block blob at the time of capture, named for the time at which the capture interval was encountered. Storage naming convention:

```
{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}
```
For example:

```
https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/myeventhub/0/2017/12/08/03/03/17.avro
```

Event Hubs traffic is controlled by throughput units. A throughput unit is 1 MB per second or 1000 events per second of ingress and twice that amount of egress. Standard Event Hubs can be configured with 1-20 throughput units, and you can purchase more. To make it easier for your downstream processing to know that Event Hub Capture is working, Event Hubs writes empty files when there is no data. This process provides a predictable cadence and marker.

Scale processing applications

To scale your event processing application, you can run multiple instances of the application and have them balance the load among themselves. Use the .NET EventProcessorClient class. Clients will automatically manage distribution and balancing of work as instances become available or unavailable for the group. Each event processor is given a unique identifier and claims ownership of partitions by adding or updating an entry in a checkpoint store. When you create an event processor, you specify the functions that will process events and errors. Each call to the function that processes events delivers a single event from a specific partition. Your code must handle this event. If you want to make sure the consumer processes every message at least once, you need to write your own code with retry logic.

Checkpointing is when an event processor marks the position of the last successfully processed event within a partition. You can use checkpointing to both mark events as "complete" by downstream applications and provide resiliency when an event processor goes down.

Thread safety is the responsibility of the consumer processor instance. Events from different partitions can be processed concurrently, but any shared state that is accessed across partitions must be synchronized.

Control access to events

Azure Event Hubs supports both Azure Active Directory and shared access signatures (SAS) to handle both authentication and authorization. Built-in AAD roles for authorizing access to Event Hubs data:

- Azure Event Hubs Data Owner: Use this role to give complete access to Event Hubs resources.
- Azure Event Hubs Data Sender: Use this role to give send access to Event Hubs resources.
- Azure Event Hubs Data Receiver: Use this role to give receiving access to Event Hubs resources.

To authorize a request to Event Hubs service from a managed identity in your application, you need to configure Azure role-based access control settings for that managed identity. When the appropriate AAD role is assigned to a managed identity, the managed identity is granted access to Event Hubs data at the appropriate scope.

Shared Access Signatures (SAS) can be used to authorize access to Event Hubs publishers but not to consumers.

Typically, an event hub employs one publisher per client. Each Event Hubs client is assigned a unique token with a SAS key. A client that holds a token can only send to one publisher. If multiple clients share the same token, then each of them shares the publisher. Clients aren't aware of the SAS key, which prevents clients from manufacturing tokens. Clients operate on the same tokens until they expire.

Data is consumed from Event Hubs using consumer groups. While SAS policy gives you granular scope, this scope is defined only at the namespace, event hub instance, or topic level, and not at the consumer level.

Event Hubs Client Library

To interact with Azure Event Hub, use Azure.Messaging.EventHubs

To find out what partitions are available:

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString, eventHubName))
{
    string[] partitionIds = await producer.GetPartitionIdsAsync();
}
```

To publish events to an Event Hub:

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString, eventHubName))
{
    using EventDataBatch eventBatch = await producer.CreateBatchAsync();
    eventBatch.TryAdd(new EventData(new BinaryData("First")));
    eventBatch.TryAdd(new EventData(new BinaryData("Second")));

    await producer.SendAsync(eventBatch);
}
```

To read events from an Event Hub:

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

// When an Event Hub is created, it provides a default consumer group that can be used to get started
string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup, connectionString, eventHubName))
{
    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in consumer.ReadEventsAsync(cancellationSource.Token))
    {
        // At this point, the loop will wait for events to be available in the Event Hub.  When an event
        // is available, the loop will iterate with the event that was received.  Because we did not
        // specify a maximum wait time, the loop will wait forever unless cancellation is requested using
        // the cancellation token.
    }
}
```

Read all published events for the first partition of the Event Hub:

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;
```

```
await using (var consumer = new EventHubConsumerClient(consumerGroup, connectionString, eventHubName))
{
    EventPosition startingPosition = EventPosition.Earliest;
    string partitionId = (await consumer.GetPartitionIdsAsync()).First();

    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in consumer.ReadEventsFromPartitionAsync(
                                        partitionId, startingPosition, cancellationSource.Token))
    {
        // At this point, the loop will wait for events to be available in the partition.  When an event
        // is available, the loop will iterate with the event that was received.  Because we did not
        // specify a maximum wait time, the loop will wait forever unless cancellation is requested using
        // the cancellation token.
    }
}
```

Process events from Azure Event Hub:

```
var cancellationSource = new CancellationTokenSource();
cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

// Since the EventProcessorClient has a dependency on Azure Storage blobs for persistence of its state,
// you'll need to provide a BlobContainerClient for the processor, which has been configured for the
// storage account and container that should be used.
var storageConnectionString = "<< CONNECTION STRING FOR THE STORAGE ACCOUNT >>";
var blobContainerName = "<< NAME OF THE BLOB CONTAINER >>";

var eventHubsConnectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";
var consumerGroup = "<< NAME OF THE EVENT HUB CONSUMER GROUP >>";

Task processEventHandler(ProcessEventArgs eventArgs) => Task.CompletedTask;
Task processErrorHandler(ProcessErrorEventArgs eventArgs) => Task.CompletedTask;

var storageClient = new BlobContainerClient(storageConnectionString, blobContainerName);
var processor = new EventProcessorClient(
                    storageClient, consumerGroup, eventHubsConnectionString, eventHubName);

processor.ProcessEventAsync += processEventHandler;
processor.ProcessErrorAsync += processErrorHandler;

await processor.StartProcessingAsync();

try
{
    // The processor performs its work in the background; block until cancellation
    // to allow processing to take place.

    await Task.Delay(Timeout.Infinite, cancellationSource.Token);
}
catch (TaskCanceledException)
{
    // This is expected when the delay is canceled.
}

try
{
    await processor.StopProcessingAsync();
}
finally
{
    // To prevent leaks, the handlers should be removed when processing is complete.

    processor.ProcessEventAsync -= processEventHandler;
    processor.ProcessErrorAsync -= processErrorHandler;
}
```

Appendix 10 – Study Notes for 2024 Renewal

After achieving Azure developer certification, you must renew it every year by taking a renewal exam.

This appendix contains my 2023 renewal exam study notes.

Skills measured in renewal assessment:

- Explore Azure Functions
- Develop Azure Functions
- Implement Azure Key Vault
- Implement Azure App Configuration
- Monitor app performance
- Implement Azure Container Apps
- Manage container images in Azure Container Registry
- Run container images in Azure Container Instances
- Work with Azure Cosmos DB
- Consume an Azure Cosmos DB for NoSQL change feed using the SDK

Microsoft Learn:

- [Renewal for Microsoft Certified: Azure Developer Associate](#)
- [Explore Azure Functions](#)
- [Develop Azure Functions](#)
- [Implement Azure Key Vault](#)
- [Implement Azure App Configuration](#)
- [Monitor app Performance](#)
- [Implement Azure Container Apps](#)
- [Manage container images in Azure Container Registry](#)
- [Run container images in Azure Container Instances](#)
- [Work with Azure Cosmos DB](#)
- [Consume an Azure Cosmos DB for NoSQL change feed using the SDK](#)

A10-1 Explore Azure Functions

Azure Functions is an integration and automation service for solving integration problems and automating business processes. They can define inputs, actions, conditions, and outputs.

Hosting options

- **Consumption plan:** *default*; only pay when functions are running;
- **Premium plan:** more powerful; pre-warmed workers; connects to virtual networks;
- **Dedicated plan:** runs functions within an App Service Plan at regular plan rates;
- **ASE:** App Service Environment; dedicated and fully isolated; securely runs apps at high scale;
- **Kubernetes:** dedicated and fully isolated on top of the Kubernetes platform

The *FunctionTimeout* property in the host.json project file specifies the timeout duration. After the trigger starts function execution, the function needs to return/respond within the timeout duration. On any plan, a function app requires a general Azure Storage account, which supports Azure Blob, Queue, Files, and Table storage. Function code files are stored on Azure Files shares on the function's main storage account. When you delete the main storage account of the function app, the function code files are deleted and can't be recovered.

Compare Functions and Logic Apps

Both Functions and Logic Apps are Azure Services that enable serverless workloads. Azure Functions is a serverless compute service, whereas Azure Logic Apps is a serverless workflow integration platform. Both can create complex *orchestrations*, a set of steps, or actions, to accomplish a complex task.

	Azure Functions	Logic Apps
Development	Code-first (imperative)	Designer-first (declarative)
Connectivity	About a dozen built-in binding types, write code for custom bindings	Large collection of connectors, Enterprise Integration Pack for B2B scenarios, build custom connectors
Actions	Each activity is an Azure function; write code for activity functions	Large collection of ready-made actions
Monitoring	Azure Application Insights	Azure portal, Azure Monitor logs
Management	REST API, Visual Studio	Azure portal, REST API, PowerShell, Visual Studio
Execution context	Runs in Azure, or locally	Runs in Azure, locally, or on premises

Compare Functions and WebJobs

Both are built on Azure App Service and support features such as source control integration, authentication, and monitoring with Application Insights integration. Azure Functions offers more developer productivity than Azure App Service WebJobs does. It also offers more options for programming languages, development environments, Azure service integration, and pricing. For most scenarios, it's the best choice.

Scaling

In the Consumption and Premium plans, Azure scales CPU and memory by adding more instances of the function app – all functions within an instance scale at the same time. The number of instances is determined by the number of events that trigger a function.

Consumption plan

Each instance of the function app is limited to 1.5 GB of memory and one CPU. Function apps that share the same Consumption plan scale independently. Scales automatically based on number of incoming events, but isn't predictive.

Premium plan

The plan size determines the available memory and CPU for all apps in an instance. Scales automatically based on number of incoming events, but isn't predictive.

Dedicated plan

Manual autoscaling based on predictive usage.

ASE

Manual autoscaling based on predictive usage.

Kubernetes

Event driven scaling based on KEDA (Kubernetes event driven auto-scaler).

Scale Controller

A component called the scale controller is used to monitor the rate of events and determine whether to scale out or scale in based on heuristics for each trigger type.
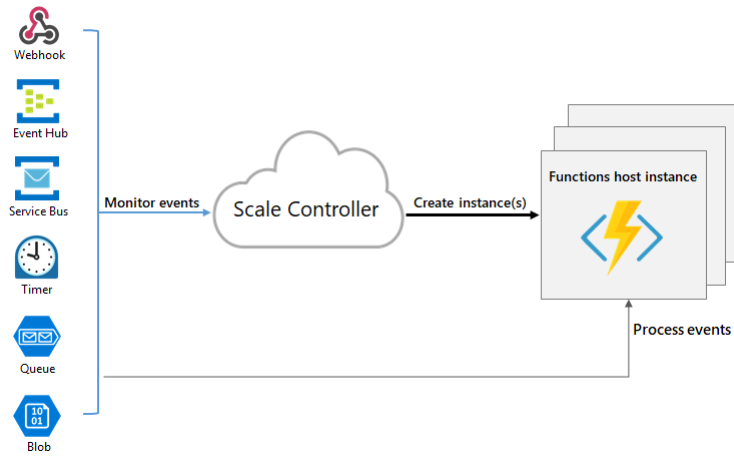
Other Scaling behaviors

- **Maximum instances**: A single function app only scales out to a maximum of 200 instances. A single instance may process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions. By default, Consumption plan functions scale out to as many as 200 instances, and Premium plan functions scales out to as many as 100 instances. You can specify a lower maximum for a specific app by modifying the *functionAppScaleLimit* value.
- **New instance rate**: For HTTP triggers, new instances are allocated, at most, once per second. For non-HTTP triggers, new instances are allocated, at most, once every 30 seconds.

A10-2 Develop Azure Functions

https://learn.microsoft.com/en-us/training/modules/develop-azure-functions/

A function contains two important pieces - your code, and some config, the function.json file. The *function.json* file defines the function's trigger, bindings, and other configuration settings. The bindings property is where you configure both triggers and bindings. Every binding requires the following settings:

- **type**: Name of binding. For example, queueTrigger.
- **direction**: Receiving data into the function or sending data from the function.
- **name**: The name that is used for the bound data in the function. For example, myQueue.

A *function app* provides an execution context in which your functions run. It's the unit of deployment and management for your functions, so that your functions can be managed, deployed, and scaled together.

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file, host.json, that contains runtime-specific configurations. A bin folder contains packages and other library files that the function app requires. Specific folder structures required by the function app depend on language.

Example function.json file:

```
{
  "disabled":false,
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "order",
      "queueName": "myqueue-items",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "type": "table",
      "direction": "out",
      "name": "$return",
      "tableName": "outTable",
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING"
    }
  ]
}
```

In this example, for the input binding, the *name* property identifies a function parameter, in this case the function is required to have a parameter named "order". For the output binding, the name parameter specifies how the function provides the output value, in this case by using the function return value.

JavaScript example to implement the above function.json:

```
module.exports = async function (context, order) {
    order.PartitionKey = "Orders";
    order.RowKey = Math.random().toString();
```

```
        context.bindings.order = order;
    };
```

C# script example:

```
    public static Person Run(JObject order, ILogger log)
    {
        return new Person() {
                PartitionKey = "Orders",
                RowKey = Guid.NewGuid().ToString(),
                Lastname = order["Lastname"].ToString(),
    }
    public class Person
    {
        public string PartitionKey { get; set; }
        public string RowKey { get; set; }
        public string Lastname { get; set; }
    }
```

In a C# class library, the same trigger and binding information is provided by attributes instead of a function.json file:

```
    public static class QueueToTableExample
    {
        [FunctionName("QueueToTableExample ")]
        [return: Table("outTable", Connection = "MY_TABLE_STORAGE_ACCT_APP_SETTING")]
        public static Person Run(
            [QueueTrigger("myqueue-items", Connection = "MY_STORAGE_ACCT_APP_SETTING")]JObject order,
            ILogger log)
        {
            return new Person() {
                    PartitionKey = "Orders",
                    RowKey = Guid.NewGuid().ToString(),
                    Lastname = order["Lastname"].ToString(),
        }
    }
    public class Person
    {
        public string PartitionKey { get; set; }
        public string RowKey { get; set; }
        public string Lastname { get; set; }
    }
```

Triggers and bindings

Triggers and *bindings* let you avoid hardcoding access to other services. Triggers have associated data, which is often provided as the payload of the function. Data from bindings is provided to the function as parameters.

In compiled languages such as C# class libraries and Java, triggers and bindings are defined by decorating methods and parameters with attributes or annotations. The parameter type defines the data type for input data. Use a custom type to de-serialize to an object. Binding directions are inferred from the parameter type.

In scripting languages such as C# script and JavaScript, triggers and bindings are defined by updating *function.json* and can be edited in the portal. For their data types, use the dataType property in the function.json file:

```
    {
        "dataType": "binary",
        "type": "httpTrigger",
        "name": "req",
        "direction": "in"
    }
```

Connection details are referenced by names from the project's configuration provider allowing them to be changed across environments. For example, in *function.json* you would set a property called `connection` to the name of an environment variable that contains the connection string. The default configuration provider uses environment variables that are set in Application Settings when running in Azure, or from the local settings file when developing locally.

Some connections in Azure Functions are configured to use an identity instead of a secret. Identity-based connections use a managed identity. When run in other contexts, such as local development, your developer identity is used instead. Whatever identity is being used must have permissions to perform the intended actions. This is typically done by assigning a role in Azure RBAC, or specifying the identity in an access policy.

A10-3 Implement Azure Key Vault

Azure Key Vault is used to manage secrtets, keys, certificates. A vault is logical group of secrets.

- **Secrets**: anything that you want to tightly control access to such as tokens, passwords, certificates, API keys, and other secrets.
- **Keys**: encryption keys.
- **Certificates**: SSL/TLS certificates for use with Azure and your internal connected resources.

Supports two types of containers: vaults and HSM pools. HSM=Hardware Security Module

vaults		HSM pools
Software backed	HSM backed	HSM backed
keys, secrets, certificates	keys, secrets, certificates	keys only

Tiers

- **Standard**: encrypts with a software key
- **Premium**: encrypts with a software key, or HSM-protected keys

Benefits
Centralized application secrets, securely accessed using URIs. URIs allow the applications to retrieve specific versions of a secret.

Securely store secrets and keys with authentication and authorization. Authentication is done via *Microsoft Entra ID*. Authorization is done through *Azure RBAC* when dealing with the management of the vaults, and through *key vault access policy* when attempting to access data stored in a vault.

Monitor access and use by enabling logging for your vaults. Logs can be archived to a storage account, streamed to an event hub, or sent to Azure Monitor logs.

Simplified administration by securing secrets, following a life cycle for the secrets, and making secrets highly available. This is accomplished by:

- Removing the need for in-house knowledge of HSM.
- Scaling up on short notice to meet usage spikes.
- Replicating contents to a secondary region, for failover and high availability.
- Standardize administration via the portal, Azure CLI and PowerShell
- Automating tasks on certificates from Public CAs such as enrollment and renewal.

Authentication

1. **Managed identities**: <u>Recommended approach</u>. You can assign identities to Azure resources such as VM's, apps, etc. Azure then manages authentication behind the scenes, automatically authenticating the application with other Azure services.
2. **Service principal and certificate**: <u>Not recommended</u> because it's hard to rotate the certificate.
3. **Service principle and secret**: <u>Not recommended</u> because it's hard to rotate the secret.

Authenticating without managed identity

If you can't use managed identity, you instead register the application with your Microsoft Entra tenant, creating a second application object that identifies the app across all tenants.

Authentication to Key Vault in application code is done with the Key Vault SDK using the Azure Identity client library.

When authenticating to Key Vault with REST, access tokens must be sent to the service using the HTTP Authorization header:

```
PUT /keys/MYKEY?api-version=<api_version>  HTTP/1.1
Authorization: Bearer <access_token>
```

On failure an HTTP 401 error is returned to the client and will include the WWW-Authenticate header, for example:

```
401 Not Authorized
WWW-Authenticate: Bearer authorization="…", resource="…"
```

where:

- authorization is the address of the OAuth2 authorization service that may be used to obtain an access token for the request
- resource is the name of the to use in the authorization request

Encryption of data in transit

Data travelling between Key Vault and clients is protected by TLS. Connections between clients and Microsoft cloud services are also protected by PFS (Perfect Forward Security) and RSA-based 2,048-bit encryption.

Best Practices

- Use separate key vaults per application per environment.
- Secure access to your key vaults by allowing only authorized applications and users and by following the principle of least privilege.
- Create regular back ups of your vault.
- Turn on logging and alerts.
- Turn on soft-delete and purge protection to guard against accidental or force deletion of the secret.

A10-4 Implement Azure App Configuration

Azure App Configuration is a service to centrally manage application settings and feature flags.

Key-value pairs
Keys serve as the name for key-value pairs. Keys stored in App Configuration are case-sensitive, unicode-based strings. The keys app1 and App1 are distinct. Use any unicode character in key names entered into App Configuration except for *, ,, and \. There's a combined size limit of 10,000 characters on a key-value pair. You can organize keys into a hierarchical namespace by using a character delimiter, such as / or : for example

```
AppName:Service1:ApiEndpoint
AppName:Region1:DbEndpoint
```

Key values can optionally have a label attribute to differentiate key values with the same key. Common uses of labels are to specify multiple environments, and managing versioning. By default, the label is empty, or null.

Values can use all unicode characters. There's an optional user-defined content type attribute associated with each value.

Configuration data stored in an App Configuration store is encrypted at rest and in transit.

Features Flags
Feature management decouples feature release from code deployment enabling quick changes to feature availability. A *feature flag* is a variable with a state of on or off and an associated code block. The state of the flag triggers whether the code block runs. A *feature manager* is an application package that handles the lifecycle of the feature flags in an application. A *filter* is a rule for evaluating the state of a feature flag, and can be based on things like a user group, a device or browser type, a geographic location, a time window, etc.

Feature flags have two components:

- An application that makes use of feature flags.
- A separate repository that stores the feature flags and their current states.

```
if (featureFlag) {
    // This following code will run if the featureFlag value is true
} else {
    // This following code will run if the featureFlag value is false
}
```

Feature flag declarations have a name and a list of one or more filters.

```
"FeatureManagement": {
    "FeatureA": true, // Feature flag set to on
    "FeatureB": false, // Feature flag set to off
    "FeatureC": {
        "EnabledFor": [
            {
                "Name": "Percentage",
                "Parameters": {
                    "Value": 50
                }
            }
        ]
    }
}
```

You should externalize all the feature flags used in an application allowing you to change feature flag states without modifying the application itself. Azure App Configuration is a centralized repository for feature flags. You use the App Configuration libraries for various programming language frameworks to easily access these feature flags from your application.

Secure app configuration

- Customer-managed keys
- Private endpoints
- Managed identities

Customer managed keys

The following components are required to successfully enable the customer-managed key capability

- Standard tier Azure App Configuration instance
- Azure Key Vault with soft-delete and purge-protection features enabled
- An RSA or RSA-HSM key within the Key Vault

To allow Azure App Configuration to use the Key Vault key:

1. Assign a managed identity to the Azure App Configuration instance
2. Grant the identity GET, WRAP, and UNWRAP permissions in the target Key Vault's access policy.

Private endpoints

Private endpoints for Azure App Configuration allow clients on a VNet to securely access data over a private link. The private endpoint uses an IP address from the VNet address space for your App Configuration store.

Managed identities

A managed identity from Microsoft Entra ID allows Azure App Configuration to access other Microsoft Entra ID-protected resources, such as Azure Key Vault.

Two types of managed identities:

- A system-assigned identity is deleted if your configuration store is deleted. A configuration store can only have one system-assigned identity.
- A user-assigned identity is a standalone Azure resource that can be assigned to your configuration store. A configuration store can have multiple user-assigned identities.

To add a system-assigned managed identity to an existing configuration store:

```
az appconfig identity assign \
    --name myTestAppConfigStore \
    --resource-group myResourceGroup
```

To add a user-assigned managed identity to an existing configuration store, first create the identity:

```
az identity create --resource-group myResourceGroup --name myUserAssignedIdentity
```

and then assign it to the configuration store:

```
az appconfig identity assign --name myTestAppConfigStore \
    --resource-group myResourceGroup \
    --identities \
/subscriptions/[subscription
id]/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myUserAssignedIden
tity
```

A10-5 Monitor app Performance

Application Insights

Azure Monitor Application Insights provides APM (Application Performance Monitoring) by collecting events such as metrics and application Telemetry data, which describe application activities and health, as well as trace logging data.

Application Insights monitors:

- **Request rates**, response times, and failure rates. Which pages are most popular?
- **Dependency rates**, response times, and failure rates. Find out whether external services are slowing you down?
- **Exceptions** – analyze aggregated statistics, or specific instance stack traces and related requests.
- **Page views** and load performance from users' browsers.
- **AJAX calls** – rates, response times, and failure rates.
- **User and session counts** – statistics.
- **Performance counters** such as CPU, memory, and network usage from the server.
- **Host diagnostics** from Docker or Azure.
- **Diagnostic trace logs** from your app – correlate trace events with requests.
- **Custom events and metrics** that you write yourself in the client or server code

Metrics

Observe activity from your deployed application in real time with no effect on the host environment.

Log-based metrics

Developers can send events manually by writing code, or they can rely on the automatic collection of events from auto-instrumentation.

For situations when the volume of events is too high, Application Insights implements telemetry volume reduction techniques. Unfortunately, lowering the number of stored events also lowers the accuracy of the metrics.

Pre-aggregated metrics

The pre-aggregated metrics aren't stored as individual events, but rather as pre-aggregated time series enabling near real-time alerting. The newer SDKs pre-aggregate metrics during collection, resulting in less data ingestion and lower cost, and accuracy isn't affected by sampling or filtering.

Instrument app for monitoring

Application Insights is enabled either through Auto-Instrumentation agents, or by adding the Application Insights SDK or the OpenCensus SDK, to your application code.

Auto-instrumentation is the preferred instrumentation method. It requires no developer investment. It's also the only way to instrument an application in which you don't have access to the source code.

You only need to use an SDK when you want additional control of the flow of telemetry, custom events, or when auto-instrumentation isn't available. To use an SDK, you install a small instrumentation package in your app. The app and its components don't have to be hosted in Azure.

Availability tests

Probe your applications external endpoint(s) to test the overall availability and responsiveness over time. You can set up availability tests for any HTTP or HTTPS endpoint that's accessible from the public internet. You don't have to make any changes to the website, and it doesn't even have to be a site that you own, so you can test other sites that your site depends on.

There are three types of availability tests:

- **URL ping test** to validate whether an endpoint is responding and measure performance associated with that response;
- **Standard test**, similar to ping test, and also includes SSL certificate check, HTTP request verb, common headers, and other custom request data.
- **Custom TrackAvailability test** *(previously called multi-step test)* can be run using the TrackAvailability() method.

Application Map

A high-level top-down view of the application architecture, component health and responsiveness. It helps you spot problems across all components of your distributed application. Components are independently deployable parts of your application. In addition to components, you can view external dependencies such as SQL, Event Hubs, etc.

The app map finds components by following HTTP dependency calls made between servers with the Application Insights SDK installed. When you first load the application map, a set of queries is triggered to discover the components related to this component. If all of the components are within a single Application Insights resource, then this discovery step isn't required. One of the key objectives with this experience is to be able to visualize complex topologies with hundreds of components. Click on any component to see related insights and go to the performance and failure triage experience for that component.

Here is an example of an application map:
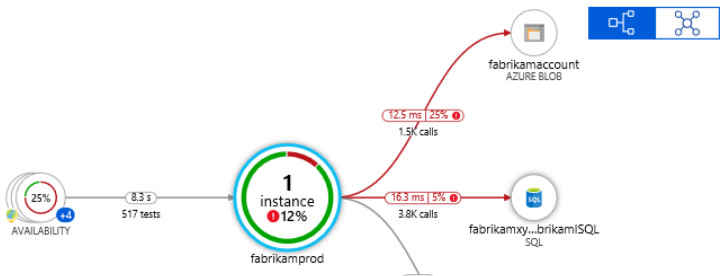
fabrikamprod

Filter by this node

View in Analytics

TOP FAILING REQUESTS BY NAME

NAME	COUNT
GET ServiceTickets/Details	600
GET Employees/Details	200
POST ServiceTickets/Create	195

Investigate failures

SLOWEST REQUESTS BY NAME

NAME	DURATION (AV...
POST ServiceTickets/Create	195.9 ms
GET Home/Index	123.3 ms
GET ServiceTickets/Index	106.9 ms

Investigate performance

A10-6 Implement Azure Container Apps

https://learn.microsoft.com/en-us/training/modules/implement-azure-container-apps/

The top-level resource used by Azure Container Apps for grouping containers is the Environment. The hierarchy is as follows:

- Environment
  - Container Apps
    - Revisions
      - Replicas
        - Containers

Containers for an Azure Container App are grouped together in pods inside revision snapshots.
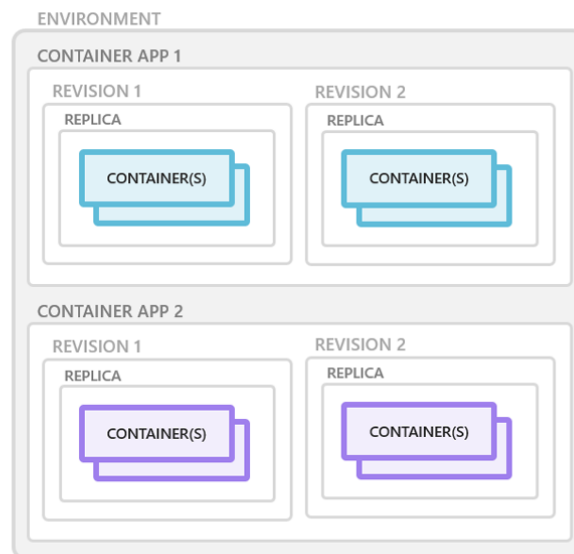
Azure Container Apps enables you to run microservices and containerized applications on a serverless platform that runs on top of Azure Kubernetes Service.

To create a container app:

1. Create resource group

```
az group create \
    --name $myRG \
    --location $myLocation
```

2. Create environment

```
az containerapp env create \
    --name $myAppContEnv \
    --resource-group $myRG \
    --location $myLocation
```

3. create the container app

```
az containerapp create \
    --name my-container-app \
    --resource-group $myRG \
    --environment $myAppContEnv \
    --image mcr.microsoft.com/azuredocs/containerapps-helloworld:latest \
    --target-port 80 \
    --ingress 'external' \
    --query properties.configuration.ingress.fqdn
```

Azure Container Apps manage Kubernetes and container orchestration for you.

Multiple containers

You can define multiple containers in a single container app to implement the sidecar pattern. They share hard disk and network resources and experience the same application lifecycle. To run multiple containers in a container app, add more than one container in the containers array of the container app template.

Container registries

You can deploy images hosted on private registries by providing credentials in the Container Apps configuration.

```
{
  ...
  "registries": [{
    "server": "docker.io",
    "username": "my-registry-user-name",
    "passwordSecretRef": "my-password-secret-name"
  }]
}
```

Limitations

- **Privileged containers**: Azure Container Apps can't run a process in a container that requires root access, the container experiences a runtime error.
- **Operating system**: Linux/amd64 container images.

Azure Container Apps provides built-in authentication and authorization, providing out-of-the-box authentication with federated identity providers. Federated identity are third-party identity providers that manage the user identities and authentication flow for you. This feature should only be used with HTTPS.

Configure your container app for authentication from your container app's ingress configuration:

- Ensure *allowInsecure* is **disabled**.
- set *Restrict access* setting to **Require authentication** to restrict access to authenticated users.
- set *Restrict access* setting to **Allow unauthenticated** to not restrict access.

The following identity providers are available:

Provider	Sign-in endpoint
Microsoft Identity Platform	/.auth/login/aad
Facebook	/.auth/login/facebook
GitHub	/.auth/login/github
Google	/.auth/login/google
Twitter	/.auth/login/twitter
Any OpenID Connect provider	/.auth/login/<providerName>

The authentication and authorization middleware component runs as a sidecar container in your application. When enabled, every incoming HTTP request passes through the security layer before being handled by your application. The middleware handles:

- Authenticating users and clients with an identity provider
- Managing the authenticated session
- Injecting identity information into HTTP request headers

The module runs in a separate container, isolated from your application code, so the security container doesn't run in-process, and no direct integration with specific language frameworks is possible.
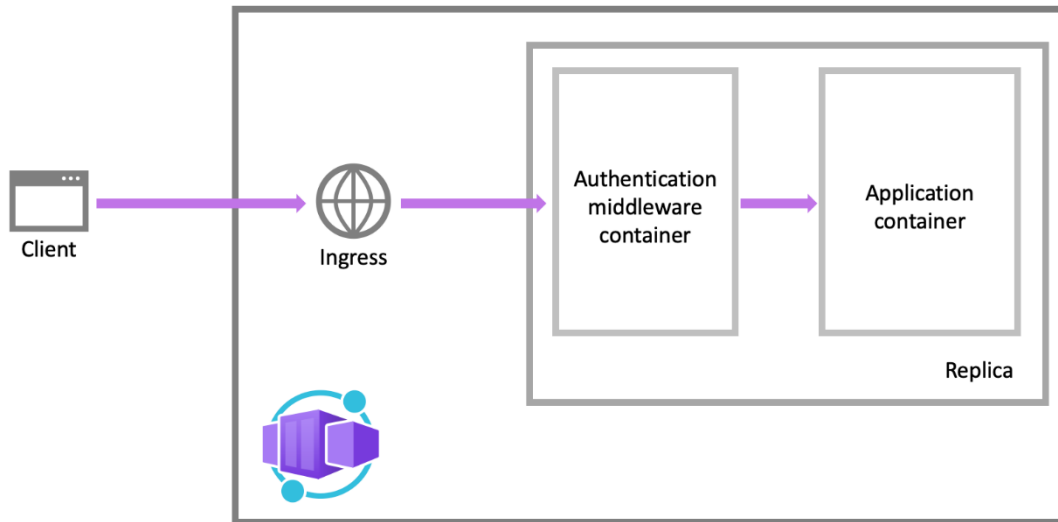
Authentication flow

The authentication flow depends on whether the container app uses the provider SDK.

- With provider SDK: (client flow): The application signs users in, with SDK.
  For example: browser app.
- Without the SDK: (server flow): The application delegates sign-in to the provider.
  For example: mobile app.

Revisions

Azure Container Apps implements versioning by creating revisions. A revision is an immutable snapshot of a container app version. New revisions are created when you update your application with revision-scope changes. A revision-scope change is any change to the parameters in the properties.template section of the container app resource template. You can control which revisions are active, and the external traffic that is routed to each active revision.

You can set the revision suffix in the ARM template, through the Azure CLI `az containerapp create` and `az containerapp update` commands, or when creating a revision via the Azure portal. With the `az containerapp update` command you can modify environment variables, compute resources, scale parameters, and deploy a different image. If your container app update includes revision-scope changes, a new revision is generated. You can list all revisions associated with your container app with `the az containerapp revision list` command.

Secrets

Container Apps doesn't support Azure Key Vault integration. Instead, enable managed identity in the container app and use the Key Vault SDK in your app to access secrets. Once secrets are defined at the application level, secured values are available to container apps.

- Secrets are scoped to an application, outside of any specific revision.
- Adding, removing, or changing secrets doesn't generate new revisions.
- Each application revision can reference one or more secrets.
- Multiple revisions can reference the same secret(s).

When a secret is updated or deleted, you can respond in one of two ways:

- Deploy a new revision.
- Restart an existing revision.

Before you delete a secret, deploy a new revision that no longer references the old secret. Then deactivate all revisions that reference the secret.

Defining secrets

When you create a container app, secrets are defined using the `--secrets` parameter. For example:

```
az containerapp create \
  --resource-group "my-resource-group" \
  --name queuereader \
  --environment "my-environment-name" \
  --image demos/queuereader:v1 \
  --secrets "queue-connection-string=$CONNECTION_STRING"
```

In this example, the local computer's CONNECTION_STRING environment variable is used to create a secret called queue-connection-string.

You can also set environment variables within the container to the value of a secret. Set its value to `secretref:`, followed by the name of the secret.

```
az containerapp create \
  --resource-group "my-resource-group" \
  --name myQueueApp \
  --environment "my-environment-name" \
  --image demos/myQueueApp:v1 \
  --secrets "queue-connection-string=$CONNECTIONSTRING" \
  --env-vars "QueueName=myqueue" "ConnectionString=secretref:queue-connection-string"
```

Dapr

The Distributed Application Runtime (Dapr) is a set of incrementally adoptable features that simplify the authoring of distributed, microservice-based applications. Dapr enables reliable and secure application intercommunication through messaging via pub/sub or service-to-service calls.

Dapr APIs:

Service-to-service invocation	State management	Publish and subscribe	Bindings (input/output)	Actors	Observability	Secrets
Perform direct, secure, service-to-service method calls	Create long running, stateless and stateful services	Secure, scalable messaging between services	Trigger code through events from inputs Input and output bindings to external resources including databases and queues	Encapsulate code and data in reusable actor objects as a common microservices design pattern	See and measure the message calls across components and networked services	Securely access secrets from your application

Azure Container Apps provides:

- managed and supported Dapr integration
- Dapr version upgrades seamlessly
- simplified Dapr interaction model to increase developer productivity

The following Dapr example is based on the Pub/sub API:
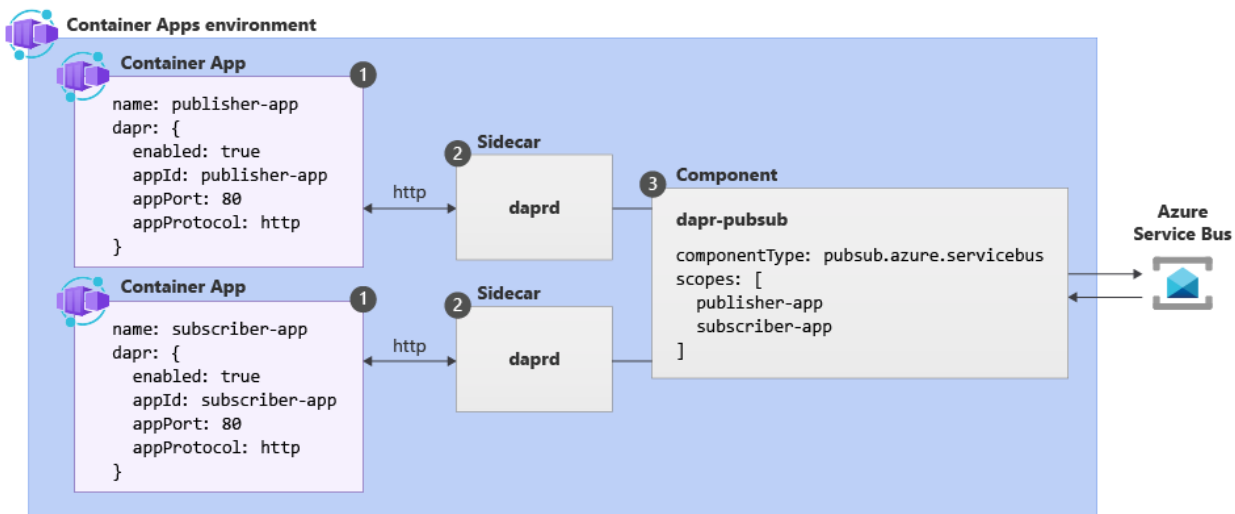
Azure Container Apps provides three channels to configure Dapr:

- Container Apps CLI
- Infrastructure as Code (IaC), Bicep or ARM templates
- The Azure portal

By default, all Dapr-enabled container apps within the same environment load the full set of deployed components. This can be overridden by using application scopes.

Terms:

KEDA: Kubernetes Event-driven Autoscaler

Blue/Green deployment: is an application release model that gradually transfers user traffic from a previous version of an app or microservice to a nearly identical new release—both of which are running in production. – https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment

A/B testing: at its most basic, is a way to compare two versions of something to figure out which performs better. – https://hbr.org/2017/06/a-refresher-on-ab-testing

Service discovery: the process of automatically detecting devices and services on a computer network. It aims to reduce the manual configuration effort required from users and administrators. A service discovery protocol (SDP) is a network protocol that helps accomplish service discovery. – https://en.wikipedia.org/wiki/Service_discovery

A10-7 Manage container images in Azure Container Registry

ACR groups images by repositories.

An image is a read-only snapshot of a Docker-compatible container. ACR can include both Windows and Linux images.

Use the Azure Container Registry (ACR) service with your existing container development and deployment pipelines, or use Azure Container Registry Tasks to build container images in Azure.

High numbers of repositories and tags can impact ACR performance. Periodically delete unused registry resources as part of your registry maintenance routine. Deleted resources can't be recovered.

Security and data protection
ACR uses encryption-at-rest for image data security. In Premium tier, ACR uses geo-redundancy for image data protection. ACR stores data in the region where the registry is created, to help meet data residency compliance requirements.

In Premium service tier, zone redundancy uses Azure availability zones to replicate your registry to a minimum of three separate zones within a region. ACR may also store registry data in a paired region in the same geography (geo-replication), in all regions except Brazil South and Southeast Asia where registry data is confined to the same region.

Tiers
Basic: cost-optimized entry point for developers. Basic has the same programmatic capabilities as Standard and Premium, but with storage and image throughput appropriate for lower usage scenarios.

Standard: same capabilities as Basic, with increased storage and throughput. Should satisfy the needs of most production scenarios.

Premium: highest amount of included storage and throughput, enabling high-volume scenarios. Adds geo-replication for managing a single registry across multiple regions, content trust for image tag signing, and private link with private endpoints to restrict access to the registry.

ACR Tasks
ACR Tasks provide cloud-based container image build automation. By default, ACR Tasks builds images for the Linux OS and the amd64 architecture. Specify the --platform tag to build Windows images or Linux images for other architectures, using the OS/architecture/variant format (for example, --platform Linux/arm64/v8).

Quick tasks
Build and push a container image to ACR in Azure, without needing a local Docker Engine. Think docker build, docker push in the cloud. Using the familiar `docker build` format, the `az acr build` CLI command sends a set of files to build to ACR Tasks and pushes the built image to its registry upon completion.

Automatically triggered tasks

Tasks can be triggered by source code updates, updates to a container's base image, or timers.

- **Source code update**: Trigger an ACR Tasks-created webhook, on an update to a repository in GitHub or Azure DevOps.
- **Base image update**: either in your registry or in a public repo, ACR Tasks can automatically build any application images based on it.
- **Schedule**: schedule a task by setting up one or more timer triggers.

Multi-step tasks

Multi-step tasks, defined in a YAML file specify individual build and push operations for container images, with each step using the container as its execution environment.

Dockerfile

A Dockerfile is a script to build a Docker image. For example:

```
# Base image
FROM mcr.microsoft.com/dotnet/runtime:6.0

# Set working directory
WORKDIR /app

# Copy the previously published app to the container's /app directory
COPY bin/publish/ .

# Expose port 80 on the container
EXPOSE 80

# Command to run when the container starts
CMD ["dotnet", "MyApp.dll"]
```

Each of these steps creates a cached container image. These temporary images are layered and presented as single image once all steps complete.

Build and run image using Azure CLI

This example shows how to Build and run a container image by using ACR Tasks.

```
# create a resource group
az group create --name az204-acr-rg --location <myLocation>

# create a basic container registry
az acr create --resource-group az204-acr-rg \
    --name <myContainerRegistry> --sku Basic

# build the image, tagged as "v1", and pushes it to your registry
az acr build --image sample/hello-world:v1 --registry <myContainerRegistry> --file Dockerfile .
```

To verify the results, list the repositories in your registry:

```
az acr repository list --name <myContainerRegistry> --output table
```

and list the tags:

```
az acr repository show-tags --name <myContainerRegistry> \
    --repository sample/hello-world --output table
```

A10-8 Run container images in Azure Container Instances

Azure Container Instances (ACI) offers the fastest and simplest way to run a container in Azure. ACI is for any scenario that can operate in isolated containers.

Benefits
- Fast start
- Easy access – expose container groups directly to the internet with an IP address and a FQDN
- Secure – Hypervisor-level security
- Confidentiality – minimum customer data is stored
- Custom sizes
- Persistent storage – mount Azure Files shares
- Cross platform – load Linux and Windows containers with same API

Multi-container Groups
The top-level resource in Azure Container Instances is the container group, a collection of containers on the same host machine that share a lifecycle, network and storage resources. It's similar to a Kubernetes pod. For example:
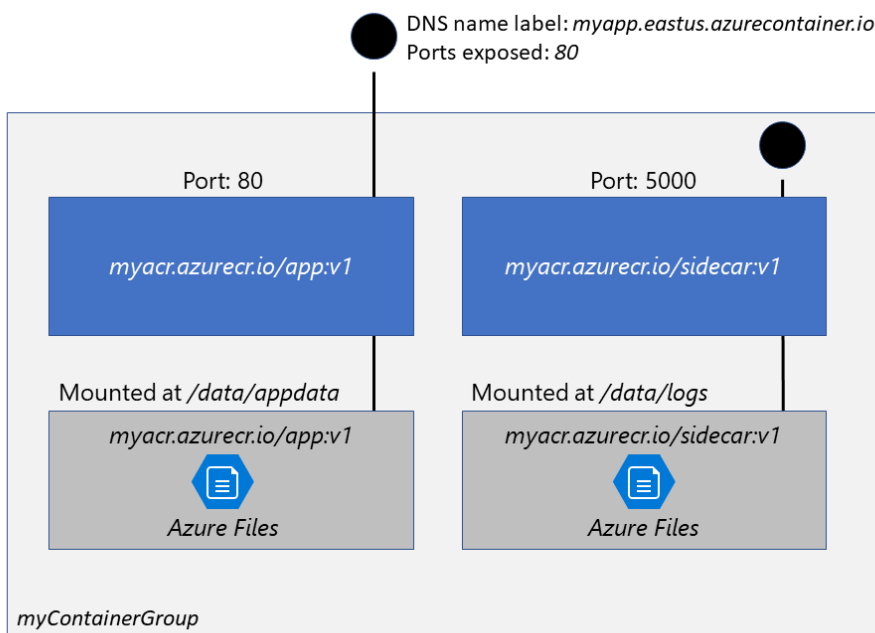
Deployment
Multi-container groups can only be deployed using ARM templates or YAML. Single containers can be deployed using the Azure CLI as well as ARM templates or YAML.

YAML: more concise. Recommended when your deployment includes only container instances.

ARM Template: when you need to deploy more Azure service resources.

YAML template is the preferred method when deploying container groups consisting of multiple containers.

Resources are requested for the container group and divided equally among the containers.

Container groups share an IP address, and a port which must be exposed on the IP address and from the container, to enable access from external clients.

External storage volumes can be mounted for:

- Azure file share
- Secret
- Empty directory
- Cloned git repo

Single container example

A single container can be deployed using the `AZ container create` CLI command.

```
# create a resource group
az group create --name az204-aci-rg --location <myLocation>
# create and start a container
az container create --resource-group az204-aci-rg \
    --name mycontainer \
    --image mcr.microsoft.com/azuredocs/aci-helloworld \
    --ports 80 \
    --dns-name-label $DNS_NAME_LABEL --location <myLocation>
```

You can verify the container is running with the `az container show` CLI command to get the FDQN, and then test it by navigating to the FDQN from a browser.

```
az container show --resource-group az204-aci-rg \
    --name mycontainer \
    --query "{FQDN:ipAddress.fqdn,ProvisioningState:provisioningState}" \
    --out table
```

Restart policy

Container instances are billed by the second for the compute resources used while the container is executing. Containers restart by default when their processes complete. You can override by setting a container restart policy to one of the following

Always: This is the default setting applied when no restart policy is specified at container creation.

Never: The containers run at most once.

OnFailure: restarted only when the process terminates with a nonzero exit code. The containers are run at least once.

When creating a single container app, you can specify the restart policy as follows:

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer \
    --image mycontainerimage \
    --restart-policy OnFailure
```

When a container's restart policy is `Never` or `OnFailure`, the container's status is set to `Terminated`.

Environment variables

Environment variables can be provided when you create the container:

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer2 \
    --image mycontainerimage2 \
    --environment-variables 'NumWords'='5' 'MinLength'='8'
```

Set a secure environment variable by specifying the `secureValue` property instead of the regular `value`. Environment variables with secure values aren't visible in your container's properties, they can be accessed only from within the container. For example:

```
apiVersion: 2018-10-01
location: eastus
name: securetest
properties:
  containers:
  - name: mycontainer
    properties:
      environmentVariables:
        - name: 'NOTSECRET'
          value: 'my-exposed-value'
        - name: 'SECRET'
          secureValue: 'my-secret-value'
      image: nginx
      ports: []
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
  osType: Linux
  restartPolicy: Always
tags: null
type: Microsoft.ContainerInstance/containerGroups
```

The YAML file can then be used to deploy the container group with the following CLI command:

```
az container create --resource-group myResourceGroup \
    --file secure-env.yaml
```

File Shares

To persist state beyond the lifetime of an ACI container, you mount a volume from an external store. You can mount an Azure Files file share making it accessible to the container via SMB protocol. Or you can use an Azure file share with Azure virtual machines. To mount an Azure file share as a volume, specify the share and volume mount point when you create the container with `az container create`.

```
az container create \
    --resource-group $ACI_PERS_RESOURCE_GROUP \
```

```
    --name hellofiles \
    --image mcr.microsoft.com/azuredocs/aci-hellofiles \
    --dns-name-label aci-demo \
    --ports 80 \
    --azure-file-volume-account-name $ACI_PERS_STORAGE_ACCOUNT_NAME \
    --azure-file-volume-account-key $STORAGE_KEY \
    --azure-file-volume-share-name $ACI_PERS_SHARE_NAME \
    --azure-file-volume-mount-path /aci/logs/
```

You can only mount Azure Files shares to Linux containers.

To mount multiple volumes in a container instance, you must deploy using an ARM template or a YAML file. There is a volumes array and a volume mounts array.

For example, ARM Template contains a definition of the volumes:

```
"volumes": [{
  "name": "myvolume1",
  "azureFile": {
    "shareName": "share1",
    "storageAccountName": "myStore",
    "storageAccountKey": "<key>"
  }
},
{
  "name": "myvolume2",
  "azureFile": {
    "shareName": "share2",
    "storageAccountName": "myStore",
    "storageAccountKey": "<key>"
  }
}]
```
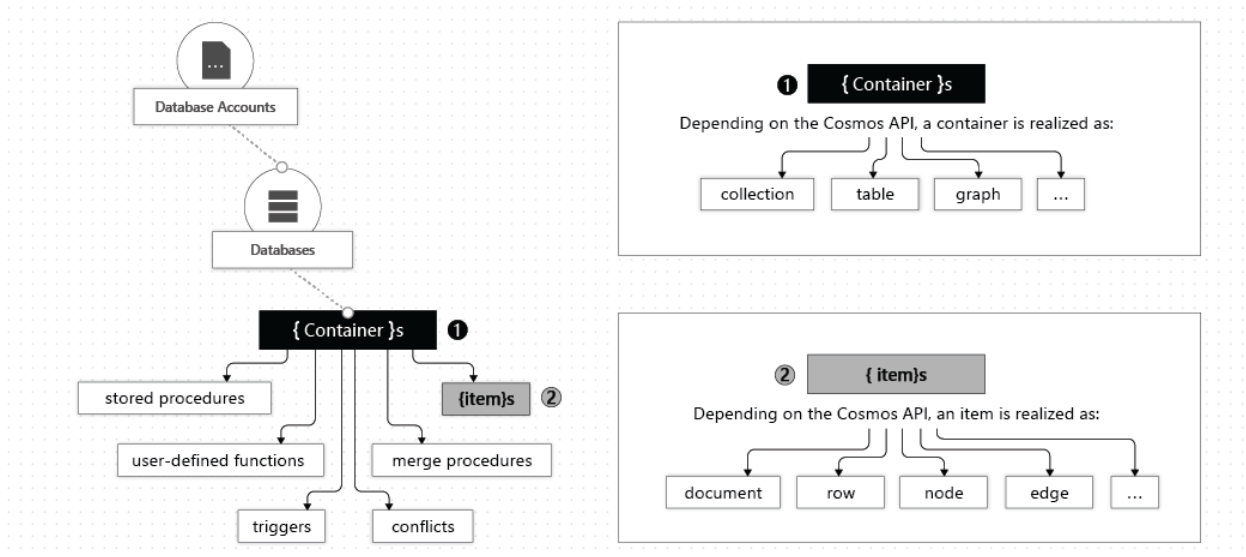
And it contains a definition of how the container access the mount in the properties section of the container definition:

```
"volumeMounts": [{
  "name": "myvolume1",
  "mountPath": "/mnt/share1/"
},
{
  "name": "myvolume2",
  "mountPath": "/mnt/share2/"
}]
```

In this example, the volume mounts connect to the volumes at runtime using the name property, myvolume1 or myvolume2.

A10-9 Work with Azure Cosmos DB

https://learn.microsoft.com/en-us/training/modules/work-with-cosmos-db/

A Cosmos DB account is as follows:

A database is analogous to a namespace serving as the unit of management for a set of containers. A container is a schema-agnostic container of items. Items in the same container can have arbitrary schemas. Depending on which API you use, an item can represent either a document in a collection, a row in a table, or a node or edge in a graph.

Microsoft .NET SDK for Azure Cosmos DB

Note: in the following code, the `DatabaseResponse` object contains information about the operation that resulted in the response object that was returned.

Database example

```
CosmosClient client = new CosmosClient(endpoint, key);

// Create a database (if it doesn't already exist)
DatabaseResponse databaseResponse = await client.CreateDatabaseIfNotExistsAsync(databaseId, 10000);

// Read the database you just created
Database database = client.GetDatabase(database_id);
DatabaseResponse readResponse = await database.ReadAsync();

// Delete the database when it's no longer needed
await database.DeleteAsync();
```

Container example

For this example, the database object was assumed to have previously been obtained as shown in the previous example.

```
// Set throughput to the minimum value of 400 RU/s
ContainerResponse simpleContainer = await database.CreateContainerIfNotExistsAsync(
    id: containerId,
    partitionKeyPath: partitionKey,
    throughput: 400);

// Read a container
Container container = database.GetContainer(containerId);
ContainerProperties containerProperties = await container.ReadContainerAsync();

// Delete a container
await database.GetContainer(containerId).DeleteContainerAsync();
```

Item example

```
// Create item
ItemResponse<SalesOrder> response = await container.CreateItemAsync(
    salesOrder,
    new PartitionKey(salesOrder.AccountNumber));

// Read item
string id = "[id]";
string accountNumber = "[acctnum]";
PartitionKey partitionKey = new PartitionKey(accountNumber);
ItemResponse<SalesOrder> response = await container.ReadItemAsync(
    id,
    partitionKey);
```

The Container.GetItemQueryIterator method creates a query for items under a container in an Azure Cosmos database using a SQL statement with parameterized values. It returns a FeedIterator.

```
QueryDefinition query = new QueryDefinition(
    "select * from sales s where s.AccountNumber = @AccountInput ")
    .WithParameter("@AccountInput", "Account1");

FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
    query,
    requestOptions: new QueryRequestOptions() {PartitionKey = partitionKey});
```

Create Cosmos DB account with CLI

```
# connect to Azure
az login

# create a resource group
az group create --location <myLocation> --name az204-cosmos-rg

# create the cosmos db account
az cosmosdb create --name <myCosmosDBacct> --resource-group az204-cosmos-rg

# retrieve the access key
az cosmosdb keys list --name <myCosmosDBacct> --resource-group az204-cosmos-rg
```

Azure Cosmos DB supports transactional execution of JavaScript for stored procedures, triggers, and user-defined functions (UDFs). To call a stored procedure, trigger, or user-defined function, you need to register it using the SDK. For example in C#:

```
StoredProcedureResponse storedProcedureResponse = await client
    .GetContainer("myDatabase", "myContainer")
    .Scripts
    .CreateStoredProcedureAsync(
        new StoredProcedureProperties
        {
            Id = "spCreateToDoItems",
            Body = File.ReadAllText($@"..\js\spCreateToDoItems.js")
        }
    );
```

Stored procedures

Stored procedures can create, update, read, query, and delete items inside a collection. They are registered per collection, and can operate on any document present in that collection.

```
var helloWorldStoredProc = {
    id: "helloWorld",
    serverScript: function () {
        var context = getContext();
        var response = context.getResponse();

        response.setBody("Hello, World");
    }
}
```

To create an item (document), the stored procedure object has an id and a body property. The body is a function that takes a documentToCreate object, and calls the collection's createDocument() method passing in the documentToCreate and a callback function that is called after the document is added. The callback function receives 2 parameters, an error object in case the operation fails, and a documentCreated object.

```
var createDocumentStoredProc = {
    id: "createMyDocument",
    body: function createMyDocument(documentToCreate) {

        var context = getContext();
        var collection = context.getCollection();

        var accepted = collection.createDocument(
            collection.getSelfLink(),
            documentToCreate,
            function (err, documentCreated) {
                if (err) throw new Error(err.message);
```

```
                context.getResponse().setBody(documentCreated.id)
            });

        if (!accepted) return;
    }
}
```

The stored procedure returns the ID of the item (document) that was created. This is done by the callback function, which sets the ID as the context response body.

Stored procedure input parameters are received as a string even if you pass an array of strings as input. To work around this, you parse the string as an array. For example:

```
function sample(arr) {
    if (typeof arr === "string") arr = JSON.parse(arr);

    arr.forEach(function(a) {
        // do something here
        console.log(a);
    });
}
```

Azure Cosmos DB operations must complete within a limited amount of time. This is called bounded execution. The context collection functions all return a Boolean "accepted" value that tell you if the operation completed.

Stored procedures implement transactions using a continuation-based model, allowing your application to resume the transaction at a new starting point, until the function finishes its entire processing workload.
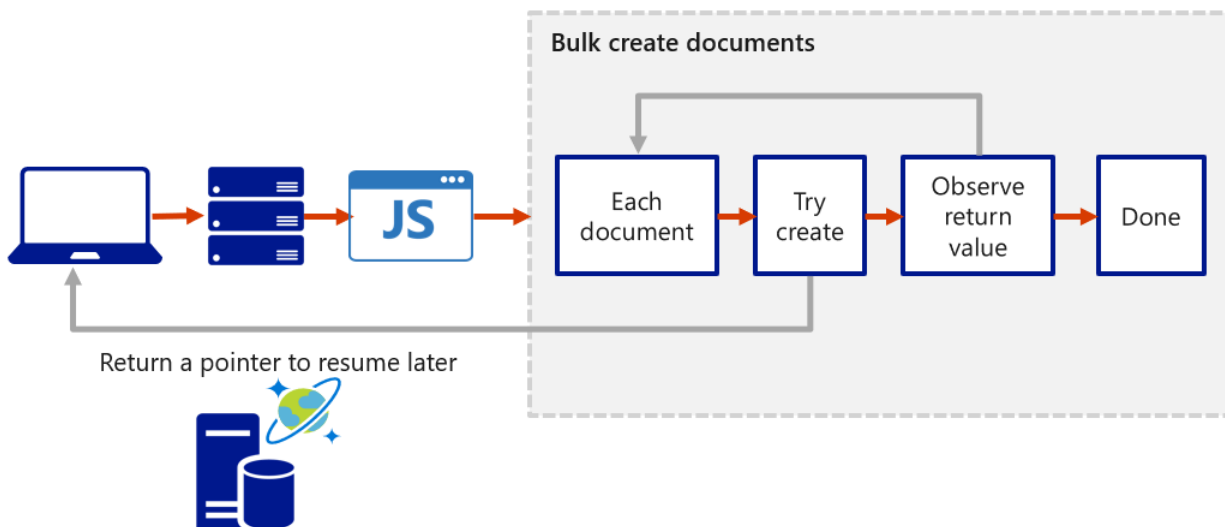


### Triggers and user-defined functions (UDFs)
Azure Cosmos DB supports pre-triggers and post-triggers. Pre-triggers can be used to validate the properties of an item being created. Post triggers can be used to update metadata with details about the newly created item. Triggers don't receive parameters, instead they use the context object by calling the JavaScript getContext() method. The pre-trigger

uses the context to access the request body for the item to be created. The post-trigger uses the context to access the response body for the created item.

User-defined functions are used inside a query. For example, suppose there is an "Income" container with properties:

```
{
    "name": "User One",
    "country": "USA",
    "income": 70000
}
```

Then you can create a UDF as follows:

```
function tax(income) {

    if (income == undefined)
        throw 'no input';

    if (income < 1000)
        return income * 0.1;
    else if (income < 10000)
        return income * 0.2;
    else
        return income * 0.4;
}
```

Explore change feed in Azure Cosmos DB

Change feed is a persistent record of insert and update changes to a container in the order they occur. The sorted list can be processed in order asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing.

Push vs Pull

With the push model, the change feed processor pushes work to a client and the complexity in checking for work and storing state for the last processed work is handled within the change feed processor. With the pull model, in addition to business logic the client has to have logic for storing state for the last processed work, handling load balancing across multiple clients processing work in parallel, and handling errors. Only use the pull model when you need additional low level of control:

- Reading changes from a particular partition key
- Controlling the pace at which your client receives changes
- Doing a one-time read of the existing data (for example, a data migration)

There are two ways you can read from the change feed with a push model: Azure Functions Azure Cosmos DB triggers, and the change feed processor library.

Change feed processor

Four main components:

1. The monitored source container has the data from which the change feed is generated.
2. The lease container provides state storage and coordinates processing across multiple workers.
3. A compute instance hosts the change feed processor. It has a unique identifier called the instance name.
4. The delegate is the code that defines what you want to do with each batch of changes.

Example change feed processor

```
private static async Task<ChangeFeedProcessor> StartChangeFeedProcessorAsync(
    CosmosClient C,
    IConfiguration configuration)
{
    ChangeFeedProcessor changeFeedProcessor =
        C.GetContainer("db-name", "monitored-source-container-name")
        .GetChangeFeedProcessorBuilder<ToDoItem>(
                processorName: "example-descriptive-processor-name",
                onChangesDelegate: myDelegateAsync)
        .WithInstanceName("compute-instance-name")
        .WithLeaseContainer(C.GetContainer("db-name", "lease-container-name"))
        .Build();

    await changeFeedProcessor.StartAsync();
    return changeFeedProcessor;
}
```

Example delegate

```
static async Task myDelegateAsync (
    ChangeFeedProcessorContext context,
    IReadOnlyCollection<ToDoItem> changes,
    CancellationToken cancellationToken)
{
    foreach (ToDoItem item in changes)
    {
        // Simulate some asynchronous operation
        await Task.Delay(10);
    }
}
```

A10-10 Consume an Azure Cosmos DB for NoSQL change feed using the SDK

Change Feed Delegates

In C#, a delegate is a special type of variable or member that references a method with a specific parameter list and return type. The handler passed into the change feed processor can be declared as follows:

```
static async Task HandleChangesAsync(
    IReadOnlyCollection<Product> changes,
    CancellationToken cancellationToken
)
{
    foreach(Product product in changes)
    {
        // Do something with each change
    }
}
```

This can be assigned to a variable:

```
ChangesHandler<Product> changeHandlerDelegate = HandleChangesAsync;
```

and passed to the change feed processor. For example:

```
var myBuilder = myContainer.GetChangeFeedProcessorBuilder<Product>(
            processorName: "example",
            onChangesDelegate: changeHandlerDelegate);
```

Or using a more concise syntax with an anonymous function:

```
ChangesHandler<Product> changeHandlerDelegate = async (
    IReadOnlyCollection<Product> changes,
    CancellationToken cancellationToken
) => {
    foreach(Product product in changes)
    {
        // Do something with each change
    }
};
```

And then passed to the change feed processor as before.

Change Feed Processor

The change feed processor is created as follows:

1. Get the processor builder from the monitored (source) container variable

2. Use the builder to build-out the processor by specifying the delegate, processor name, lease container, and host instance name
3. Start the processor

Simple example:

```
Container sourceContainer = client.GetContainer("myMonitoredSourceContainer", "products");
Container leaseContainer = client.GetContainer("myLeaseContainer", "productslease");

var builder = sourceContainer.GetChangeFeedProcessorBuilder<Product>(
    processorName: "productItemProcessor",
    onChangesDelegate: changeHandlerDelegate
);

ChangeFeedProcessor processor = builder
    .WithInstanceName("desktopApplication")
    .WithLeaseContainer(leaseContainer)
    .Build();

await processor.StartAsync();
// Wait while processor handles items
await processor.StopAsync();
```

Change Feed Estimator

Identifying if your change feed solution needs to scale out requires an estimator. The change feed estimator is a sidecar feature to the processor that measures the number of changes that are pending to be read by the processor at any point in time.

For example:

```
Container sourceContainer = client.GetContainer("myMonitoredSourceContainer", "products");
Container leaseContainer = client.GetContainer("myLeaseContainer", "productslease");

// a processor is created
ChangeFeedProcessor processor = sourceContainer
    .GetChangeFeedProcessorBuilder<Product>(
        processorName: "productItemProcessor",
        onChangesDelegate: changeHandlerDelegate)
    .WithInstanceName("desktopApplication")
    .WithLeaseContainer(leaseContainer)
    .Build();

// implement a delegate to do estimation using the type ChangesEstimationHandler to handle each time
// the estimator polls the change feed to see how many changes have not been processed yet.
ChangesEstimationHandler changeEstimationDelegate = async (
    long estimation,
    CancellationToken cancellationToken
) => {
    // Do something with the estimation
};

// build the estimator in a manner similar to the processor reusing the same lease container
ChangeFeedProcessor estimator = sourceContainer
    .GetChangeFeedEstimatorBuilder(
        processorName: "productItemEstimator",
        estimationDelegate: changeEstimationDelegate)
    .WithLeaseContainer(leaseContainer)
    .Build();

// start everything up
await processor.StartAsync();
```

A10 REFERENCES

- Application-scope changes
  https://learn.microsoft.com/en-us/azure/container-apps/revisions#application-scope-changes
- az containerapp
  https://learn.microsoft.com/en-us/cli/azure/containerapp
- Azure Key Vault availability and redundancy
  https://learn.microsoft.com/en-us/azure/key-vault/general/disaster-recovery-guidance
- Azure Key Vault developer's guide
  https://learn.microsoft.com/en-us/azure/key-vault/general/developers-guide
- Consume an Azure Cosmos DB for NoSQL change feed using the SDK
  https://learn.microsoft.com/en-us/training/modules/consume-azure-cosmos-db-sql-api-change-feed-use-sdk/
- Dapr
  https://docs.dapr.io/concepts/overview/
- Dapr tutorial
  https://learn.microsoft.com/en-us/azure/container-apps/microservices-dapr
- Develop Azure Functions
  https://learn.microsoft.com/en-us/training/modules/develop-azure-functions/
- Docker build reference
  https://docs.docker.com/engine/reference/commandline/build/
- Docker run reference (CLI)
  https://docs.docker.com/engine/reference/run/
- Explore Azure Functions
  https://learn.microsoft.com/en-us/training/modules/explore-azure-functions/
- Implement Azure App Configuration
  https://learn.microsoft.com/en-us/training/modules/implement-azure-app-configuration/
- Implement Azure Container Apps
  https://learn.microsoft.com/en-us/training/modules/implement-azure-container-apps/
- Implement Azure Key Vault
  https://learn.microsoft.com/en-us/training/modules/implement-azure-key-vault/
- Manage container images in Azure Container Registry
  https://learn.microsoft.com/en-us/training/modules/publish-container-image-to-azure-container-registry/
- Microsoft.App containerApps
  https://learn.microsoft.com/en-us/azure/templates/microsoft.app/containerapps?pivots=deployment-language-bicep
- Monitor app performance
  https://learn.microsoft.com/en-us/training/modules/monitor-app-performance/
- Revision-scope changes
  https://learn.microsoft.com/en-us/azure/container-apps/revisions#revision-scope-changes
- Run container images in Azure Container Instances
  https://learn.microsoft.com/en-us/training/modules/create-run-container-images-azure-container-instances/
- Service tier features and limits
  https://learn.microsoft.com/en-us/azure/container-registry/container-registry-skus#service-tier-features-and-limits
- What is Bicep?
  https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview?tabs=bicep
- Work with Azure Cosmos DB
  https://learn.microsoft.com/en-us/training/modules/work-with-cosmos-db/