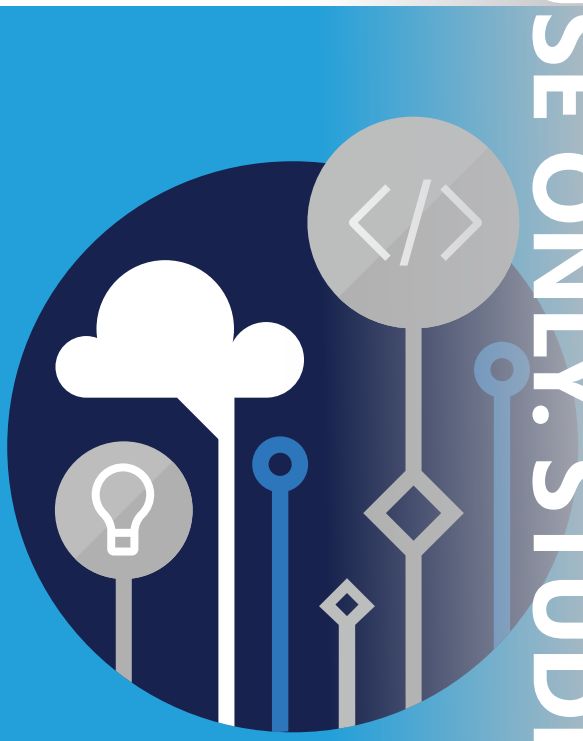
Microsoft

Microsoft Official Course



AZ-300T06

Developing for the Cloud

AZ-300T06
Developing for the Cloud

Contents

Module 0 Start Here

Welcome to Developing for the Cloud

Welcome to Developing for the Cloud

Course Overview: Developing for the Cloud

Welcome to *Developing for the Cloud*. This course is part of a series of five courses to help students prepare for Microsoft's Azure Solutions Architect technical certification exam AZ-300: Microsoft Azure Architect Technologies. These courses are designed for IT professionals and developers with experience and knowledge across various aspects of IT operations, including networking, virtualization, identity, security, business continuity, disaster recovery, data management, budgeting, and governance.

The outline for this course is as follows:

Module 1 - Developing Long-Running Tasks and Distributed Transactions

Topics for this module include:

- Implementing large-scale, parallel, and high-performance apps using batches
- HPC using Microsoft Azure Virtual Machines
- Implementing resilient apps by using queues

As well as, implementing code to address application events by using webhooks. Implementing a web-hook gives an external resource a URL for an application. The external resource then issues an HTTP request to that URL whenever a change is made that requires the application to take an action.

Module 2 - Configuring a Message-Based Integration Architecture

Topics for this module include:

- Configure an app or service to send emails
- Configure an event publish and subscribe model
- Configure the Azure Relay service
- Configuring apps and services with Microsoft Graph

Module 3 - Developing for Asynchronous Processing

Topics for this module include:

- Implement parallelism, multithreading, and processing
- Implement Azure Functions and Azure Logic Apps
- Implement interfaces for storage or data access
- Implement appropriate asynchronous computing models
- Implement autoscaling rules and patterns

This module includes recommendations for implementing code that addresses a transient state. Involving, momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that occur when a service is busy.

Module 4 - Developing for Autoscaling

Topics for this module include:

- Implementing autoscaling rules and patterns
- Implementing code that addresses singleton application instances
- Implementing code that addresses a transient state

Module 5 - Developing Azure Cognitive Services Solutions

Topics for this module include:

- Developing Solutions using Computer Vision
- Developing solutions using Bing Web Search
- Developing solutions using Custom Speech Service
- Developing solutions using QnA Maker

Additionally, you'll receive an overview of Azure IoT Hub service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages.

Module 6 - Develop for Azure Storage

Topics for this module include:

- Develop Solutions that use Azure Cosmos DB Storage
- Develop Solutions that use a Relational Database
- Modeling a Database by using Entity Framework Core
- Develop Solutions that use Microsoft Azure Blob Storage
- Manipulating Blob Container Properties in .NET

What You'll Learn:

- How to configure a message-based integration architecture

- Understand how to Develop for Asynchronous Processing
- Begin creating apps for Autoscaling
- Understand Azure Cognitive Services Solutions

Prerequisites:

Successful Cloud Solutions Architects begin this role with practical experience with operating systems, virtualization, cloud infrastructure, storage structures, billing, and networking.

Module 1 Module Developing Long-Running Tasks and Distributed Transactions

Implement large-scale, parallel, and high-performance apps by using batches

High-performance computing (HPC)

Traditionally, complex processing was something saved for universities and major research firms. A combination of cost, complexity, and accessibility served to keep many from pursuing potential gains for their organizations by processing large and complex simulations or models. Cloud platforms have democratized hardware so much that massive computing tasks are within the reach of hobbyist developers and small and medium-sized enterprises.

High-performance computing (HPC) typically describes the aggregation of complex processes across many different machines, thereby maximizing the computing power of all of the machines. Through HPC in the cloud, one could create enough compute instances to create a model or perform a calculation and then destroy the instances immediately afterward. Advancements in the HPC field have led to improvements in the way that machines can share memory or communicate with each other in a low-latency manner.

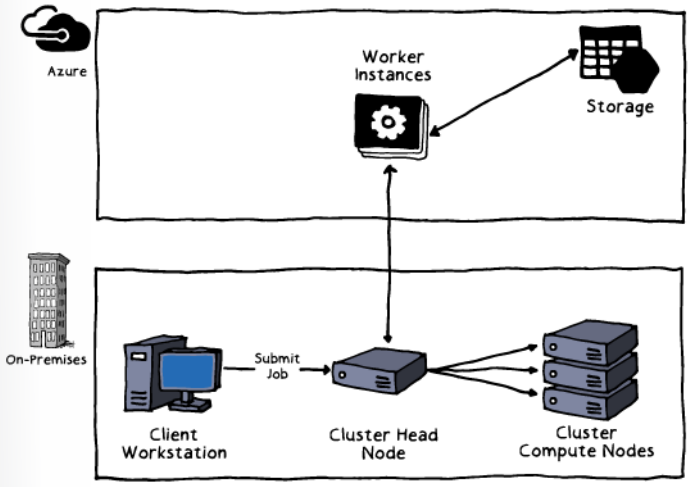
HPC using Microsoft Azure Virtual Machines

Azure H-series virtual machines (VMs) are the latest in HPC VMs aimed at high-end computational needs, like molecular modeling and computational fluid dynamics. These 8 and 16 virtual CPU (vCPU) VMs are built on the Intel Haswell processor technology featuring a later generation (DDR4) memory and solid-state drive (SSD)-based temporary storage.

In addition to substantial CPU power, the H-series offers diverse options for low-latency Remote Direct Memory Access (RDMA) networking using Fourteen Data Rate (FDR) InfiniBand and several memory configurations to support memory-intensive computational requirements.

HPC Pack using Azure Virtual Machines

Microsoft HPC Pack is the Microsoft HPC cluster and job management solution for Windows. HPC Pack can be installed on a server that functions as the "head node," and the server can be used to manage compute nodes in an HPC cluster. HPC Pack can also be used in hybrid scenarios where you want to "burst to Azure" with cloud instances to obtain more processing power.



Starting with HPC Pack 2012 R2 Update 2, HPC Pack has supported several Linux distributions to run on compute nodes deployed in Azure VMs, managed by a Windows Server head node. With the latest release of HPC Pack, you can deploy a Linux-based cluster that can run Message Passing Interface (MPI) applications that access the RDMA network in Azure. Using HPC Pack, you can create a cluster of virtual machines using either Windows or Linux that use the Intel MPI Library to spread the workload of simulations and computations among the compute nodes of many virtual machines.

Remote Direct Memory Access (RDMA)

RDMA is a technology that provides a low-latency network connection between the processing running on two servers or virtual machines in Azure. This technology is essential for engineering simulations and other compute applications that are too large to fit in the memory of a single machine. From a developer's perspective, RDMA is implemented in a way that makes it seem as if the machines are sharing memory. RDMA is efficient, because it copies data from the network adapter directly to memory and avoids wasting CPU cycles.

A subset of the compute-intensive instances (H16r, H16mr, A8, and A9) feature a network interface for RDMA connectivity. Selected N-series sizes designated with "r" (such as NC24r) are also RDMA-capable. This interface is available in addition to the standard Azure network interface available to other VM sizes.

This interface allows the RDMA-capable instances to communicate over an InfiniBand network, operating at FDR rates for H16r, H16mr, and RDMA-capable N-series VMs and at Quad Data Rate (QDR) rates for A8 and A9 VMs. These RDMA capabilities can boost the scalability and performance of certain MPI applications.

Azure Batch for HPC

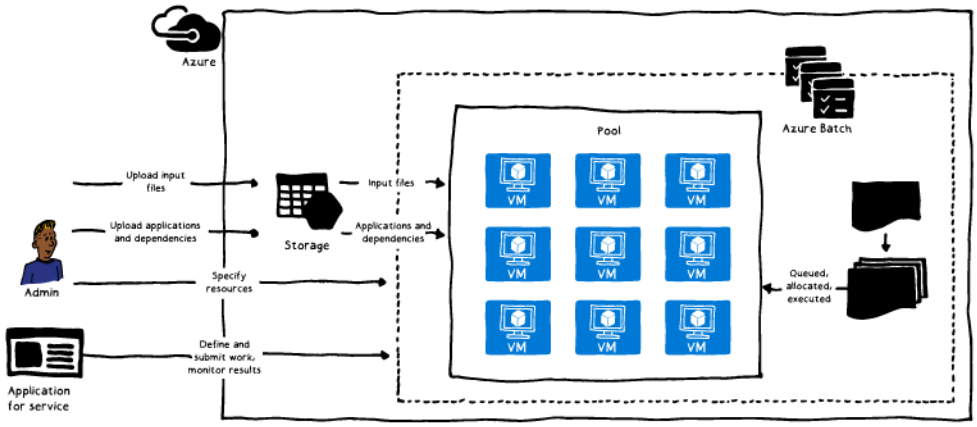
Azure Batch is a service that manages VMs for large-scale parallel and HPC applications. Batch is a platform as a service (PaaS) offering that manages the VMs necessary for your compute jobs for you

instead of forcing you to manage an HPC cluster or job schedule. Batch provides autoscaling functionality and job scheduling functionality in addition to managing compute nodes.

Batch computing is a common pattern for organizations that process, transform, and analyze large amounts of data either on a schedule or on demand. It includes end-of-cycle processing, such as a bank's daily risk reporting or a payroll that must be done on schedule. It also includes large-scale business, science, and engineering applications that typically need the tools and resources of a compute cluster or grid. Applications include traditional HPC applications, such as fluid dynamics simulations and specialized workloads in fields ranging from digital content creation to financial services to life sciences research. Batch works well with intrinsically parallel (sometimes called "embarrassingly parallel") applications or workloads, which lend themselves to running as parallel tasks on multiple computers.

Scaling out parallel workloads

The Batch API can handle scaling out an intrinsically parallel workload, such as image rendering, on a pool of up to thousands of compute cores. Instead of having to set up a compute cluster or write code to queue and schedule your jobs and move the necessary input and output data, you automate the scheduling of large compute jobs and scale a pool of compute VMs up and down to run them. You can write client apps or front ends to run jobs and tasks on demand, on a schedule, or as part of a larger workflow managed by tools such as Azure Data Factory.



You can also use the Batch API to wrap an existing application so it runs as a service on a pool of compute nodes that Batch manages in the background. The application might be one that runs today on client workstations or a compute cluster. You can develop the service to let users offload peak work to the cloud or to run their work entirely in the cloud. The Batch Apps framework handles the movement of input and output files, the splitting of jobs into tasks, job and task processing, and data persistence.

Implement resilient apps by using queues

Asynchronous messaging

Many software curriculums teach that separating your application into smaller modules will make it easier to manage the application over the long term. Modules can be swapped, modified, and updated without having to update the entire application. Partitioning your workloads into modules also carries the additional benefit of allowing each logic center in your application to scale in isolation.

If you have a web application that allows people to upload images for processing, your image processing module can become CPU intensive and easily account for the majority of your CPU time and disk usage. By separating the image processing module out to another distinct server (or set of servers), you can scale this module in isolation without having to modify, scale, or change the module that serves the web pages. It then becomes very important to figure out how to communicate among these modules.

Messaging is a key strategy employed in many distributed environments, such as the cloud. It enables applications and services to communicate and cooperate and can help to build scalable and resilient solutions. Messaging supports asynchronous operations, enabling you to decouple a process that consumes a service from the process that implements the service.

Message queues

Asynchronous messaging in the cloud is usually implemented by using message queues. Regardless of the technology used to implement them, most message queues support three fundamental operations:

- A sender can post a message to the queue.
- A receiver can retrieve a message from the queue (the message is removed from the queue).
- A receiver can examine (or peek at) the next available message in the queue (the message is not removed from the queue).

Conceptually, you can think of a message queue as a buffer that supports send and receive operations. A sender constructs a message in an agreed format and posts the message to a queue. A receiver retrieves the message from the queue and processes it. If a receiver attempts to retrieve a message from an empty queue, the receiver may be blocked until a new message arrives on that queue. Many message queues enable a receiver to query the current length of a queue or peek to see whether one or more messages are available, enabling the receiver to avoid being blocked if the queue is empty.

Many modern queue systems also support transactions, visibility, and leasing on top of the standard queue operations.

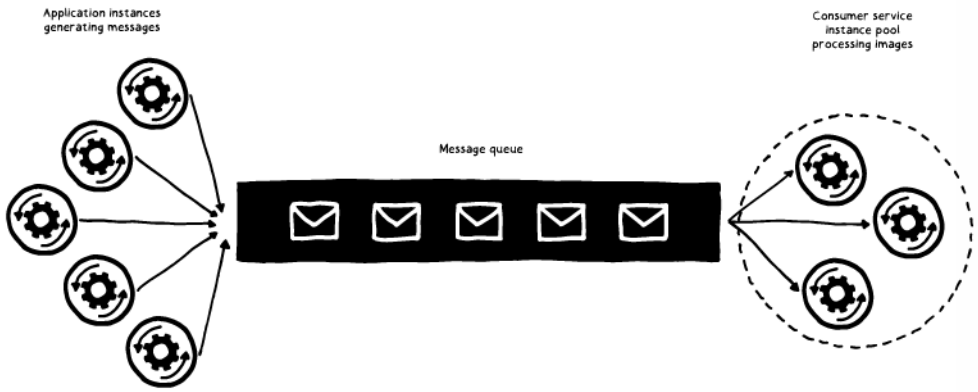
Concurrent queue consumers

An application running in the cloud is expected to handle a large number of requests. Rather than processing each request synchronously, a common technique is for the application to pass them through a messaging system to another service (a consumer service) that handles them asynchronously. This

strategy helps to ensure that the business logic in the application isn't blocked while the requests are being processed.

The number of requests can vary significantly over time for many reasons. A sudden increase in user activity or aggregated requests coming from multiple tenants can cause an unpredictable workload. At peak hours, a system might need to process many hundreds of requests per second, while at other times, the number could be very small. Additionally, the nature of the work performed to handle these requests might be highly variable. Using a single instance of the consumer service can cause that instance to become flooded with requests, or the messaging system might be overloaded by an influx of messages coming from the application. To handle this fluctuating workload, the system can run multiple instances of the consumer service. However, these consumers must be coordinated to ensure that each message is delivered only to a single consumer. The workload also needs to be load balanced across consumers to prevent an instance from becoming a bottleneck.

Use a message queue to implement the communication channel between the application and the instances of the consumer service. The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them. This approach enables the same pool of consumer service instances to handle messages from any instance of the application. The figure illustrates using a message queue to distribute work to instances of a service.



Azure provides **Azure Storage queues** and **Azure Service Bus queues** that both act as mechanisms for implementing this pattern.

Using Azure Storage queues in code

Azure Queue storage provides cloud messaging between application components. In designing applications for scale, application components are often decoupled so that they can scale independently. Queue storage delivers asynchronous messaging for communication between application components, whether they are running in the cloud, on the desktop, on an on-premises server, or on a mobile device. Queue storage also supports managing asynchronous tasks and building process workflows.

Sending a message to a queue

To insert a message into an existing queue, first create a new **CloudQueueMessage**. Next, call the **AddMessage** method. A **CloudQueueMessage** can be created from either a string (in UTF-8 format) or a byte array. Here is code that creates a queue (if it doesn't exist):

```
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(CloudConfigu-
rationManager.GetSetting("StorageConnectionString"));
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();
```

```
CloudQueue queue = queueClient.GetQueueReference("myqueue");
queue.CreateIfNotExists();
```

The next code sample inserts the message **'Hello, World**' into the referenced queue:

```
CloudQueueMessage message = new CloudQueueMessage("Hello, World");
queue.AddMessage(message);
```

Receiving messages from a queue

You can peek at the message in the front of a queue without removing it from the queue by calling the **PeekMessage** method:

```
CloudQueueMessage peekedMessage = queue.PeekMessage();


Console.WriteLine(peekedMessage.AsString);
```

Your code dequeues a message from a queue in two steps. When you call **GetMessage**, you get the next message in a queue. A message returned from **GetMessage** becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call **DeleteMessage**. This two-step process of removing a message assures that if your code fails to process a message due to a hardware or software failure, another instance of your code can get the same message and try again. Your code calls **DeleteMessage** right after the message has been processed:

```
CloudQueueMessage retrievedMessage = queue.GetMessage();
…
queue.DeleteMessage(retrievedMessage);
```

Using Azure Service Bus queues in code

Azure Service Bus supports a set of cloud-based, message-oriented middleware technologies, including reliable message queuing and durable publish/subscribe messaging. These "brokered" messaging capabilities can be thought of as decoupled messaging features that support publish-subscribe, temporal decoupling, and load balancing scenarios using the Service Bus messaging workload.

Service Bus queues offer **first in, first out (FIFO)** message delivery to one or more competing consumers. That is, receivers typically receive and process messages in the order in which they were added to the queue, and only one message consumer receives and processes each message. A key benefit of using queues is achieving the temporal decoupling of application components.

Sending messages to a queue

Using Service Bus, you create a client that grants access to a specific queue for message processing. To create this client, you should use the **QueueClient** class, passing in a connection string for the Service Bus namespace and the name of a specific queue:

```
QueueClient queueClient = new QueueClient(ServiceBusConnectionString,
QueueName);
```

Once the queue client has been created, you can use the client to send one or more messages to the queue. To send a single message, you simpy need to use the **SendAsync** method. To send a group of messages, you can use the **SendBatchAsync** method. Both methods require that you encapsulate the queue message in an object of type **Message**:

```
string messageBody = $"First Message";
Message message = new Message(Encoding.UTF8.GetBytes(messageBody));
await queueClient.SendAsync(message);

var messages = new List<Message>();
for (int i = 0; i < 10; i++)
{
    var message = new Message(Encoding.UTF8.GetBytes($"Message {i:00})};
    messages.Add(message);
}

await queueClient.SendBatchAsync(messages);
```

Receiving messages from a queue

Using the same queue client, you can register a method that will handle any new messages that arrive asynchronously:

```
queueClient.RegisterMessageHandler(MethodToHandleNewMessage, new Register-
HandlerOptions());
```

Now, you will need to implement a new method to handle each message that may come in. Within this method, you can implement any business logic you like, but it must conform to a specific method signature—**Handler(Message, CancellationToken)**. As an example, this method will write each message to the console:

```
static async Task MethodToHandleNewMessage (Message message, Cancellation-
Token token)
{
    string messageString = Encoding.UTF8.GetString(message.Body);
    Console.WriteLine($"Received message: {messageString}");
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);
}
```

You will notice that we need to call the **CompleteAsync** method of the queue client at the end of the message handler method. This ensures that the message is not received again. Alternatively, you can call **AbandonAsync** if you wish to stop handling the message and receive it again.

Implement code to address application events by using webhooks

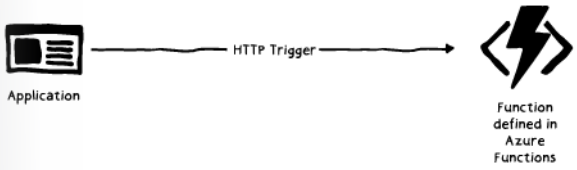
Webhooks

Most applications look at external data and **react** to changes to that data. For example, your retail application may need to watch a database and wait to see if any records are updated to a status of "pending shipment". If a record has that value, your application will ship a physical package and then update the record's status to "shipped". To implement this logic in the past, we would have needed to query the database on a regular basis. This technique is referred to as polling. While it is a simple technique to implement, it has significant drawbacks. Polling requires you to overallocate resources to checking a data store or third-party service even if there are no changes waiting.

As the internet evolved, many applications began implementing "user-defined HTTP callbacks." In this pattern, the third-party service or data store would contact the application whenever it had changes. This saves your application from having to waste compute cycles polling data. This pattern is referred to as **webhooks**. When implementing a webhook, you give an external resource a URL for your application. The external resource will then issue an HTTP request to that URL whenever a change is made that requires your application to take action.

Implementing webhooks using Azure Functions

By far, the simplest way to implement an application that supports webhooks in Azure is to use the **Azure Functions** service. When defining a function with Azure Functions, you define a **trigger** and then **code** that will execute once triggered.



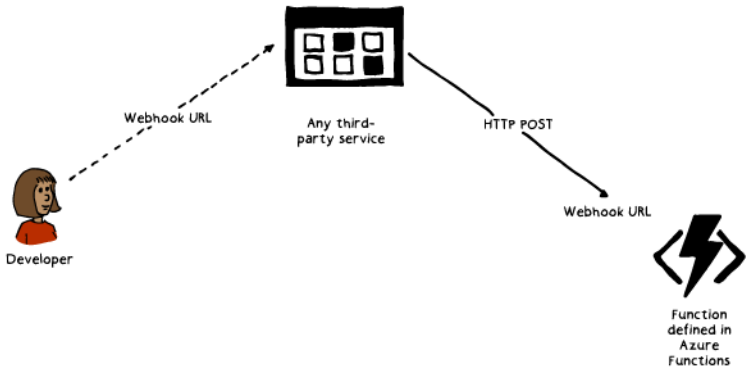
In this simple example, the function is triggered by an HTTP POST request, and it echoes the request body:

```
{
    "disabled": false,
    "bindings": [
        {
            "authLevel": "function",
            "name": "req",
            "type": "httpTrigger",
            "direction": "in",
            "methods": [
                "post"
            ]
        },
        {
            "name": "$return",
```

```
            "type": "http",
            "direction": "out"
        }
    ]
}

module.exports = function(context, req) {
    context.res = {
          body: req.body
    };
    context.done();
};
```

A webhook-based function defined with Azure Functions will create a URL that can be accessed by any external or third-party service. The external (or third-party) service will then issue an HTTP POST request to this URL whenever there is a change. The function's code will then execute any business logic relevant to the change:



The following example shows a webhook trigger binding in a function.json file and a JavaScript function that uses the binding. This example function creates a webhook in GitHub that is called whenever there is an issue comment. GitHub then issues an HTTP POST request to the function URL whenever a comment is created for an issue. The function will then be triggered and handle the new comment:

```
{
    "bindings": [
          {
                "type": "httpTrigger",
                "direction": "in",
                "webHookType": "github",
                "name": "req"
          },
          {
                "type": "http",
                "direction": "out",
                "name": "res"
          }
    ],
    "disabled": false
```

```
}

module.exports = function (context, data) {
    context.log('GitHub WebHook triggered!', data.comment.body);
    context.res = { body: 'New GitHub comment: ' + data.comment.body };
    context.done();
};
```

Online Lab - Configuring a Message-Based Integration Architecture

Lab Steps

Online Lab: Configuring a Message-Based Integration Architecture

NOTE: For the most recent version of this online lab, see: **https://github.com/MicrosoftLearning/AZ-300-MicrosoftAzureArchitectTechnologies**

Scenario

Adatum has several web applications that process files uploaded in regular intervals to their on-premises file servers. Files sizes vary, but they can reach up to 100 MB. Adatum is considering migrating the applications to Azure App Service or Azure Functions-based apps and using Azure Storage to host uploaded files. You plan to test two scenarios:

- using Azure Functions to automatically process new blobs uploaded to an Azure Storage container.
- using Event Grid to generate Azure Storage queue messages that will reference new blobs uploaded to an Azure Storage container.

These scenarios are intended to address a challenge common to a messaging based architecture, when sending, receiving, and manipulating large messages. Sending large messages to a message queue directly is not recommended as they would require more resources to be used, result in more bandwidth to be consumed, and negatively impact the processing speed, since messaging platforms are usually fine-tuned to handle high volumes of small messages. In addition, messaging platforms usually limit the maximum message size they can process.

One potential solution is to store the message payload into an external service, like Azure Blob Store, Azure SQL or Azure Cosmos DB, get the reference to the stored payload and then send to the message bus only that reference. This architectural pattern is known as "claim check". The clients interested in processing that specific message can use the obtained reference to retrieve the payload, if needed. On Azure this pattern can be implemented in several ways and with different technologies, but it typically it relies on eventing to either automate the claim check generation and to push it into the message bus to be used by clients or to trigger payload processing directly. One of the common components included in such implementations is Event Grid, which is an event routing service responsible for delivery of events within a configurable period of time (up to 24 hours). After that, events are either discarded or dead lettered. If archival of event contents or replayability of event stream are needed, it is possible to facilitate this requirement by setting up an Event Grid subscription to the Event Hub or a queue in Azure Storage where messages can be retained for longer periods and archival of messages is supported.

In this lab, you will use Azure Storage Blob service to store files to be processed. A client just needs to drop the files to be shared into a designated Azure Blob container. In the first exercise, the files will be consumed directly by an Azure Function, leveraging its serverless nature. You will also take advantage of the Application Insights to provide instrumentation, facilitating monitoring and analyzing file processing. In the second exercise, you will use Event Grid to automatically generate a claim check message and send it to an Azure Storage queue. This allows a client application to poll the queue, get the message and then use the stored reference data to download the payload directly from Azure Blob Storage.

It is worth noting that the Azure Function in the first exercise relies on the Blob Storage trigger. You should opt for Event Grid trigger instead of the Blob storage trigger when dealing with the following requirements:

- blob-only storage accounts: blob storage accounts are supported for blob input and output bindings but not for blob triggers. Blob storage triggers require a general-purpose storage account.
- high scale: high scale can be loosely defined as containers that have more than 100,000 blobs in them or storage accounts that have more than 100 blob updates per second.
- reduced latency: if your function app is on the Consumption plan, there can be up to a 10-minute delay in processing new blobs if a function app has gone idle. To avoid this latency, you can use an Event Grid trigger or switch to an App Service plan with the Always On setting enabled.
- processing of blob delete events: blob delete events are not supported by blob storage triggers.

Objectives

After completing this lab, you will be able to:

- Configure and validate an Azure Function App Storage Blob trigger
- Configure and validate an Azure Event Grid subscription-based queue messaging

Lab

Estimated Time: 60 minutes

Exercise 1: Configure and validate an Azure Function App Storage Blob trigger

The main tasks for this exercise are as follows:

1. Configure an Azure Function App Storage Blob trigger
2. Validate an Azure Function App Storage Blob trigger

Task 1: Configure an Azure Function App Storage Blob trigger

1. From the lab virtual machine, start Microsoft Edge, browse to the Azure portal at **http://portal.azure.com**, and sign in by using the Microsoft account that has the Owner role in the target Azure subscription.

2. In the Azure portal, in the Microsoft Edge window, start a **Bash** session within the **Cloud Shell**.
3. If you are presented with the **You have no storage mounted** message, configure storage using the following settings:
   - Subsciption: the name of the target Azure subscription
   - Cloud Shell region: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Resource group: **az300T0601-LabRG**
   - Storage account: a name of a new storage account
   - File share: a name of a new file share
4. From the Cloud Shell pane, run the following to generate a pseudo-random string of characters that will be used as a prefix for names of resources you will provision in this exercise:

```
export PREFIX=$(echo `openssl rand 5 -base64 | cut -c1-7 | tr '[:upper:]' '[:lower:]' | tr -cd '[[:alnum:]]._-'`)
```

5. From the Cloud Shell pane, run the following to designate the Azure region into which you want to provision resources in this lab (make sure to replace the placeholder `<Azure region>` with the name of the target Azure region):

```
export LOCATION='<Azure_region>'
```

6. From the Cloud Shell pane, run the following to create a resource group that will host all resources that you will provision in this lab:

```
export RESOURCE_GROUP_NAME='az300T0602-LabRG'

az group create --name "${RESOURCE_GROUP_NAME}" --location $LOCATION
```

7. From the Cloud Shell pane, run the following to create an Azure Storage account and a container that will host blobs to be processed by the Azure function:

```
export STORAGE_ACCOUNT_NAME="az300t06st2${PREFIX}"

export CONTAINER_NAME="workitems"

export STORAGE_ACCOUNT=$(az storage account create --name "${STORAGE_ACCOUNT_NAME}" --kind "StorageV2" --location "${LOCATION}" --resource-group "${RESOURCE_GROUP_NAME}" --sku "Standard_LRS")

az storage container create --name "${CONTAINER_NAME}" --account-name "${STORAGE_ACCOUNT_NAME}"
```

8. **Note**: The same storage account will be also used by the Azure function to facilitate its own processing requirements. In real-world scenarios, you might want to consider creating a separate storage account for this purpose.
9. From the Cloud Shell pane, run the following to create a variable storing the value of the connection string property of the Azure Storage account:

```
export STORAGE_CONNECTION_STRING=$(az storage account show-connec-
tion-string --name "${STORAGE_ACCOUNT_NAME}" --resource-group "${RESOURCE_
GROUP_NAME}" -o tsv)
```

10. From the Cloud Shell pane, run the following to create an Application Insights resource that will provide monitoring of the Azure Function processing blobs and store its key in a variable:

```
export APPLICATION_INSIGHTS_NAME="az300t06ai${PREFIX}"

az resource create --name "${APPLICATION_INSIGHTS_NAME}" --location "${LO-
CATION}" --properties '{"Application_Type": "other", "ApplicationId":
"function", "Flow_Type": "Redfield"}' --resource-group "${RESOURCE_GROUP_
NAME}" --resource-type "Microsoft.Insights/components"

export APPINSIGHTS_KEY=$(az resource show --name "${APPLICATION_INSIGHTS_
NAME}" --query "properties.InstrumentationKey" --resource-group "${RE-
SOURCE_GROUP_NAME}" --resource-type "Microsoft.Insights/components" -o tsv)
```

11. From the Cloud Shell pane, run the following to create the Azure Function that will process events corresponding to creation of Azure Storage blobs:

```
export FUNCTION_NAME="az300t06f${PREFIX}"

az functionapp create --name "${FUNCTION_NAME}" --resource-group "${RE-
SOURCE_GROUP_NAME}" --storage-account "${STORAGE_ACCOUNT_NAME}" --consump-
tion-plan-location "${LOCATION}"
```

12. From the Cloud Shell pane, run the following to configure Application Settings of the newly created function, linking it to the Application Insights and Azure Storage account:

```
az functionapp config appsettings set --name "${FUNCTION_NAME}" --re-
source-group "${RESOURCE_GROUP_NAME}" --settings "APPINSIGHTS_INSTRUMENTA-
TIONKEY=$APPINSIGHTS_KEY" FUNCTIONS_EXTENSION_VERSION=~2

az functionapp config appsettings set --name "${FUNCTION_NAME}" --re-
source-group "${RESOURCE_GROUP_NAME}" --settings "STORAGE_CONNECTION_
STRING=$STORAGE_CONNECTION_STRING" FUNCTIONS_EXTENSION_VERSION=~2
```

13. Switch to the Azure portal and navigate to the blade of the Azure Function app you created earlier in this task.
14. On the Azure Function app blade, click **Functions** and then, click **+ New function**.
15. On the **Function App runtime stack** blade, ensure that the **.NET** entry appears in the **Function Runtime stack** drop down list and click **Go**.
16. On the **Choose a template below or go to the quickstart** blade, click **Azure Blob Storage trigger** template.
17. On the **Extensions not Installed** blade, click **Install** and wait until the installation of the extension completes. Wait until the installation completes and click **Continue**.

18. **Note**: Azure Functions V2 runtime implement bindings in the form of Nuget packages, referred to as "binding extensions" (in particular, the Azure Storage Blob binding uses the Microsoft.Azure.WebJobs.Extensions.Storage package).
19. On the **Azure Blob Storage trigger** blade, specify the following and click **Create** to create a new function within the Azure function:
    - Name: **BlobTrigger**
    - Path: **workitems/{name}**
    - Storage account connection: **STORAGE_CONNECTION_STRING**
20. On the Azure Function app **BlobTrigger** function blade, review the content of the run.csx file.

```
public static void Run(Stream myBlob, string name, ILogger log)
{
    log.LogInformation($"C# Blob trigger function Processed blob\n
Name:{name} \n Size: {myBlob.Length} Bytes");
}
```

21. **Note**: By default, the function is configured to simply log the event corresponding to creation of a new blob. In order to carry out blob processing tasks, you would modify the content of this file.

Task 2: Validate an Azure Function App Storage Blob trigger

1. If necessary, restart the Bash session in the Cloud Shell.
2. From the Cloud Shell pane, run the following to repopulate variables that you used in the previous task:

```
export RESOURCE_GROUP_NAME='az300T0602-LabRG'

export STORAGE_ACCOUNT_NAME="$(az storage account list --resource-group
"${RESOURCE_GROUP_NAME}" --query "[0].name" --output tsv)"

export CONTAINER_NAME="workitems"
```

3. From the Cloud Shell pane, run the following to upload a test blob to the Azure Storage account you created earlier in this task:

```
export STORAGE_ACCESS_KEY="$(az storage account keys list --account-name
"${STORAGE_ACCOUNT_NAME}" --resource-group "${RESOURCE_GROUP_NAME}" --query
"[0].value" --output tsv)"

export WORKITEM='workitem1.txt'

touch "${WORKITEM}"

az storage blob upload --file "${WORKITEM}" --container-name "${CONTAINER_
NAME}" --name "${WORKITEM}" --auth-mode key --account-key "${STORAGE_AC-
```

```
CESS_KEY}" --account-name "${STORAGE_ACCOUNT_NAME}"
```

4. In the Azure portal, navigate back to the blade displaying the Azure Function app you created in the previous task.
5. On the Azure Function app blade, click **Monitor** entry in the **Functions** section.
6. Note a single event entry representing uploading of the blob. Click the entry to view the **Invocation Details** blade.
7. **Note**: Since the Azure function app in this exercise runs in the Consumption plan, there may be a delay of up to several minutes between uploading a blob and the function being triggered. It is possible to minimize the latency by implementing the Function app by using an App Service (rather than Consumption) plan.
8. On the **Invocation Details** blade, click the link **Run in Application Insights**.
9. In the Application Insights portal, review the autogenerated Kusto query and its results.

Exercise 2: Configure and validate an Azure Event Grid subscription-based queue messaging

The main tasks for this exercise are as follows:

1. Configure an Azure Event Grid subscription-based queue messaging
2. Validate an Azure Event Grid subscription-based queue messaging

Task 1: Configure an Azure Event Grid subscription-based queue messaging

1. If necessary, restart the Bash session in the Cloud Shell.
2. From the Cloud Shell pane, run the following to generate a pseudo-random string of characters that will be used as a prefix for names of resources you will provision in this exercise:

```
export PREFIX=$(echo `openssl rand 5 -base64 | cut -c1-7 | tr '[:upper:]'
'[:lower:]' | tr -cd '[[:alnum:]]._-'`)
```

3. From the Cloud Shell pane, run the following to identify the Azure region hosting the target resource group and its existing resources:

```
export RESOURCE_GROUP_NAME_EXISTING='az300T0602-LabRG'

export LOCATION=$(az group list --query "[?name == '${RESOURCE_GROUP_NAME_
EXISTING}'].location" --output tsv)

export RESOURCE_GROUP_NAME='az300T0603-LabRG'
```

```
az group create --name "${RESOURCE_GROUP_NAME}" --location $LOCATION
```

4. From the Cloud Shell pane, run the following to create an Azure Storage account and its container that will be used by the Event Grid subscription that you will configure in this task:

```
export STORAGE_ACCOUNT_NAME="az300t06st3${PREFIX}"

export CONTAINER_NAME="workitems"

export STORAGE_ACCOUNT=$(az storage account create --name "${STORAGE_AC-
COUNT_NAME}" --kind "StorageV2" --location "${LOCATION}" --resource-group
"${RESOURCE_GROUP_NAME}" --sku "Standard_LRS")

az storage container create --name "${CONTAINER_NAME}" --account-name
"${STORAGE_ACCOUNT_NAME}"
```

5. From the Cloud Shell pane, run the following to create a variable storing the value of the Resource Id property of the Azure Storage account:

```
export STORAGE_ACCOUNT_ID=$(az storage account show --name "${STORAGE_AC-
COUNT_NAME}" --query "id" --resource-group "${RESOURCE_GROUP_NAME}" -o tsv)
```

6. From the Cloud Shell pane, run the following to create the Storage Account queue that will store messages generated by the Event Grid subscription that you will configure in this task:

```
export QUEUE_NAME="az300t06q3${PREFIX}"

az storage queue create --name "${QUEUE_NAME}" --account-name "${STORAGE_
ACCOUNT_NAME}"
```

7. From the Cloud Shell pane, run the following to create the Event Grid subscription that will facilitate generation of messages in Azure Storage queue in response to blob uploads to the designated container in the Azure Storage account:

```
export QUEUE_SUBSCRIPTION_NAME="az300t06qsub3${PREFIX}"

az eventgrid event-subscription create --name "${QUEUE_SUBSCRIPTION_NAME}"
--included-event-types 'Microsoft.Storage.BlobCreated' --endpoint "${STOR-
AGE_ACCOUNT_ID}/queueservices/default/queues/${QUEUE_NAME}" --endpoint-type
"storagequeue" --source-resource-id "${STORAGE_ACCOUNT_ID}"
```

Task 2: Validate an Azure Event Grid subscription-based queue messaging

1. From the Cloud Shell pane, run the following to upload a test blob to the Azure Storage account you created earlier in this task:

```
export AZURE_STORAGE_ACCESS_KEY="$(az storage account keys list --ac-
count-name "${STORAGE_ACCOUNT_NAME}" --resource-group "${RESOURCE_GROUP_
NAME}" --query "[0].value" --output tsv)"

export WORKITEM='workitem2.txt'

touch "${WORKITEM}"

az storage blob upload --file "${WORKITEM}" --container-name "${CONTAINER_
NAME}" --name "${WORKITEM}" --auth-mode key --account-key "${AZURE_STORAGE_
ACCESS_KEY}" --account-name "${STORAGE_ACCOUNT_NAME}"
```

2. In the Azure portal, navigate to the blade displaying the Azure Storage account you created in the previous task of this exercise.
3. On the blade of the Azure Storage account, click **Queues** to display the list of its queues.
4. Click the entry representing the queue you created in the previous task of this exercise.
5. Note that the queue contains a single message. Click its entry to display the **Message properties** blade.
6. In the **MESSAGE BODY**, note the value of the **url** property, representing the URL of the Azure Storage blob you uploaded in the previous task.

Exercise 3: Remove lab resources

The main tasks for this exercise are as follows:

1. Remove lab resources

Task 1: Remove lab resources

1. In the Azure portal, start the Bash session in the Cloud Shell.
2. From the Cloud Shell pane, run the following to delete resource groups that host all resources that you provisioned in this lab

```
for RESOURCE_GROUP_NAME in 'az300T0602-LabRG' 'az300T0603-LabRG'
do
   az group delete --name "${RESOURCE_GROUP_NAME}" --no-wait --yes
done
```

Review Questions

Module 1 Review Questions

Azure Batch

You are designing a video editing solution. The solution will require extension hardware resources to perform require processing on large video files.

You need to recommend an Azure high performance computing option for the solution.

What options are available? Which option should you recommend?

Suggested Answer ↓

Azure Batch is a service that manages VMs for large-scale parallel and HPC applications. Batch is a platform as a service (PaaS) offering that manages the VMs necessary for your compute jobs for you instead of forcing you to manage an HPC cluster or job schedule. Batch provides autoscaling functionality and job scheduling functionality in addition to managing compute nodes.

Messaging

You manage an e-Commerce solution that uses Azure Batch.

During periods of high user activity, several thousand complex operations run at the same time.

How does Azure Batch handle concurrent operations?

Suggested Answer ↓

An application running in the cloud is expected to handle a large number of requests. Rather than processing each request synchronously, a common technique is for the application to pass them through a messaging system to another service (a consumer service) that handles them asynchronously. This strategy helps to ensure that the business logic in the application isn't blocked while the requests are being processed.

Azure Batch uses a message queue to implement the communication channel between the application and the instances of the consumer service. The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them. This approach enables the same pool of consumer service instances to handle messages from any instance of the application.

Azure Service Bus

You manage a solution that uses Azure Batch. The solution uses an Azure Service Bus message queue to process data during periods of peak activity.

In what order are messages processed?

Suggested Answer ↓

Service Bus queues offer first in, first out (FIFO) message delivery to one or more competing consumers. Messages are processed in the order in which they were added to the queue.

Module 2 Module Configuring a Message-Based Integration Architecture

Configure an app or service to send emails

SendGrid

SendGrid (**https://sendgrid.com**) is a third-party, cloud-based email messaging service that has a deep level of integration with the Azure platform and portal. SendGrid provides transactional email delivery, scalability based on email volume, and real-time analytics for the sent messages. SendGrid also has a flexible API to enable custom integration scenarios.

Sending emails through SendGrid by using C#

The SendGrid team provides an official library, which uses C#, to interact with the SendGrid API. The library is open source and hosted on GitHub at **https://github.com/sendgrid/sendgrid-csharp**. The C# library is also available as a NuGet package to make it simple to import the library into an existing Microsoft .NET project.

Once the library is successfully imported into a .NET project, the various API features can be used by creating a new instance of the **SendGridClient** class and passing in a string parameter containing your **API Key**:

```
var client = new SendGridClient(apiKey);
```

The client class contains multiple methods to perform common API tasks. For example, you can create a new instance of the **SendGridMessage** class and pass it in to the **SendGridClient.SendEmailAsync** method to send a simple email:

```
SendGridMessage message = new SendGridMessage()
{
    From = new EmailAddress("admin@contoso.com", "Contoso Admin"),
    Subject = "Greetings!"
```

```
};
message.AddTo(new EmailAddress("test.person@adventureworks.net", "Test
Person"));
message.AddContent(MimeType.Text, "Hello World");
await client.SendEmailAsync(message);
```

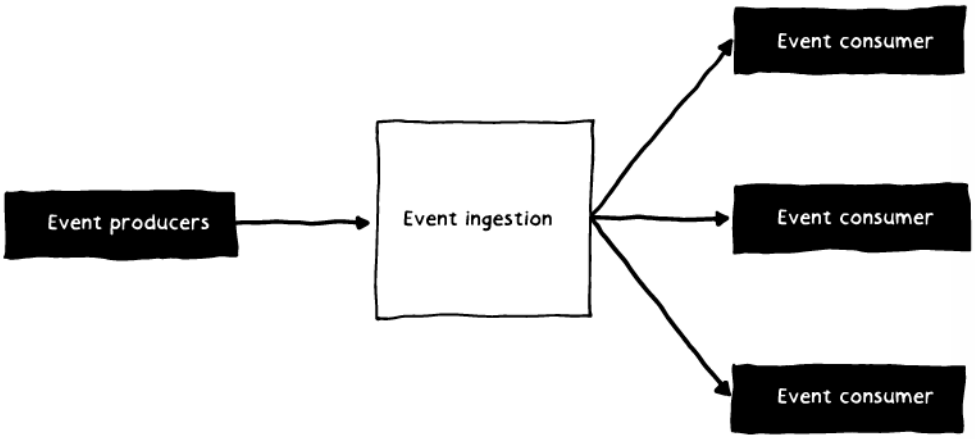
You can also use the C# library to perform common tasks like attaching a file to an e-mail message:

```
Attachment inlineGraphic = new Attachment()
{
    Content = Convert.ToBase64String(raw_image_bytes),
    Type = "image/png",
    Filename = "graphic.png",
    Disposition = "inline",
    ContentId = "Graphic"
};
message.AddAttachment(inlineGraphic);
```

Configure an event publish and subscribe model

Event-driven architecture

Modern application requirements stipulate that applications we build should be able to handle a high volume and velocity of data, process that data in real time, and allow multiple systems to respond to the same data. If we were to build applications in a serial manner, this stipulation would be incredibly difficult to meet. To help handle these application scenarios, many modern systems are built using an architectural style referred to as event-driven architecture. An event-driven architecture consists of event producers that generate a stream of events and event consumers that listen for the events.



In an event-driven architecture, events are delivered in nearly real time, so consumers can respond immediately to events as they occur. Producers are decoupled from consumers—that is, a producer doesn't know which consumers are listening. Consumers are also decoupled from each other, and every consumer sees all of the events.

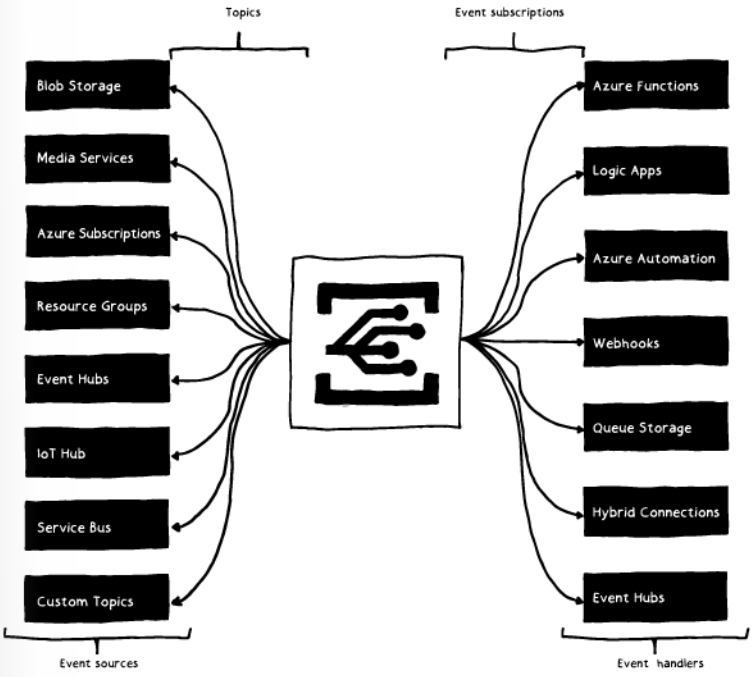
There are a few common implementations of an event-driven architecture that you will commonly see on the Microsoft Azure platform:

- **Simple event processing**. An event immediately triggers an action in the consumer. For example, you could use Azure Functions with an Azure Service Bus trigger so that a function executes whenever a message is published to a Service Bus topic.
- **Complex event processing**. A consumer processes a series of events, looking for patterns in the event data, by using a technology such as Azure Stream Analytics or Apache Storm. For example, you could aggregate readings from an embedded device over a time window and generate a notification if the moving average crosses a certain threshold.
- **Event stream processing**. You use a data streaming platform, such as Azure IoT Hub or Apache Kafka, as a pipeline to ingest events and feed them to stream processors. The stream processors act to process or transform the stream. There may be multiple stream processors for different subsystems of

the application. This approach is a good fit for Internet of Things (IoT) workloads.

Azure Event Grid

Azure Event Grid allows you to easily build applications with event-based architectures. You select the Azure resource you would like to subscribe to and give the event handler or webhook endpoint to send the event to. Event Grid has built-in support for events coming from Azure services, like those for storage blobs and resource groups. Event Grid also has custom support for application and third-party events using custom topics and custom webhooks. You can use filters to route specific events to different endpoints, use multicasting to send events to multiple endpoints, and make sure your events are reliably delivered. Event Grid also has built-in support for custom and third-party events.



The following Azure services can send events to Event Grid:

- Azure subscription management operations
- Custom topics
- Azure Event Hubs
- IoT Hub
- Azure Media Services
- Resource group management operations
- Service Bus
- Azure Blob storage
- General Purpose v2 storage

The following Azure services can handle events from Event Grid:

- Azure Automation
- Azure Functions
- Event Hubs
- Hybrid Connections
- Azure Logic Apps
- Microsoft Flow
- Azure Queue storage
- Webhooks (external)

Subscribing to Blob storage events using Azure CLI

Event Grid connects data sources and event handlers. For example, an Event Grid can instantly trigger a serverless function to run an image analysis each time a new photo is added to a Blob storage container. In this example, we will create an **event source** using Blob storage.

Note: To use Blob storage events, you need either a **Blob storage** account or a **General Purpose v2 storage** account. **General Purpose v2 storage** accounts support all features for all storage services, including those for blobs, files, queues, and tables. A **Blob storage** account is a specialized storage account for storing your unstructured data as blobs (objects) in Azure Storage. Blob storage accounts are like General Purpose v2 storage accounts and share all the great durability, availability, scalability, and performance features that you use today, including 100% API consistency for block blobs and append blobs.

Event Grid topics are Azure resources and must be placed in an Azure resource group. The resource group is a logical collection into which Azure resources are deployed and managed. The **az group create** command will create a new resource group. The following example creates a resource group named **DemoGroup** in the **East US** location:

```
az group create --name DemoGroup --location eastus
```

The **az storage account create** command will create a new Azure Storage account. The following example creates a **Blob storage** account named **demostor** in the **East US** location and uses **locally redundant** replication:

```
az storage account create --name demostor --location eastus --re-
source-group
DemoGroup --sku Standard_LRS --kind BlobStorage --access-tier Hot
```

Now that the storage account has been created, we will need the **ID** of the storage account to **subscribe** to events from the storage account. We can use the **az storage account show** command to return a JavaScript Object Notation (JSON) object containing metadata about the newly created account:

```
az storage account show --name demostor --resource-group DemoGroup
```

Azure CLI also contains the capability to query JSON objects and arrays for specific values. We can further refine the query by using the **–query** parameter to only return the **id** property in **tab-separated values** format. This will effectively print out the value of the **id** JSON property on its own line:

```
az storage account show --name demostor --resource-group DemoGroup --query
id
--output tsv
```

Depending on your OS of choice, you can then store this value in a variable. In this example, we are using a **Bash** shell to store the output of the query in a variable named **storageid**:

```
storageid=\$(az storage account show --name demostor --resource-group
DemoGroup
--query id --output tsv)
```

Before you can create a subscription, you will need somewhere to send the messages. This destination is referred to as an **endpoint**. For our example, we have a web API endpoint at the URL https://contoso.com/api/updates. You subscribe to a topic to tell Event Grid which events you want to track and where to send those events. The following example uses the **az eventgrid event-subscription create** command to subscribe to the storage account you created and pass the URL from your web app as the endpoint for event notifications:

```
az eventgrid event-subscription create --resource-id \$storageid --name
contosostoragesub --endpoint https://contoso.com/api/updates
```

Configure the Azure Relay service

Azure Relay service

The Azure Relay service facilitates hybrid applications by enabling you to more-securely expose services that reside within a corporate enterprise network to the public cloud—without having to open a firewall connection or require intrusive changes to a corporate network infrastructure. Azure Relay supports a variety of different transport protocols and web service standards. The Azure Relay service supports traditional one-way, request/response, and peer-to-peer traffic. It also supports event distribution at internet scope to enable publish/subscribe scenarios and bidirectional socket communication for increased point-to-point efficiency.

In the relayed data transfer pattern, an on-premises service connects to the Azure Relay service through an outbound port and creates a bidirectional socket for communication tied to a particular rendezvous address. The client can then communicate with the on-premises service by sending traffic to the Azure Relay service and targeting the rendezvous address. The Azure Relay service then relays data to the on-premises service through a bidirectional socket dedicated to each client. The client does not need a direct connection to the on-premises service, it is not required to know where the service resides, and the on-premises service does not need any inbound ports open on the firewall.

The key capability elements provided by Azure Relay are bidirectional, unbuffered communication across network boundaries with Transmission Control Protocol (TCP)-like throttling; endpoint discovery; connectivity status; and overlaid endpoint security.

Note: The Azure Relay capabilities differ from network-level integration technologies, such as virtual private network (VPN) technology, in that Azure Relay can be scoped to a single application endpoint on a single machine, while VPN technology is far more intrusive, as it relies on altering the network environment.

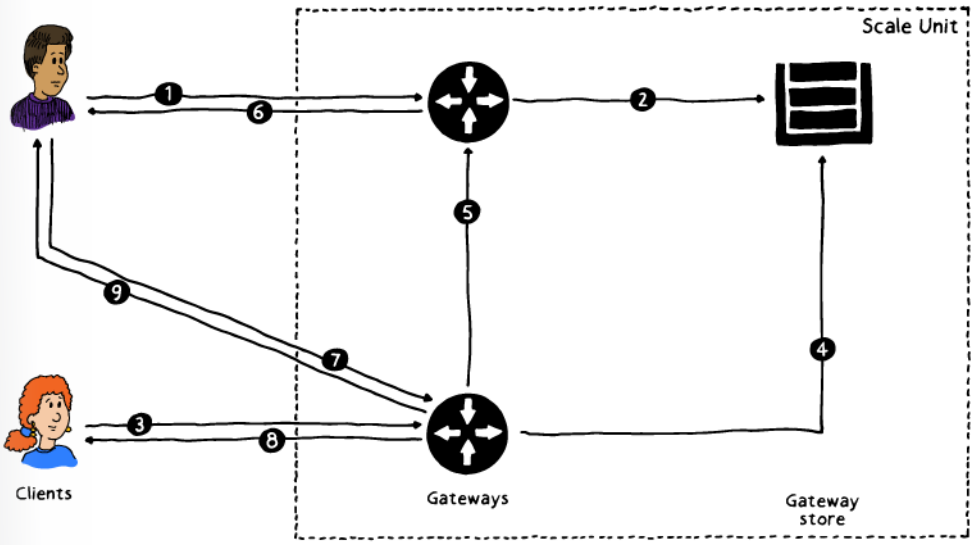
Hybrid Connections

The Hybrid Connections capability of Azure Relay is a more-secure, open-protocol evolution of the existing Azure Relay features that can be implemented essentially on any platform and in any language. Hybrid Connections can relay WebSocket protocol, HTTP, and HTTPS requests and responses. These capabilities are compatible with the WebSocket protocol API in common web browsers. Hybrid Connections is based on HTTP and the WebSocket protocol.

Azure Relay service architecture

When a client sends a request to the Azure Relay service, the Azure load balancer routes it to any of the gateway nodes. If the request is a listening request, the gateway node creates a new relay. If the request is a connection request to a specific relay, the gateway node forwards the connection request to the gateway node that owns the relay. The gateway node that owns the relay sends a rendezvous request to the listening client, asking the listener to create a temporary channel to the gateway node that received the connection request. When

the relay connection is established, the clients can exchange messages via the gateway node that is used for the rendezvous.



If you follow the diagram above, the steps are as follows:

1. A client (Client A) creates a listening request that is routed to a gateway.
2. The gateway creates a new relay.
3. A different client (Client B) creates a connection request.
4. The connection request from Client B is handled first by looking up the associated relay.
5. Once the correct relay has been identified, the request is forwarded to that specific relay.
6. A rendezvous request is sent to Client A.
7. Client A creates a temporary channel to Client B using the original gateway that Client B used in its original connection request.
8. Client B can receive messages sent from Client A directly from its original gateway.
9. Client A can receive messages sent from Client B through Client B's original gateway and the established temporary channel.

Using Azure Relay in Node.js

The **ws** package in Node.js is a WebSocket protocol client library that streamlines the implementation of Node.js applications that communicate using the WebSocket protocol. The **hyco-ws** Node package for Hybrid Connections in Azure Relay is built on and extends the **ws** npm package. This package re-exports all exports of that base package and adds new exports that enable integration with the Hybrid Connections feature of the Azure Relay service. The key differences between the base package and this **hyco-ws** is that it adds a

new server class, exported via **require('hyco-ws').RelayedServer**, and a few helper methods.

Note: Remember, Azure Relay is built on top of the WebSocket protocol, so you can naturally use libraries that support the WebSocket protocol in the language of your choice. In this example, we will use the **hyco-ws** Node package to simplify development against Azure Relay.

There are several utility methods available in the package export that you can reference as follows:

```
const WebSocket = require('hyco-ws');

var listenUri =
WebSocket.createRelayListenUri('namespace.servicebus.windows.net', 'path');

listenUri = WebSocket.appendRelayToken(listenUri, 'ruleName', '...key...')
```

The helper methods are for use with this package but can also be used by a Node server for enabling web or device clients to create listeners or senders. The server uses these methods by passing them URIs that embed short-lived tokens. These URIs can also be used with common WebSocket protocol stacks that do not support setting HTTP headers for the WebSocket protocol handshake. Embedding authorization tokens into the URI is supported primarily for those library-external usage scenarios.

The **hycows.RelayedServer** class is an alternative to the **ws.Server** class that does not listen on the local network but delegates listening to the Azure Relay service.

The two classes are mostly contract compatible, meaning that an existing application using the **ws.Server** class can easily be changed to use the relayed version. The main differences are in the constructor and in the available options. The **RelayedServer** constructor supports a different set of arguments than the **Server**, because it is not a standalone listener or able to be embedded into an existing HTTP listener framework. There are also fewer options available, since the WebSocket protocol management is largely delegated to the Azure Relay service:

```
var server = ws.RelayedServer;
var wss = new server({
          server: ws.createRelayListenUri(ns, path),
          token: function() { return ws.createRelayToken('http://' + ns,
keyrule, key); }
});
```

The **RelayedServer** constructor has two required arguments to establish a connection over the WebSocket protocol using Azure Relay:

- **server** - The fully qualified URI for a Hybrid Connection name on which to listen, usually constructed with the **WebSocket.createRelayListenUri()** helper method.
- **token** - This argument holds either a previously issued token string or a callback function that can be called to obtain such a token string. The callback option is preferred, as it enables token renewal.

Create and configure a notification hub

Azure Notification Hubs

Azure Notification Hubs is a scaled-out push engine that allows you to send notifications to essentially any platform (iOS, Android, Windows, Kindle, BAIDU, etc.) from essentially any back end (cloud or on-premises). You can use Notification Hubs in the following common scenarios:

- Sending breaking news notifications to millions with low latency
- Sending location-based coupons to interested user segments
- Sending event-related notifications to users or groups for media/sports/finance/gaming applications
- Pushing promotional content to applications to engage and market to customers
- Notifying users of enterprise events, like new messages and work items
- Sending codes for multi-factor authentication

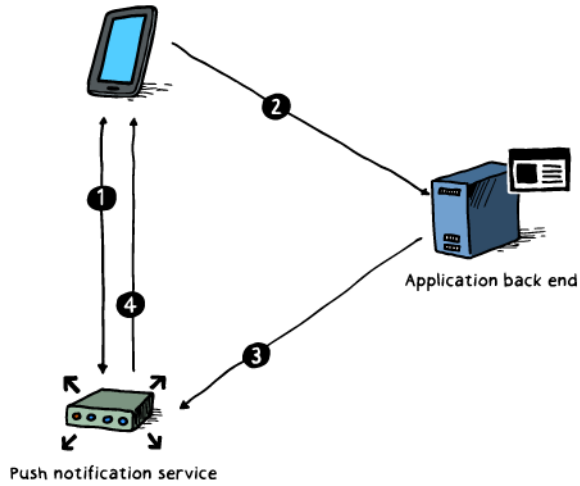
Push notifications

Push notifications are a form of app-to-user communication where users of mobile apps are notified of certain desired information, usually in a pop-up window or dialog box. Users can generally choose to view or dismiss the message. Choosing the former opens the mobile application that communicated the notification. Push notifications are vital for consumer apps in increasing app engagement and usage and for enterprise apps in communicating up-to-date business information. It's the best app-to-user communication, because it is energy efficient for mobile devices, flexible for the notification senders, and available when corresponding applications are not active.

Push notifications are delivered through platform-specific infrastructures called Platform Notification Systems (PNSs). They offer bare-bones push functionalities to deliver a message to a device with a provided handle and have no common interface. To send a notification to all customers across the iOS, Android, and Windows versions of an app, the developer must work with the Apple Push Notification Service (APNs), Firebase Cloud Messaging (FCM), and Windows Push Notification Service (WNS).

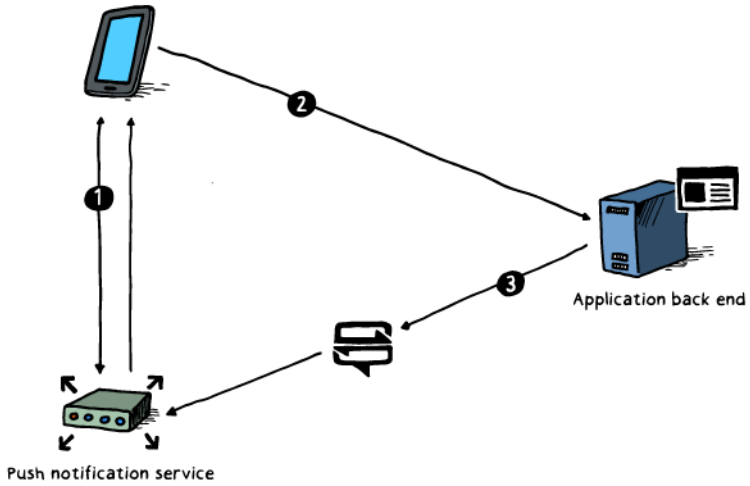
At a high level, here are how push notifications work:

1. The client app decides it wants to receive notifications. Hence, it contacts the corresponding PNS to retrieve its unique and temporary push handle. The handle type depends on the system (for example, WNS has URIs, while APNs has tokens).
2. The client app stores this handle in the app's back end or provider.
3. To send a push notification, the app's back end contacts the PNS using the handle to target a specific client app.
4. The PNS forwards the notification to the device specified by the handle.



5.

Notification Hubs eliminates all the complexities associated with pushing notifications on your own from your app back end. Its multi-platform, scaled-out push notification infrastructure reduces push-related coding and simplifies your back end. With Notification Hubs, devices are merely responsible for registering their PNS handles with a hub, while the back end sends messages to users or interest groups, as shown in the following figure:



Configuring Notification Hubs in iOS

To configure Notification Hubs for an iOS native application, you will need to configure your **Xcode** application with the following prerequisites:

- The **Push Notifications** capability must be enabled.
- The files distributed in the **Azure messaging framework** must be included in the Xcode project.

In most examples, you would first add a few constants to **HubInfo.h** that will contain important connection details for your notification hub:

```
#ifndef HubInfo_h
    #define HubInfo_h
        #define HUBNAME @"<Enter the name of your hub>"
        #define HUBLISTENACCESS @"<Enter your DefaultListenSharedAccess
connection string"
    #endif /* HubInfo_h */
```

After that, you will need to import the **WindowsAzureMessaging/WindowsAzureMessaging** and **UserNotifications/UserNotifications** directives in to your project:

```
#import <WindowsAzureMessaging/WindowsAzureMessaging.h>
#import <UserNotifications/UserNotifications.h>
#import "HubInfo.h"
```

Finally, you can add the following code to connect to the notification hub using the connection information you specified in HubInfo.h. It then gives the device token to the notification hub so that the notification hub can send notifications:

```
-(void) application:(UIApplication *) application didRegisterForRemoteNoti-
ficationsWithDeviceToken:(NSData *) deviceToken {
    SBNotificationHub* hub = [[SBNotificationHub alloc] initWithConnection-
String:HUBLISTENACCESS
                                                    notificationHubPath:HUB-
NAME];

    [hub registerNativeWithDeviceToken:deviceToken tags:nil comple-
tion:^(NSError* error) {
          if (error != nil) {
                NSLog(@"Error registering for notifications: %@", error);
          }
          else {
                [self MessageBox:@"Registration Status" message:@"Regis-
tered"];
          }
    }];
    }

-(void)MessageBox:(NSString *) title message:(NSString *)messageText
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:title mes-
sage:messageText delegate:self
          cancelButtonTitle:@"OK" otherButtonTitles: nil];
    [alert show];
}
```

Configuring Notification Hubs in Xamarin.Android

To configure Notification Hubs for an Android application written using the Xamarin platform, you will need to configure your **Xamarin** project with the following prerequisites:

- The **Xamarin.GooglePlayServices.Base**, **Xamarin.Firebase.Messaging**, and **Xamarin.Azure.NotificationHubs.Android** NuGet packages must be installed.
- The **google-services.json** file must be downloaded from the **Google Firebase Console** and then copied to the root of your project folder.
- You must register the **com.google.firebase.iid.FirebaseInstanceIdReceiver** receiver to enable PNS registration and message receipt.

In most examples, you would first add a few constants to a C# class that will contain important connection details for your notification hub:

```
public static class Constants
{
    public const string ListenConnectionString = "<Listen connection
string>";
    public const string NotificationHubName = "<hub name>";
}
```

First, you will need to create a C# class to manage PNS registration:

```
using Android.App;
using Android.Util;
using WindowsAzure.Messaging;
using Firebase.Iid;

[Service]
[IntentFilter(new[] { "com.google.firebase.INSTANCE_ID_EVENT" })]
public class MyFirebaseIIDService : FirebaseInstanceIdService
{
    const string TAG = "MyFirebaseIIDService";
    NotificationHub hub;

    public override void OnTokenRefresh()
    {
        var refreshedToken = FirebaseInstanceId.Instance.Token;
        Log.Debug(TAG, "FCM token: " + refreshedToken);
        SendRegistrationToServer(refreshedToken);
    }

    void SendRegistrationToServer(string token)
    {
        hub = new NotificationHub(Constants.NotificationHubName,
                                  Constants.ListenConnectionString, this);

        var tags = new List<string>() { };
```

```
            var regID = hub.Register(token, tags.ToArray()).RegistrationId;
        }
    }
```

Second, you will need to create a separate C# class to handle the receipt of a new message and display that message in the application's UI:

```
using Android.App;
using Android.Util;
using Firebase.Messaging;

[Service]
[IntentFilter(new[] { "com.google.firebase.MESSAGING_EVENT" })]
public class MyFirebaseMessagingService : FirebaseMessagingService
{
    const string TAG = "MyFirebaseMsgService";
    public override void OnMessageReceived(RemoteMessage message)
    {
        if(message.GetNotification()!= null)
        {
            SendNotification(message.GetNotification().Body);
        }
        else
        {
            //Only used for debugging payloads sent from the Azure portal
            SendNotification(message.Data.Values.First());

        }

    }

    void SendNotification(string messageBody)
    {
        var intent = new Intent(this, typeof(MainActivity));
        intent.AddFlags(ActivityFlags.ClearTop);
        var pendingIntent = PendingIntent.GetActivity(this, 0, intent,
PendingIntentFlags.OneShot);

        var notificationBuilder = new Notification.Builder(this)
                    .SetContentTitle("FCM Message")
                    .SetSmallIcon(Resource.Drawable.ic_launcher)
                    .SetContentText(messageBody)
                    .SetAutoCancel(true)
                    .SetContentIntent(pendingIntent);

        var notificationManager = NotificationManager.FromContext(this);

        notificationManager.Notify(0, notificationBuilder.Build());
    }
}
```

Sending messages from an application back end to Notification Hubs using C#

Using the various client libraries for Notification Hubs, you can send notifications from any application's back end to your registered devices. In the context of a .NET application, you would need to import the **Microsoft.Azure.NotificationHubs** NuGet package. Once imported, the library contains a **NotificationHubClient** class that can be used to send messages:

```
using Microsoft.Azure.NotificationHubs;

public class NotificationManager
{
    public async void SendNotifications()
    {
          NotificationHubClient hub = NotificationHubClient.CreateClientFrom-
ConnectionString("<your hub's DefaultFullSharedAccessSignature>", "<hub
name>");

          var toast = @"<toast><visual><binding template=""ToastText01""><-
text id=""1"">From contoso-admin: Hello World</text></binding></visual></
toast>";
          await hub.SendWindowsNativeNotificationAsync(toast);

          var alert = @"{"aps": {"alert": "From contoso-admin: Hello
World"}}";
          await hub.SendAppleNativeNotificationAsync(alert);

          var notif = @"{"data": {"message":"From contoso-admin: Hello
World"}}";
          await hub.SendGcmNativeNotificationAsync(notif);
    }
}
```

Typically, multiple requests were required to send a notification to each supported client platform. Azure Notification Hubs supports templates—with which you can specify how a specific device wants to receive notifications. This method simplifies sending cross-platform notifications.

```
var notification = new Dictionary\<string, string\> { { "message", "Hello
World"
} };

await Notifications.Instance.Hub.SendTemplateNotificationAsync(notification);
```

This code sends a notification to all platforms at the same time without you having to specify a native payload. Notification Hubs builds and delivers the correct payload to every device as specified in the registered templates.

Create and configure an event hub

Azure Event Hubs

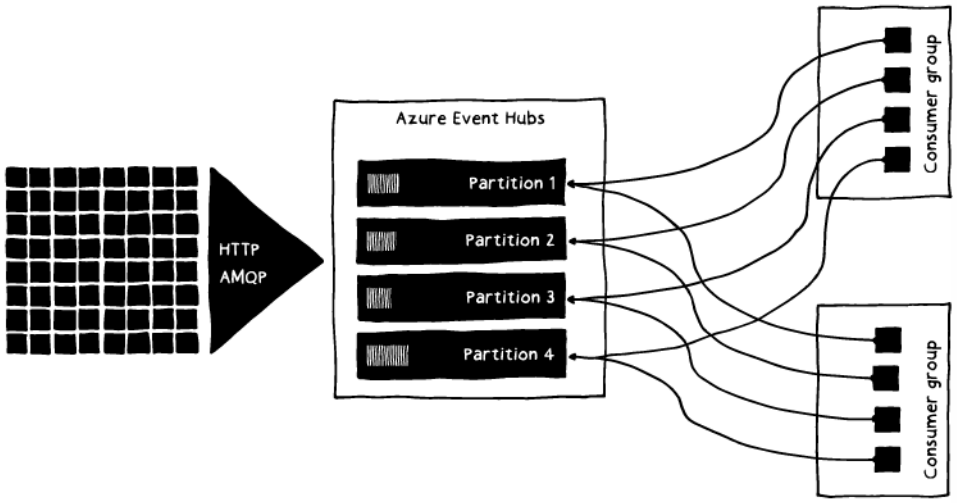
Data is valuable only when there is an easy way to process and get timely insights from data sources. Azure Event Hubs is a big data streaming platform and event ingestion service, capable of receiving and processing millions of events per second. Event Hubs can process and store events, data, or telemetry produced by distributed software and devices. Data sent to an event hub can be transformed and stored using any real-time analytics provider or batching/storage adapters. Event Hubs provides a distributed stream processing platform with low latency and seamless integration with data and analytics services inside and outside Azure to build a complete big data pipeline.

Event Hubs represents the "front door" for an event pipeline, often called an event ingestor in solution architectures. An event ingestor is 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 Hubs provides a unified streaming platform with a time retention buffer, decoupling the event producers from event consumers.

Event Hubs capabilities are built around high throughput and event processing scenarios. Event Hubs contains the following key components:

- **Event producers**: Any entities that send data to an event hub. Event publishers can publish events using HTTPS or Advanced Message Queuing Protocol (AMQP) 1.0 or Apache Kafka (1.0 and above).
- **Partitions**: Each consumer only reads a specific subset, or partition, of the message stream.
- **Consumer groups**: Views (state, position, or offset) of an entire event hub. Consumer groups enable multiple consuming applications to each have a separate view of the event stream and to each read the stream independently at its own pace and with its own offsets.
- **Throughput units**: Prepurchased units of capacity that control the throughput capacity of Event Hubs.
- **Event receivers**: Any entities that read event data from event hubs. All Event Hubs consumers connect via the AMQP 1.0 session, and events are delivered through the session as they become available.

The following figure shows the Event Hubs stream processing architecture:



Azure IoT Hub

Azure IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bidirectional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and encrypted communications between millions of IoT devices and a cloud-hosted solution back-end. You can connect virtually any device to IoT Hub.

IoT Hub supports communications both from the device to the cloud and from the cloud to the device. IoT Hub supports multiple messaging patterns, such as device-to-cloud telemetry, file upload from devices, and request-reply methods, to control your devices from the cloud. IoT Hub monitoring helps you maintain the health of your solution by tracking events such as device creation, device failures, and device connections.

The capabilities of IoT Hub help you build scalable, full-featured IoT solutions, such as managing industrial equipment used in manufacturing, tracking valuable assets in healthcare, and monitoring office building usage.

Comparing IoT Hub and Event Hubs

Azure provides services specifically developed for diverse types of connectivity and communication to help you connect your data to the power of the cloud. Both Azure IoT Hub and Azure Event Hubs are cloud services that can ingest large amounts of data and process or store that data for business insights. The two services are similar in that they both support the ingestion of data with low latency and high reliability, but they are designed for different purposes. IoT Hub was developed specifically to address the unique requirements of connecting IoT devices—at scale—to Azure Cloud Services, while Event Hubs was designed for big data streaming. This is why Microsoft recommends using Azure IoT Hub to connect IoT devices to Azure.

Azure IoT Hub is the cloud gateway that connects IoT devices to gather data to drive business insights and automation. In addition, IoT Hub includes

features that enrich the relationship between your devices and your back-end systems. Bidirectional communication capabilities mean that while you receive data from devices, you can also send commands and policies back to devices—for example, to update properties or invoke device management actions. This cloud-to-device connectivity also powers the important capability of delivering cloud intelligence to your edge devices with Azure IoT Edge. The unique device-level identity provided by IoT Hub helps better secure your IoT solution from potential attacks.

Azure Event Hubs is the big data streaming service of Azure. It is designed for high throughput data streaming scenarios where customers may send billions of requests per day. Event Hubs uses a partitioned consumer model to scale out your stream and is integrated into the big data and analytics services of Azure, including Azure Databricks, Stream Analytics, Azure Data Lake Storage (ADLS), and Azure HDInsight. With Event Hubs features like Capture and Auto-Inflate, this service is designed to support your big data apps and solutions. Additionally, IoT Hub leverages Event Hubs for its telemetry flow path, so your IoT solution also benefits from the tremendous power of Event Hubs.

To summarize, while both solutions are designed for data ingestion at a massive scale, only IoT Hub provides the rich IoT-specific capabilities that are designed for you to maximize the business value of connecting your IoT devices to Azure Cloud Services. If your IoT journey is just beginning, starting with IoT Hub to support your data ingestion scenarios will help assure that you have instant access to the full-featured IoT capabilities once your business and technical needs require them.

Create and configure a service bus

Azure Service Bus

Whether an application or service runs in the cloud or on-premises, it often needs to interact with other applications or services. Different situations call for different styles of communication. Sometimes, letting applications send and receive messages through a simple queue is the best solution. In other situations, an ordinary queue isn't enough; a queue with a publish-and-subscribe mechanism is better. In some cases, all that's needed is a connection between applications, and queues are not required.

To provide a broadly useful way to do this, Azure offers Azure Service Bus. Azure Service Bus provides all three options, enabling your applications to interact in several different ways. Service Bus is a multitenant cloud service, which means that the service is shared by multiple users.

A namespace is a scoping container for all messaging components. Multiple queues and topics can reside within a single namespace, and namespaces often serve as application containers. Each user, such as an application developer, creates a namespace, and then defines the communication mechanisms needed within that namespace. Within a namespace, you can use one or more instances of three different communication mechanisms, each of which connects applications in a different way. The three communication mechanisms are:

- **Queues**, which allow one-directional communication. Each queue acts as an intermediary (sometimes called a broker) that stores sent messages until they are received. Each message is received by a single recipient.
- **Topics**, which provide one-directional communication using subscriptions. A single topic can have multiple subscriptions. Like a queue, a topic acts as a broker, but each subscription can optionally use a filter to receive only messages that match specific criteria.
- **Relays**, which provide bidirectional communication. Unlike a queue or a topic, a relay doesn't store in-flight messages; it's not a broker. Instead, it just passes them on to the destination application.

When you create a queue, topic, or relay, you give it a name. Combined with whatever you called your namespace, this name creates a unique identifier for the object. Applications can provide this name to Service Bus and then use that queue, topic, or relay to communicate with each other.

Service Bus queues

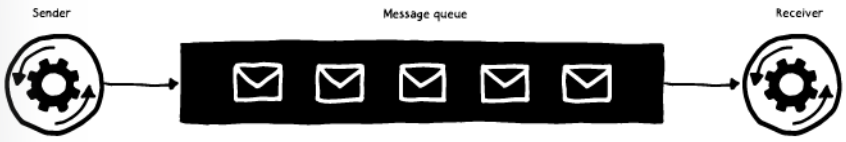
Azure Service Bus is a fully managed enterprise integration message broker. Service Bus is most commonly used to decouple applications and services from each other, and it’s a reliable and more-secure platform for asynchronous data and state transfer. Data is transferred between different applications and services using messages. A message is in binary format and can contain a JSON object, XML code, or just text.

Some common messaging scenarios are:

- **Messaging**: Transfer business data, such as sales or purchase orders, journals, or inventory movements.
- **Decoupling applications**: Improve the reliability and scalability of applications and services (the client and the service do not have to be online at the same time).
- **Topics and subscriptions**: Enable 1:n relationships between publishers and subscribers.
- **Message sessions**: Implement workflows that require message ordering or message deferral.

Queues

Messages are sent to and received from queues. Queues enable you to store messages until the receiving application is available to receive and process them. Messages in queues are ordered and time stamped on arrival. Once accepted, the message is held more safely in redundant storage. Messages are delivered in pull mode, which delivers messages on request.



Service Bus queues support a brokered messaging communication model. When using queues, components of a distributed application do not communicate directly with each other; instead, they exchange messages via a queue, which acts as an intermediary (broker). A message producer (sender) hands off a message to the queue and then continues its processing. Asynchronously, a message consumer (receiver) pulls the message from the queue and processes it. The producer does not have to wait for a reply from the consumer in order to continue to process and send further messages. Queues offer first in, first out (FIFO) message delivery to one or more competing consumers. That is, messages are typically received and processed by the receivers in the order in which they were added to the queue, and each message is received and processed by only one message consumer.

Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between the components of a distributed application running on-premises in different organizations or departments of an organization.

Using queues enables you to scale your applications more easily and enables more resiliency in your architecture.

Manipulating a Service Bus queue using Ruby

To use Service Bus, you will first need to download and use the Azure Ruby package, which includes a set of convenience libraries that communicate with the storage representational state transfer (REST) services. The easiest way to accomplish this is to install the gem and dependencies:

```
gem install azure
```

At the top of any Ruby file where you want to use the Azure Ruby libraries, you need to include the **azure** directive:

```
require "azure"
```

To create a connection to Service Bus using the client object, use the following code to set the values of the namespace, key name, key, signer, and host:

```
Azure.configure do |config|
    config.sb_namespace = '<your azure service bus namespace>'
    config.sb_sas_key_name = '<your azure service bus access keyname>'
    config.sb_sas_key = '<your azure service bus access key>'
end
signer = Azure::ServiceBus::Auth::SharedAccessSigner.new
sb_host = "https://#{Azure.sb_namespace}.servicebus.windows.net"
The Azure::ServiceBusService object enables you to work with queues. To
create a queue, use the create_queue() method. The following example cre-
ates a queue or prints out any errors:
azure_service_bus_service = Azure::ServiceBus::ServiceBusService.new(sb_
host, { signer: signer})
begin
    queue = azure_service_bus_service.create_queue("test-queue")
rescue
    puts $!
end
```

To send a message to a Service Bus queue, your application calls the **send_queue_message()** method on the **Azure::ServiceBusService** object. Messages sent to (and received from) Service Bus queues are **Azure::ServiceBus::BrokeredMessage** objects and have a set of standard properties (such as **label** and **time_to_live**); a dictionary that is used to hold custom, application-specific properties; and a body of arbitrary application data. An application can set the body of the message by passing a string value as the message, and any required standard properties are populated with default values.

The following example demonstrates how to send a test message to the queue named **test-queue** using **send_queue_message()**:

```
message = Azure::ServiceBus::BrokeredMessage.new("test queue message")

message.correlation_id = "test-correlation-id"
```

```
azure_service_bus_service.send_queue_message("test-queue", message)
```

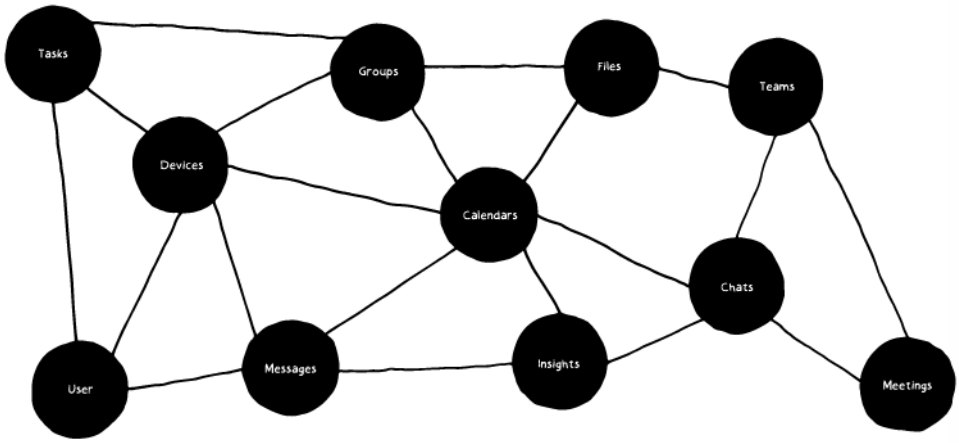
Configuring apps and services with Microsoft Graph

Microsoft Graph

Microsoft Graph is the gateway to data and intelligence in Microsoft 365. Microsoft Graph provides a unified programmability model that you can use to query and manipulate data in Microsoft Office 365, Microsoft Enterprise Mobility

- Security, and Windows 10. Microsoft Graph exposes APIs for:
- Azure Active Directory
- Office 365 services: Microsoft SharePoint, Microsoft OneDrive, Microsoft Outlook, Microsoft Exchange, Microsoft Teams, Microsoft OneNote, Microsoft Planner, and Microsoft Excel
- Enterprise Mobility and Security services: Microsoft Identity Manager, Microsoft Intune, Microsoft Advanced Threat Analytics, and Azure Advanced Threat Protection.
- Windows 10 services: activities and devices

Microsoft Graph connects all the resources across these services using relationships. For example, a user can be connected to a group through a **memberOf** relationship and to another user through a manager relationship. Your app can traverse these relationships to access these connected resources and perform actions on them through the API.



You can also get valuable insights and intelligence about the data from Microsoft Graph. For example, you can get the popular files trending around a particular user or get the most relevant people around a user.

REST API in Microsoft Graph

Microsoft Graph contains a REST API with endpoints for each resource that you can query on the graph. The table below lists example endpoints for common tasks you may want to perform with the REST API:

Operation	URL
GET my profile	**https://graph.microsoft.com/v1.0/me** *(https://developer.microsoft.com/graph/graph-explorer/?request=me&version=v1.0)*
GET my files	**https://graph.microsoft.com/v1.0/me/drive/root/children** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fdrive%2Froot%2Fchildren&version=v1.0)*
GET my photo	**https://graph.microsoft.com/v1.0/me/photo/$value** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fphoto%2F%24value&version=v1.0)*
GET my mail	**https://graph.microsoft.com/v1.0/me/messages** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fmessages&version=v1.0)*
GET my high importance email	**https://graph.microsoft.com/v1.0/me/messages?$filter=importance%20eq%20'high'** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fmessages%3F%24filter%3Dimportance%2520eq%2520%27high%27&version=v1.0)*
GET my calendar events	**https://graph.microsoft.com/v1.0/me/events** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fevents&version=v1.0)*
GET my manager	**https://graph.microsoft.com/v1.0/me/manager** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fmanager&version=v1.0)*
GET the last user to modify the file foo.txt	**https://graph.microsoft.com/v1.0/me/drive/root/children/foo.txt/lastModifiedByUser** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fdrive%2Froot%2Fchildren%2Ffoo.txt%2FlastModifiedByUser&version=v1.0)*
GET the Office 365 groups I'm member of	**https://graph.microsoft.com/v1.0/me/memberOf/$/microsoft.graph.group?$filter=groupTypes/any(a:a%20eq%20'unified')** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2FmemberOf%2F%24%2Fmicrosoft.graph.group%3F%24filter%3DgroupTypes%2Fany(a%3Aa%2520eq%2520%27unified%27)&version=v1.0)*
GET users in my organization	**https://graph.microsoft.com/v1.0/users** *(https://developer.microsoft.com/graph/graph-explorer/?request=users&version=v1.0)*
GET groups in my organization	**https://graph.microsoft.com/v1.0/groups** *(https://developer.microsoft.com/graph/graph-explorer/?request=groups&version=v1.0)*

Operation	URL
GET people related to me	**https://graph.microsoft.com/v1.0/me/people** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fpeople&version=beta)*
GET items trending around me	**https://graph.microsoft.com/beta/me/insights/trending** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Finsights%2Ftrending&version=beta)*
GET my notes	**https://graph.microsoft.com/v1.0/me/onenote/notebooks** *(https://developer.microsoft.com/graph/graph-explorer/?request=me%2Fonenote%2Fnotebooks&version=beta)*

Review Questions

Module 2 Review Questions

SendGrid

You are designing a solution that allows users to purchase merchandise from your organization.

Customers must automatically receive an email notification at the end of any transaction.

Which Azure service should you recommend?

Suggested Answer ↓

SendGrid (https://sendgrid.com) is a third-party, cloud-based email messaging service that has a deep level of integration with the Azure platform and portal. SendGrid provides transactional email delivery, scalability based on email volume, and real-time analytics for the sent messages. SendGrid also has a flexible API to enable custom integration scenarios.

Event-driven architecture

You are designing a solution that ingests and processes large volumes of data in real time.

What type of architecture should you consider? Which Azure service provides support for this architecture?

Suggested Answer ↓

In an event-driven architecture, a system delivers event data to consumer applications and services. These applications and series can in turn respond to these events quickly.Azure Event Grid allows you to easily build applications with event-based architectures.

Event-driven architecture

You are designing a solution that allows users to purchase merchandise for a grocery store. When customers place items into a shopping cart, the solution must send the customer a notification to their mobile device.

Which Azure service should you recommend?

Suggested Answer ↓

Azure Notification Hubs allows you to send push notification messages to mobile devices. This service is highly scalable and is compatible with most mobile devices and back-end systems.

Module 3 Module Developing for Asynchronous Processing

Implement parallelism, multithreading, and processing

Asynchronous processing

Many personal computers and workstations have several CPU cores that enable the simultaneous execution of multiple threads. As client devices increase in performance and server computers opt for horizontal over vertical scaling, it's becoming increasingly common for a local client device to have more computing power and threads than the server it is currently accessing. Shortly, computers are expected to have even more cores, and this divide will continue to grow. It is more critical now than ever to take advantage of parallelism in client devices, server-side computers, and cloud-managed computing.

Parallelism

In the past, parallelization required the low-level manipulation of threads and locks. Writing parallelized code was an esoteric skill that could take years of experience to master. Later versions of the Microsoft .NET Framework introduced the Task Parallel Library (TPL) to simplify the development of parallelized code. With the TPL, developers can focus on writing efficient, fine-grained code, while the specific .NET runtime manages the threads in a manner that can efficiently scale in the cloud.

Task Parallel Library (TPL)

The TPL is a set of public types and APIs in the **System.Threading** namespace. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL dynamically scales the degree of concurrency to most efficiently use all the processors that are available. Also, the TPL handles the partitioning of the work, the scheduling of threads in the thread pool, cancellation support, state management, and other low-level details.

To get started with the TPL, create a **Task** instance that has a delegate parameter. The delegate contains the code that is encapsulated and executed in the context of the task. The actual **Task** instance contains metadata about the delegate, including but not limited to:

- Whether the delegate has started.
- Whether the delegate has completed executing.
- The resulting value of the delegate.

After you have a **Task** instance, invoke the **Task.Run** method to start the task. The **Task** class contains a **Wait** method that can be invoked to block execution until the asynchronous task has finished executing. The **Result** property of the **Task** class contains the resulting value of executing the delegate. A typical usage of the TPL looks like this:

```
Task<string> asyncOperation = new Task<string>(() =>
    {
          string greeting = $"You are running {Environment.OSVersion}";
          return greeting;
    }
);

asyncOperation.Start();

asyncOperation.Wait();

string message = asyncOperation.Result;
```

Note: The TPL also contains convenient helper methods, such as **Task.Run** and **TaskFactory.StartNew**, to automatically start a newly created task.

Refactoring TPL code

You can simplify the use of the TPL even further by using the **async** modifier and the **await** keyword in a .NET method. Instead of directly using the **Task** methods, this keyword instructs the compiler to handle the tracking of the asynchronous methods. The keyword and its abstraction makes your code more efficient and easier to read. To get started with this keyword, first define the delegate as a separate method by using the **async** modifier:

```
public async Task<string> AsyncOperation()
{
    string greeting = $"You are running {Environment.OSVersion}";
    return greeting;
}
```

You can then define another method to use the asynchronous method. Within this method, use the await keyword to asynchronously call the first method:

```
public async void DoSomething()
{
    string message = await AsyncOperation();
}
```

Implement Azure Functions and Azure Logic Apps

Serverless computing

Serverless computing promises agility and power in building the next generation of solutions. You can use services such as Azure Functions and Azure Logic Apps to build these solutions. All of these services are useful for "gluing" together disparate systems. They can all define input, actions, conditions, and output. You can run each of them on a schedule or via a trigger. However, each service has unique advantages, and comparing them is not a question of "Which service is the best?" but one of "Which service is best suited for this situation?" Often, a combination of these services is the best way to rapidly build a scalable, full-featured integration solution.

Azure Functions

Azure Functions is a solution for running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development languages of choice, such as C#, F#, Node.js, Java, or PHP. Functions is an excellent solution for processing data, integrating systems, working with the Internet of Things (IoT), and building simple APIs and microservices.

Azure Functions integrates with various Azure and third-party services. These services can trigger your function and start execution, or they can serve as input and output for your code. Azure Functions supports the following service integrations:

- Azure Cosmos DB
- Azure Event Hubs
- Azure Event Grid
- Azure Service Bus (queues and topics)
- Azure Storage (blobs, queues, and tables)
- GitHub (webhooks)
- On-premises applications (by using Service Bus)
- Twilio (SMS messages)

Example Azure function for .NET

Azure Functions supports the compiled C# and C# script programming languages. The Azure WebJobs software development kit (SDK) serves as the foundation for the C# script experience for Azure Functions. First, data flows into your C# function via method parameters. The **function.json** file specifies any argument names that also exist as C# parameters in the script. An example **function.json** file that binds the function instance to an Azure storage queue looks like this:

```
{
"disabled": false,
"bindings": [
        {
            "type": "queueTrigger",
```

```
            "direction": "in",
            "name": "message",
            "queueName": "announcementqueue",
            "connection":"StorageConnectionString"
        }
    ]
}
```

The C# script for the corresponding function uses the **name** property of the "in" binding as a method parameter:

public static void Run(string message, System.TraceWriter log)

```
public static void Run(string message, System.TraceWriter log)
{
    log.Info($"New message: {message}");
}
```

The preceding example function immediately prints the message received from the queue to the function's log.

Logic Apps

Logic Apps helps you build, schedule, and automate processes as workflows so you can integrate apps, data, systems, and services across enterprises or organizations. Logic Apps simplifies how you design and create scalable solutions for app integration, data integration, system integration, enterprise application integration (EAI), and business-to-business (B2B) communication—whether in the cloud, on-premises, or both.

For example, here are just a few workflows that you can automate with logic apps:

- Process and route orders across on-premises systems and cloud services.
- Move uploaded files from an FTP server to Azure Storage.
- Monitor tweets for a specific subject, analyze the sentiment and create alerts or tasks for items that need review.

Every logic app workflow starts with a trigger, which fires when a specific event happens or when newly available data meets specific criteria. Many triggers include essential scheduling capabilities so you can specify how regularly your workloads will run. For more custom scheduling scenarios, start your workflows with the Schedule trigger.

Each time the trigger fires, the Logic Apps engine creates a logic app instance that runs the workflow's actions. These actions can include data conversions and flow controls, such as conditional statements, switch statements, loops, and branching. For example, a logic app might start with a Dynamics 365 trigger that has the built-in criterion "When a record is updated." If the trigger detects an event that matches this criterion, the trigger will fire and run the workflow's actions. These actions might include XML transformation, data updates, decision branching, and email notifications.

You can visually build your logic apps by using the Logic Apps designer, which is available in the Azure portal through your browser and in Visual Studio. You can also use Azure PowerShell commands and Azure Resource Manager templates for select tasks. For more-custom logic apps, you can create or edit logic app definitions in JavaScript Object Notation (JSON) by working in code view mode. Here is an example of a JSON template for a logic app:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2016-06-01/
Microsoft.Logic.json",
    "contentVersion": "1.0.0.0",
    "parameters": {
          "uri": {
                "type": "string"

          }
    },
    "triggers": {
          "request": {
                "type": "request",
          "kind": "http"

          }
    },
    "actions": {
          "readData": {
                "type": "Http",
            "inputs": {
                      "method": "GET",
                "uri": "@parameters('uri')"

                }
          }
    },
    "outputs": { }
}
```

Create an Azure function by using a C# script

1. Navigate to **https://portal.azure.com**.
2. If you're not already signed in, sign in with the credentials you typically use for your Azure subscription.
3. In the navigation pane on the left side, select **Create a resource**.
4. In the **New** blade, in the **Search the Marketplace** box, enter **function app**, and then press Enter.
5. In the search results, select the **Function App** template.
6. In the **Function App** blade, select **Create**.
7. In the form that displays, perform the following steps:
    1. In the **App name** dialog box, create a unique name for your function.
    2. In the **Resource Group** section, select **Create new**, and then enter the value **FUNCDEMO**.
    3. In the **OS** group, select **Windows**.
    4. In the **Hosting Plan** list, select **Consumption Plan**.
    5. In the **Location** list, select the region that is closest to your location.

6. In the **Storage** section, select **Create new**, and leave the name set to the recommended value that is displayed.
7. In the **Application Insights** section, select **Off**.
8. Select **Create** to create your function app.

8. Wait for the function app to finish being created. You will see a notification in the portal indicating that the resource was successfully created.
9. In the navigation pane on the left side, select **All services**.
10. Scroll down, and then locate and select the **App Services** option.
11. In the **App Services** list, select the function app that you created earlier.
12. On the left side of the **Function Apps** blade, locate and select the **Functions** link.
13. In the **Functions** section, at the top of the section, select **New function**.
14. In the template list, select the **HTTP trigger** template.
15. In the **HTTP Trigger** dialog box, perform the following steps:
    1. In the **Language** list, select **C#**.
    2. In the **Name** box, enter **Echo**.
    3. Select **Create**.
16. Observe the default C# code in the following function:

```
using System.Net;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,
TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");

    // parse query parameter
    string name = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
        .Value;

    if (name == null)
    {
        // Get request body
        dynamic data = await req.Content.ReadAsAsync<object>();
        name = data?.name;
    }

    return name == null
        ? req.CreateResponse(HttpStatusCode.BadRequest, "Please pass a name
on the query string or in the request body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
}
```

17. Replace the existing function code with the following simplified script:

```
public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,
TraceWriter log)
{
    log.Info("C# HTTP trigger executed");
    return req.CreateResponse(
        await req.Content.ReadAsAsync<object>()
    );
}
```

18. **Note**: This script will echo the body of a request back to the calling client.
19. Select **Save and run** to test the function.

- You should quickly notice that the example request body is echoed back as a response.

1. Locate the request body field in the **Test** section:

```
{
    "name": "Azure"
}
```

2. Replace the request body field's value with the following JSON object:

```
{
    "sizeSystem": "US",
    "sizeType": "regular",
    "targetCountry": "US",
    "taxes": [
        {
            "country": "US",
            "rate": "9.9",
            "region": "CA",
            "taxShip": "True"
        }
    ],
    "title": "Cute Toddler Sundress"
}
```

3. Select **Save and run** again to verify that the new JSON object is correctly echoed.
4. In the navigation pane on the left side of the Azure portal, select **All services**.
5. In the **All services** blade, select **Resource groups**.
6. In the **Resource groups** blade, select the **FUNCDEMO** resource group.
7. At the top of the **FUNCDEMO** blade, select **Delete resource group**.
8. In the deletion confirmation dialog box, enter the name **FUNCDEMO**, and then select **Delete**.
9. Wait for the resource group to be deleted.

Implement interfaces for storage or data access

Asynchronous interfaces for external services

Blocking the calling thread during I/O can reduce performance and affect vertical scalability. You can see this antipattern when your application attempts to access data in local storage via either Azure Storage or another Azure data service. After the application blocks the calling thread, the thread enters a wait state during which other operations can't use it, and it's effectively wasting valuable metered resources. Because cloud-based services can experience latency or lag, any type of I/O-based wait might be disastrous, and even a single I/O block can block an entire call chain—crippling a useful thread in a computing instance.

To fix this problem, implement asynchronous interfaces for all of your data tier code that accesses any storage or data resource in Azure. Ideally, pair these asynchronous interfaces with implementations that invoke the asynchronous version of an SDK or API method that you want to invoke.

Asynchronous interfaces for Azure services in ASP.NET MVC

In a common example, you have a Microsoft ASP.NET Model - View - Controller (MVC) application that accesses data in an Azure SQL Database instance using Entity Framework Core. In the simplest implementation, you create a **DbContext** implementation that is used directly in the controller action:

```
public class ItemsContext: DbContext
{
    public DbSet<Item> Items { get; set; }
}

public class Item
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class ItemsController : Controller
{
    public string Get(int id)
    {
           ItemsContext context = new ItemsContext();
           Item item = context.Items.Find(id);
           return item.Name;
    }
}
```

This is less than ideal for a few reasons.

First, your controller action strongly couples with the Entity Framework implementation. If you chose to use a different object-relational mapper in the future, you would need to rewrite a large quantity of code to implement the object-relational mapper, because Entity Framework–specific code exists directly in

your web application. Things might be even worse if you chose to use a different database service altogether.

Second, you might observe I/O blocking while your code uses the synchronous version of the Entity Framework methods in the **System.Data.Entity** namespace.

To solve both of these issues, create a data layer for the sample application by creating an interface with asynchronous method signatures. This interface will be the only thing coupled to your MVC web application code—making it easier to switch your application code if you change your object-relational mapper or database service. The interface will also enforce the use of asynchronous methods in your data layer by requiring asynchronous signatures on your classes. ASP.NET MVC also has built-in dependency injection, which will make it easier to implement this pattern. Here is a similar example that uses an interface and a data layer class:

```
public class ItemsContext: DbContext
{
    public DbSet<Item> Items { get; set; }
}

public class Item
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public interface IDataLayer
{
    Task<string> GetNameForIdAsync(int id);
}

public class EntityFrameworkDataLayer : IDataLayer
{
    public async Task<string> GetNameForIdAsync(int id)
    {
          ItemsContext context = new ItemsContext();
          Item item = await context.Items.FindAsync(id);
          return item.Name;
    }
}

public class ItemsController : Controller
{
    public async Task<string> Get(IDataLayer dataLayer, int id)
    {
          return await dataLayer.GetNameForIdAsync(id);
    }
}
```

Implement appropriate asynchronous computing models

Asynchronous computing

Modern apps make extensive use of file and networking I/O. I/O APIs traditionally block the calling thread by default, resulting in what the user perceives as an application that has stopped responding. In the server-side world, blocking an application on I/O can have devastating consequences for other users of the same server application.

To solve this, many developers have implemented asynchronous code, which:

- Handles more server requests by yielding threads while waiting for I/O requests to return.
- Enables UIs to be more responsive by yielding threads to UI interaction while waiting for I/O requests and by transitioning long-running work to other CPU cores.

In cloud-based applications, most SDKs, APIs, and code are asynchronous by default. Asynchronous SDKs exist because most cloud services are metered (billed) based on usage. Synchronous code increases usage, blocks other users, and ultimately results in increased billing.

Asynchronous code in .NET

In .NET, we recommend authoring asynchronous code by using the Task-based Asynchronous Pattern. This pattern leverages the **Task** types in the TPL to represent arbitrary asynchronous operations.

In the past, you might have written a cloud-hosted API by using ASP.NET MVC and a controller action that accesses your data tier, like this:

```
[Route("/api/users/")]
public class UserController : Controller
{
    [HttpGet("{id:int}")]
    public string GetName(int id)
    {
        return DataTier.GetNameForUniqueId(id);
    }
}
```

Although this code technically works, you might create a blocking scenario if your data tier needs to access I/O or a third-party service. First, use the asynchronous version of your data tier method. Second, update the MVC controller action to use an asynchronous signature. The .NET and ASP.NET runtimes will automatically handle the parallel asynchronous processing of MVC controller actions when you use an asynchronous signature. The updated controller action looks like this:

```
[Route("/api/users/")]
public class UserController : Controller
{
    [HttpGet("{id:int}")]
    public async Task<string> GetName(int id)
    {
        return await DataTier.GetNameForUniqueIdAsync(id);
    }
```

```
}
```

Review Questions

Module 3 Review Questions

Logic Apps

You are designing a solution to handle order processing from an online e-Commerce site.

The solution must route messages to manufacturing, create orders in a sales system, and notify your warehouse that an order is ready for processing.

You need to create the workflow modules to support the solution.

What should you use?

Suggested Answer ↓

Suggested Answer:

Logic Apps helps you build, schedule, and automate processes as workflows so you can integrate apps, data, systems, and services across enterprises or organizations. Logic Apps simplifies how you design and create scalable solutions for app integration, data integration, system integration, enterprise application integration (EAI), and business-to-business (B2B) communication—whether in the cloud, on-premises, or both.

Asynchronous code in .NET

You are designing a solution to handle order processing from an online e-Commerce site.

The solution must be able to handle multiple requests simultaneously without affecting the experience of the users of the platform.

How can you ensure that a user request does not affect the experience of other users? Which design pattern should be used to develop the code? Which tools are available?

Suggested Answer ↓

You should implement asynchronous code for the solution. Instead of waiting for processes to finish, the solution makes a request and then performs other activities. When the requested action completes, the code is alerted and can respond. This pattern allows solutions to handle more requests while remaining responsive.

Computing patterns

You are designing a solution to handle order processing from an online e-Commerce site.

The solution must scale out as demand grows.

What type of patterns can you use? Which is most appropriate for the scenario?

Suggested Answer ↓

On and off, growing fast, unpredictable bursting, and predictable bursting are four common computing design patterns. For this scenario, unpredictable bursting is most appropriate. Solutions that use this pattern are designed to respond to unexpected spikes in activity. Resources are automatically added to

handle increased workloads and then released when no longer necessary.

Module 4 Module Developing for Autoscaling

Implement autoscaling rules and patterns

Computing patterns

Computing Patterns

In distributed application scenarios, you are often advised to scale out your application horizontally by adding multiple, small instances of your application to the cloud. With horizontal scaling, you can serve more client devices and ensure high availability and resiliency against any specific fault.

Unfortunately, application workloads are unpredictable. To illustrate this, here are four of the most common computing patterns you will see for web applications hosted in the cloud:

Pattern	Graphic	Description
On and off		Workloads occur occasionally (for example, batch processing). Over-provisioned capacity is wasted. The time to market can be cumbersome.
Growing fast		A successful service tends to grow on a curve. Keeping up with growth is a management challenge. The IT team may not be able to provision hardware fast enough to keep pace with growth.
Unpredictable bursting		Bursts correlate with unexpected or unplanned peaks in demand. Sudden spikes impact the user experience across the board. Over-provisioning will lead to tremendous wasted capacity.

Pattern	Graphic	Description
Predictable bursting		Bursts can be mapped to specific trends that are planned and well known. Resources will need to be provisioned and de-provisioned to match the peaks and valleys of the usage curve. The complexity of the infrastructure can be challenging for the IT team.

You can easily overestimate or underestimate the number of instances required to provide the best user experience. If you overestimate, you can end up paying for unnecessary compute resources. If you underestimate, you can provide a poor user experience. Ideally, you want your web platform to use the extra instances only when it is necessary and to shut down the same instances when they are no longer needed.

Scale and auto-scale

To provide redundancy and improved performance, applications are typically distributed across multiple instances. Customers may access your application through a load balancer that distributes requests to one of the application instances. If you need to perform maintenance or update an application instance, your customers must be distributed to another available application instance. To keep up with additional customer demand, you may need to increase the number of application instances that run your application. To save money during times of reduced customer utilization, you may need to decrease the number of application instances.

A primary advantage of the cloud is elastic scaling—the ability to use as much capacity as you need, scaling out as load increases and scaling in when the extra capacity is not needed. In the context of Microsoft Azure, many services provide the capability to both manually and automatically scale to closely match demand. These options include infrastructure services such as Azure Virtual Machine Scale Sets, application services such as Azure App Service, and even database services such as Azure Cosmos DB.

Auto-scale refers to the capability of many of these services to monitor the application instances and automatically scale appropriately to handle the current usage of the application. Using auto-scale, your cloud service can scale out and in to exactly match the amount of instances needed for your specific computing pattern.

Auto-scale metrics

App Service apps hosted in Basic, Standard, or Premium App Service plans support autoscale. Autoscale allows you to configure rules that monitor the App Service plan metrics. Rules can increase or decrease the instance count, providing additional resources as needed. Rules can also help you save money when the application is over-provisioned.

Auto-scale settings help ensure that you have the right amount of resources running to handle the fluctuating load of your application. You can configure auto-scale settings to be either triggered based on metrics that indicate the load or performance or triggered at a scheduled date and time.

Below is the list of auto-scale metrics that are available to an App Service plan instance:

Metric	Metric identifier	Description
CPU	CpuPercentage	The average amount of CPU time used across all instances of the plan
Memory	MemoryPercentage	The average amount of memory used across all instances of the plan
Data in	BytesReceived	The average incoming bandwidth used across all instances of the plan
Data out	BytesSent	The average outgoing bandwidth used across all instances of the plan
HTTP queue	HttpQueueLength	The average number of both read and write requests that were queued on storage. A high disk queue length is an indication of an application that might be slowing down due to excessive disk I/O.
Disk queue	DiskQueueLength	The average number of HTTP requests that had to sit in the queue before being fulfilled. A high or increasing HTTP queue length is a symptom of a plan under a heavy load.

Alerts and auto-scale

In addition to being used for automatic scale, metrics for an App Service plan can be used to trigger alerts from Azure. An alert triggers when the value of a specified metric crosses a threshold that you assign in either direction. That is, it triggers both when the condition is first met and when that condition is no longer being met.

You can configure a metric alert to do the following when it triggers:

- Send email notifications to the service administrator and co-administrators
- Send email to additional email addresses that you specify
- Call a webhook
- Start the execution of an Azure runbook

Implement code that addresses singleton application instances

Querying resources using Azure CLI

The Azure Command-Line Interface (Azure CLI) is the Microsoft cross-platform command-line experience for managing Azure resources. You can use it in your browser with Azure Cloud Shell or install it on macOS, Linux, or Windows and run it from the command line. Azure CLI is optimized for managing and administering Azure resources from the command line and for building automation scripts that work against the Azure Resource Manager.

The Azure CLI uses the **–query** argument to execute a **JMESPath** query on the results of commands. **JMESPath** is a query language for JavaScript Object Notation (JSON) that gives you the ability to select and present data from Azure CLI output. These queries are executed on the JSON output before they perform any other display formatting. The **–query** argument is supported by all commands in the Azure CLI.

Many CLI commands will return more than one value. These commands always return a JSON array instead of a JSON document. Arrays can have their elements accessed by index, but there's never an order guarantee from the Azure CLI. To make the arrays easier to query, we can flatten them using the JMESPath **[]** operator.

In the following example, we use the **az vm list** command to query for a list of virtual machine (VM) instances:

```
az vm list
```

The query will return an array of large JSON objects for each VM in your subscription:

```
[
    {
          "availabilitySet": null,
          "diagnosticsProfile": null,
          "hardwareProfile": {
                "vmSize": "Standard_B1s"
          },
          "id": "/subscriptions/9103844d-1370-4716-b02b-69ce936865c6/
resourceGroups/VM/providers/Microsoft.Compute/virtualMachines/simple",
          "identity": null,
          "instanceView": null,
          "licenseType": null,
          "location": "eastus",
          "name": "simple",
          "networkProfile": {
                "networkInterfaces": [{
                      "id": "/subscriptions/9103844d-1370-4716-b02b-
69ce936865c6/resourceGroups/VM/providers/Microsoft.Network/networkInterfac-
es/simple159",
                      "primary": null,
                      "resourceGroup": "VM"
                }]
          },
```

```
            "osProfile": {
                  "adminPassword": null,
                  "adminUsername": "simple",
                  "computerName": "simple",
                  "customData": null,
                  "linuxConfiguration": {
                        "disablePasswordAuthentication": false,
                        "ssh": null
                  },
                  "secrets": [],
                  "windowsConfiguration": null
            },
            "plan": null,
            "provisioningState": "Creating",
            "resourceGroup": "VM",
            "resources": null,
            "storageProfile": {
                  "dataDisks": [],
                  "imageReference": {
                        "id": null,
                        "offer": "UbuntuServer",
                        "publisher": "Canonical",
                        "sku": "17.10",
                        "version": "latest"
                  },
                  "osDisk": {
                        "caching": "ReadWrite",
                        "createOption": "FromImage",
                        "diskSizeGb": 30,
                        "encryptionSettings": null,
                        "image": null,
                        "managedDisk": {
                              "id": "/subscriptions/9103844d-1370-4716-b02b-
69ce936865c6/resourceGroups/VM/providers/Microsoft.Compute/disks/simple_Os-
Disk_1_4da948f5ef1a4232ad2f632077326d0a",
                              "resourceGroup": "VM",
                              "storageAccountType": "Premium_LRS"
                        },
                        "name": "simple_OsDisk_1_4da948f5ef1a4232ad2f-
632077326d0a",
                        "osType": "Linux",
                        "vhd": null,
                        "writeAcceleratorEnabled": null
                  }
            },
            "tags": null,
            "type": "Microsoft.Compute/virtualMachines",
            "vmId": "6aed2e80-64b2-401b-a8a0-b82ac8a6ed5c",
            "zones": null
      },
      {
```

```
            ...
    }
]
```

Using the **–query** argument, we can specify project-specific fields to make the JSON object more useful and easier to read. This is useful if you are deserializing the JSON object into a specific type in your code:

```
az vm list --query '[].{name:name, image:storageProfile.imageReference.
offer}'
```

```
[
    {
            "image": "UbuntuServer",
            "name": "linuxvm"
    },
    {
            "image": "WindowsServer",
            "name": "winvm"
    }
]
```

Using the **[]** operator, you can create queries that filter your result set by comparing the values of various JSON properties:

```
az vm list --query "[?starts_with(storageProfile.imageReference.offer, 'Win-
dowsServer')]"
```

You can even combine filtering and projection to create custom queries that only return the resources you need and project only the fields that are useful to your application:

```
az vm list --query "[?starts_with(storageProfile.imageReference.offer, 'Ubun-
tu')].{name:name, id:vmId}"
```

```
[
    {
            "name": "linuxvm",
            "id": "6aed2e80-64b2-401b-a8a0-b82ac8a6ed5c"
    }
]
```

Querying resources using the fluent Azure SDK

In a manner similar to how you use the Azure CLI, you can use the Azure SDK to query resources in your subscription. The SDK may be a better option if you intend to write code to find connection information for a specific application instance. For example, you may need to write code to get the IP address of a specific VM in your subscription.

Connecting using the fluent Azure SDK

To use the APIs in the Azure management libraries for Microsoft .NET, as the first step, you need to create an authenticated client. The Azure SDK requires that you invoke the **Azure.Authenticate** static method to return an object that can fluently query resources and access their metadata. The **Authenticate** method requires a parameter that specifies an authorization file:

```
Azure azure = Azure.Authenticate("azure.auth").WithDefaultSubscription();
```

The authentication file, referenced as **azure.auth** above, contains information necessary to access your subscription using a service principal. The authorization file will look similar to the format below:

```
{
    "clientId": "b52dd125-9272-4b21-9862-0be667bdf6dc",
    "clientSecret": "ebc6e170-72b2-4b6f-9de2-99410964d2d0",
    "subscriptionId": "ffa52f27-be12-4cad-b1ea-c2c241b6cceb",
    "tenantId": "72f988bf-86f1-41af-91ab-2d7cd011db47",
    "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",
    "resourceManagerEndpointUrl": "https://management.azure.com/",
    "activeDirectoryGraphResourceId": "https://graph.windows.net/",
    "sqlManagementEndpointUrl": "https://management.core.windows.
net:8443/",
    "galleryEndpointUrl": "https://gallery.azure.com/",
    "managementEndpointUrl": "https://management.core.windows.net/"
}
```

If you do not already have a service principal, you can generate a service principal and this file using the Azure CLI:

```
az ad sp create-for-rbac --sdk-auth > azure.auth
```

Listing virtual machines using the fluent Azure SDK

Once you have a variable of type **IAzure**, you can access various resources by using properties of the **IAzure** interface. For example, you can access VMs using the **VirtualMachines** property in the manner displayed below:

```
azure.VirtualMachines
```

The properties have both synchronous and asynchronous versions of methods to perform actions such as **Create**, **Delete**, **List**, and **Get**. If we wanted to get a list of VMs asynchronously, we could use the **ListAsync** method:

```
var vms = await azure.VirtualMachines.ListAsync();

foreach(var vm in vms)
{
    Console.WriteLine(vm.Name);
}
```

You can also use any language-integrated query mechanism, like language-integrated query (LINQ) in C#, to filter your VM list to a specific subset of VMs that match a filter criteria:

```
var allvms = await azure.VirtualMachines.ListAsync();

IVirtualMachine targetvm = allvms.Where(vm => vm.Name == "simple").Single-
OrDefault();

Console.WriteLine(targetvm?.Id);
```

Gathering virtual machine metadata to determine the IP address

Now that we can filter to a specific VM, we can access various properties of the **IVirtualMachine** interface and other related interfaces to get that resource's IP address.

To start, the **IVirtualMachine.GetPrimaryNetworkInterface** method implementation will return the network adapter that we need to access the VM:

```
INetworkInterface targetnic = targetvm.GetPrimaryNetworkInterface();
```

The **INetworkInterface** interface has a property named **PrimaryIPConfiguration** that will get the configuration of the primary IP address for the current network adapter:

```
INicIPConfiguration targetipconfig = targetnic.PrimaryIPConfiguration;
```

The **INicIPConfiguration** interface has a method named **GetPublicIPAddress** that will get the IP address resource that is public and associated with the current specified configuration:

```
IPublicIPAddress targetipaddress = targetipconfig.GetPublicIPAddress();
```

Finally, the **IPublicIPAddress** interface has a property named **IPAddress** that contains the current IP address as a string value:

```
Console.WriteLine($"IP Address:\t{targetipaddress.IPAddress}");
```

Your application can now use this specific IP address to communicate directly with the intended compute instance.

Implement code that addresses a transient state

Transient errors

An application that communicates with elements running in the cloud has to be sensitive to the transient faults that can occur in this environment. Faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that occur when a service is busy.

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay, it's likely to be successful. For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay, it might succeed.

Handling transient errors

In the cloud, transient faults aren't uncommon, and an application should be designed to handle them elegantly and transparently. This minimizes the effects faults can have on the business tasks the application is performing.

If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

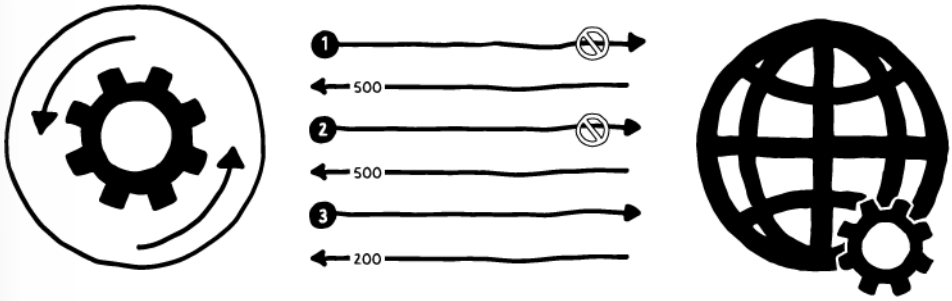
- **Cancel**: If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
- **Retry**: If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances, such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately, because the same failure is unlikely to be repeated, and the request will probably be successful.
- **Retry after a delay**: If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period of time while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable amount of time before retrying the request.

For the more common transient failures, the period between retries should be chosen to spread requests from multiple instances of the application as evenly as possible. This reduces the chance of a busy service continuing to be overloaded. If many instances of an application are continually overwhelming a service with retry requests, it'll take the service longer to recover.

If the request still fails, the application can wait and make another attempt. If necessary, this process can be repeated with increasing delays between retry attempts, until some maximum number of requests have been attempted. The delay can be increased incrementally or exponentially depending on the type of failure and the probability that it'll be corrected during this time.

Retrying after a transient error

The following diagram illustrates invoking an operation in a hosted service using this pattern. If the request is unsuccessful after a predefined number of attempts, the application should treat the fault as an exception and handle it accordingly.



1. The application invokes an operation on a hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2. The application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3. The application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

The application should wrap all attempts to access a remote service in code that implements a retry policy matching one of the strategies listed above. Requests sent to different services can be subject to different policies. Some vendors provide libraries that implement retry policies, where the application can specify the maximum number of retries, the amount of time between retry attempts, and other parameters.

An application should log the details of faults and failing operations. This information is useful to operators. If a service is frequently unavailable or busy, it's often because the service has exhausted its resources. You can reduce the frequency of these faults by scaling out the service. For example, if a database service is continually overloaded, it might be beneficial to partition the database and spread the load across multiple servers.

Handling transient errors in code

This example in C# illustrates an implementation of this pattern. The **OperationWithBasicRetryAsync** method, shown below, invokes an external service asynchronously through the **TransientOperationAsync** method. The details of the **TransientOperationAsync** method will be specific to the service and are omitted from the sample code:

```
private int retryCount = 3;
private readonly TimeSpan delay = TimeSpan.FromSeconds(5);

public async Task OperationWithBasicRetryAsync()
{
    int currentRetry = 0;
    for (;;)
    {
          try
```

```
            {
                  await TransientOperationAsync();
                  break;
            }
            catch (Exception ex)
            {
                  Trace.TraceError("Operation Exception");
                  currentRetry++;
                  if (currentRetry > this.retryCount || !IsTransient(ex))
                  {
                        throw;
                  }
            }
            await Task.Delay(delay);
      }
}

private async Task TransientOperationAsync()
{
      ...
}
```

The statement that invokes this method is contained in a try/catch block wrapped in a for loop. The for loop exits if the call to the **TransientOperationAsync** method succeeds without throwing an exception. If the **TransientOperationAsync** method fails, the catch block examines the reason for the failure. If it's believed to be a transient error, the code waits for a short delay before retrying the operation.

The for loop also tracks the number of times that the operation has been attempted, and if the code fails three times, the exception is assumed to be more long lasting. If the exception isn't transient or it's long lasting, the catch handler will throw an exception. This exception exists in the for loop and should be caught by the code that invokes the **OperationWithBasicRetryAsync** method.

Detecting if an error is transient in code

The **IsTransient** method, shown below, checks for a specific set of exceptions that are relevant to the environment the code is run in. The definition of a transient exception will vary according to the resources being accessed and the environment the operation is being performed in:

```
private bool IsTransient(Exception ex)
{
      if (ex is OperationTransientException)
            return true;

      var webException = ex as WebException;
      if (webException != null)
      {
            return new[] {
                  WebExceptionStatus.ConnectionClosed,
                  WebExceptionStatus.Timeout,
                  WebExceptionStatus.RequestCanceled
            }.Contains(webException.Status);
      }
```

```
        return false;
    }
```

Review Questions

Module 4 Review Questions

Scale and auto-scale

You are designing a solution for a university. Students will use the solution to register for classes.

You plan to implement elastic scaling to handle the high levels of activity at the beginning of each class registration period.

What services in Azure are available to support this scenario?

Suggested Answer ↓

You can use Azure services to manually or automatically scale solutions to meet demands. Scaling options include Azure Virtual Machine Scale Sets, Azure App Service, and Azure Cosmos DB.

Auto-scale metrics

You are designing a solution for a university. Students will use the solution to register for classes.

You plan to implement elastic scaling to handle the high levels of activity at the beginning of each class registration period.

You plan to implement an App Service instance and trigger auto-scaling based on metrics?

Which auto-scaling metrics are available for the App Service instance?

Suggested Answer ↓

You can monitor CPU usage, memory usage, data in, data out, average queue requests, and average disk reads o determine when to scale the app.

Handling transient errors

You manage an application for a university that allows students to register for classes.

The application occasionally loses connectivity with the registration database.

What options are available to handle transient errors?

Suggested Answer ↓

When an application loses a connection to a resource such as a database, you can use cancel, retry, or retry with delay logic to handle and potentially resolve the issue. In this case, the database may become unavailable due to network issues. Alternatively, the database may be busy with maintenance tasks. You can implement retry logic to reconnect to the database.

Module 5 Module Developing Azure Cognitive Services Solutions

Cognitive Services Overview

About Cognitive Services

Microsoft Cognitive Services (formerly Project Oxford) are a set of APIs, SDKs and services available to developers to make their applications more intelligent, engaging and discoverable. Microsoft Cognitive Services expands on Microsoft's evolving portfolio of machine learning APIs and enables developers to easily add intelligent features – such as emotion and video detection; facial, speech and vision recognition; and speech and language understanding – into their applications. Our vision is for more personal computing experiences and enhanced productivity aided by systems that increasingly can see, hear, speak, understand and even begin to reason.

Getting started with free trials

Signing up for free trials only takes an email and a few **simple steps**[1] You will need a Microsoft Account if you don't already have one. You will receive a unique pair of keys for each API requested. The second one is just a spare. Please do not share the secret keys with anyone. Trials have both rate limit, in terms of transactions per second or minute, and a monthly usage cap. A transaction is simply an API call. You can upgrade to paid tiers to unlock the restrictions.

Regional availability

The APIs in Cognitive Services are hosted on a growing network of Microsoft-managed data centers. You can find the regional availability for each API in **Azure region list**[2].

1 https://azure.microsoft.com/try/cognitive-services/
2 https://azure.microsoft.com/regions

Develop Solutions using Computer Vision

Computer Vision API v2 overview

The cloud-based Computer Vision API provides developers with access to advanced algorithms for processing images and returning information. By uploading an image or specifying an image URL, Microsoft Computer Vision algorithms can analyze visual content in different ways based on inputs and user choices.

Requirements

- Supported input methods: Raw image binary in the form of an application/octet stream or image URL.
- Supported image formats: JPEG, PNG, GIF, BMP.
- Image file size: Less than 4 MB.
- Image dimension: Greater than 50 x 50 pixels.

Tagging images

Computer Vision API returns tags based on more than 2000 recognizable objects, living beings, scenery, and actions. When tags are ambiguous or not common knowledge, the API response provides 'hints' to clarify the meaning of the tag in context of a known setting. Tags are not organized as a taxonomy and no inheritance hierarchies exist. A collection of content tags forms the foundation for an image 'description' displayed as human readable language formatted in complete sentences. Note, that at this point English is the only supported language for image description.

After uploading an image or specifying an image URL, Computer Vision API's algorithms output tags based on the objects, living beings, and actions identified in the image. Tagging is not limited to the main subject, such as a person in the foreground, but also includes the setting (indoor or outdoor), furniture, tools, plants, animals, accessories, gadgets etc.

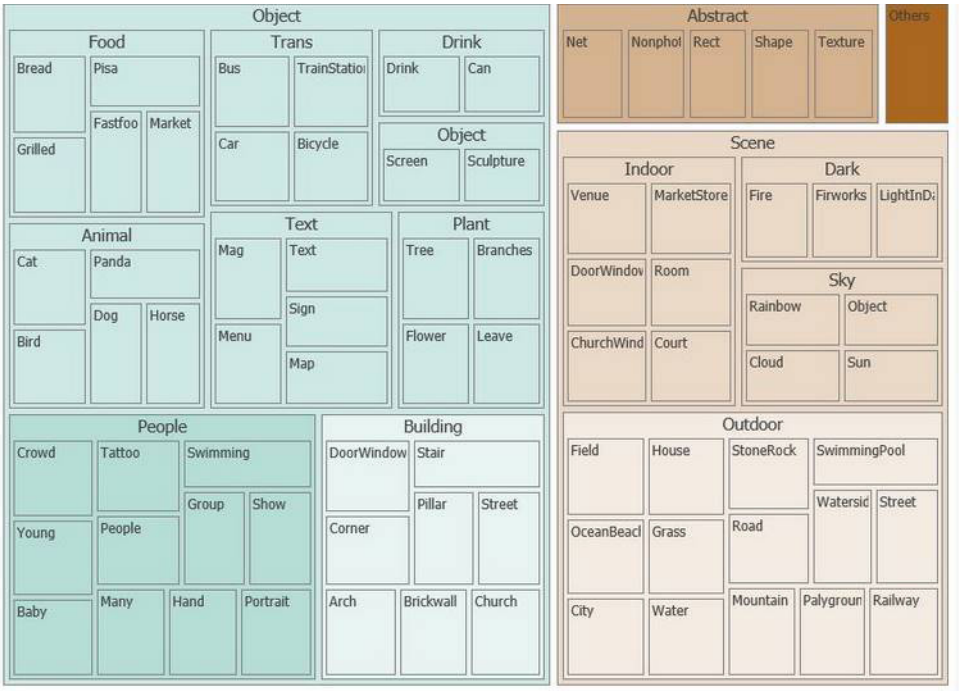
Categorizing images

In addition to tagging and descriptions, Computer Vision API returns the taxonomy-based categories defined in previous versions. These categories are organized as a taxonomy with parent/child hereditary hierarchies. All categories are in English. They can be used alone or with our new models.

The 86-category concept

Based on a list of 86 concepts seen in the following diagram, visual features found in an image can be categorized ranging from broad to specific. For the full taxonomy in text format, see **Category Taxonomy**[3].

3 https://docs.microsoft.com/azure/cognitive-services/computer-vision/category-taxonomy



Identifying image types

There are several ways to categorize images. Computer Vision API can set a boolean flag to indicate whether an image is black and white or color. The API can also set a flag to indicate whether an image is a line drawing or not. It can also indicate whether an image is clip art or not and indicate its quality on a scale of 0-3.

Domain-specific content

In addition to tagging and top-level categorization, Computer Vision API also supports specialized (or domain-specific) information. Specialized information can be implemented as a standalone method or with the high-level categorization. It functions as a means to further refine the 86-category taxonomy through the addition of domain-specific models.

Currently, the only specialized information supported are celebrity recognition and landmark recognition. They are domain-specific refinements for the people and people group categories, and landmarks around the world.

There are two options for using the domain-specific models:

Option one - scoped analysis

Analyze only a chosen model by invoking an HTTP POST call. If you know which model you want to use, specify the model's name. You only get information relevant to that model. For example, you can use this option to only look for celebrity-recognition. The response contains a list of potential matching celebrities, accompanied by their confidence scores.

Option two - enhanced analysis

Analyze to provide additional details related to categories from the 86-category taxonomy. This option is available for use in applications where users want to get generic image analysis in addition to details from one or more domain-specific models. When this method is invoked, the 86-category taxonomy classifier is called first. If any of the categories match that of known/matching models, a second pass of classifier invocations follows. For example, if 'details=all' or "details" include 'celebrities', the method calls the celebrity classifier after the 86-category classifier is called. The result includes tags starting with 'people_'.

Generating descriptions

Computer Vision API's algorithms analyze the content in an image. This analysis forms the foundation for a 'description' displayed as human-readable language in complete sentences. The description summarizes what is found in the image. Computer Vision API's algorithms generate various descriptions based on the objects identified in the image. The descriptions are each evaluated and a confidence score generated. A list is then returned ordered from highest confidence score to lowest.

Perceiving color schemes

The Computer Vision algorithm extracts colors from an image. The colors are analyzed in three different contexts: foreground, background, and whole. They are grouped into 12 dominant accent colors. Those accent colors are black, blue, brown, gray, green, orange, pink, purple, red, teal, white, and yellow. Depending on the colors in an image, simple black and white, or accent colors may be returned in hexadecimal color codes.

Flagging adult content

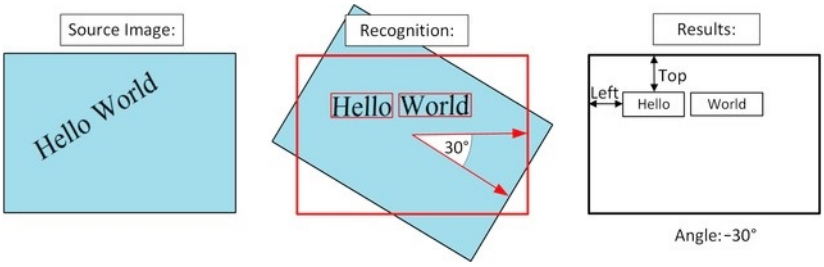
Among the various visual categories is the adult and racy group, which enables detection of adult materials and restricts the display of images containing sexual content. The filter for adult and racy content detection can be set on a sliding scale to accommodate the user's preference.

Optical character recognition (OCR)

OCR technology detects text content in an image and extracts the identified text into a machine-readable character stream. You can use the result for search and numerous other purposes like medical records, security, and banking. It automatically detects the language. OCR saves time and provides convenience for users by allowing them to take photos of text instead of transcribing the text.

OCR supports 25 languages. These languages are: Arabic, Chinese Simplified, Chinese Traditional, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Romanian, Russian, Serbian (Cyrillic and Latin), Slovak, Spanish, Swedish, and Turkish.

If needed, OCR corrects the rotation of the recognized text, in degrees, around the horizontal image axis. OCR provides the frame coordinates of each word as seen in below illustration.



Requirements for OCR:

- The size of the input image must be between 40 x 40 and 3200 x 3200 pixels.
- The image cannot be bigger than 10 megapixels.

The input image can be rotated by any multiple of 90 degrees plus a small angle of up to '40 degrees.

The accuracy of text recognition depends on the quality of the image. An inaccurate reading may be caused by the following situations:

- Blurry images.
- Handwritten or cursive text.
- Artistic font styles.
- Small text size.
- Complex backgrounds, shadows, or glare over text or perspective distortion.
- Oversized or missing capital letters at the beginnings of words
- Subscript, superscript, or strikethrough text.

Limitations: On photos where text is dominant, false positives may come from partially recognized words. On some photos, especially photos without any text, precision can vary a lot depending on the type of image.

Recognize text

This technology allows you to detect and extract printed or handwritten text from images of various objects with different surfaces and backgrounds, such as receipts, posters, business cards, letters, and whiteboards.

Text recognition saves time and effort. You can be more productive by taking an image of text rather than transcribing it. Text recognition makes it possible to digitize notes. This digitization allows you to implement quick and easy search. It also reduces paper clutter.

Input requirements:

- Supported image formats: JPEG, PNG, and BMP.
- Image file size must be less than 4 MB.
- Image dimensions must be at least 40 x 40, at most 3200 x 3200.

Note: this technology is currently in preview and is only available for English text.

Generating thumbnails

A thumbnail is a small representation of a full-size image. Varied devices such as phones, tablets, and PCs create a need for different user experience (UX) layouts and thumbnail sizes. Using smart cropping, this Computer Vision API feature helps solve the problem.

After uploading an image, a high-quality thumbnail gets generated and the Computer Vision API algorithm analyzes the objects within the image. It then crops the image to fit the requirements of the 'region of interest' (ROI). The output gets displayed within a special framework as seen in below illustration. The generated thumbnail can be presented using an aspect ratio that is different from the aspect ratio of the original image to accommodate a user's needs.

The thumbnail algorithm works as follows:

1. Removes distracting elements from the image and recognizes the main object, the 'region of interest' (ROI).
2. Crops the image based on the identified region of interest.
3. Changes the aspect ratio to fit the target thumbnail dimensions.

Analyze an image with C#

The following information shows you how to analyze both a local and a remote image to extract visual features using the Computer Vision Windows client library.

Prerequisites

- To use Computer Vision, you need a subscription key; see **Obtaining Subscription Keys**[4].
- Any edition of Visual Studio 2015 or 2017.
- The **Microsoft.Azure.CognitiveServices.Vision.ComputerVision**[5] client library NuGet package. It isn't necessary to download the package. Installation instructions are provided below.

AnalyzeImageAsync method

The `AnalyzeImageAsync` and `AnalyzeImageInStreamAsync` methods wrap the Analyze Image API for remote and local images, respectively. You can use these methods to extract visual features based on image content and choose which features to return, including:

- A detailed list of tags related to the image content.
- A description of image content in a complete sentence.
- The coordinates, gender, and age of any faces contained in the image.
- The ImageType (clip art or a line drawing).
- The dominant color, the accent color, or whether an image is black & white.
- The category defined in this **taxonomy**[6].
- Does the image contain adult or sexually suggestive content?

4 https://docs.microsoft.com/en-us/azure/cognitive-services/Computer-vision/vision-api-how-to-topics/howtosubscribe
5 https://www.nuget.org/packages/Microsoft.Azure.CognitiveServices.Vision.ComputerVision
6 https://docs.microsoft.com/en-us/azure/cognitive-services/Computer-vision/category-taxonomy

To run the sample, do the following steps:

1. Create a new Visual C# Console App in Visual Studio.
2. Install the Computer Vision client library NuGet package.
   - On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
   - Click the **Browse** tab, and in the **Search** box type "Microsoft.Azure.CognitiveServices.Vision.ComputerVision".
   - Select **Microsoft.Azure.CognitiveServices.Vision.ComputerVision** when it displays, then click the checkbox next to your project name, and **Install**.
3. Replace `Program.cs` with the following code.
4. Replace `<Subscription Key>` with your valid subscription key.
5. Change `computerVision.AzureRegion = AzureRegions.Westcentralus` to the location where you obtained your subscription keys, if necessary.
6. Optionally, replace `<LocalImage>` with the path and file name of a local image (will be ignored if not set).
7. Optionally, set `remoteImageUrl` to a different image.
8. Run the program.

```
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;

using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

namespace ImageAnalyze
{
    class Program
    {
        // subscriptionKey = "0123456789abcdef0123456789ABCDEF"
        private const string subscriptionKey = "<SubscriptionKey>";

        // localImagePath = @"C:\Documents\LocalImage.jpg"
        private const string localImagePath = @"<LocalImage>";

        private const string remoteImageUrl =
            "http://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_wa-
terfall.jpg";

        // Specify the features to return
        private static readonly List<VisualFeatureTypes> features =
            new List<VisualFeatureTypes>()
        {
            VisualFeatureTypes.Categories, VisualFeatureTypes.Description,
            VisualFeatureTypes.Faces, VisualFeatureTypes.ImageType,
            VisualFeatureTypes.Tags
```

```
        };

        static void Main(string[] args)
        {
            ComputerVisionAPI computerVision = new ComputerVisionAPI(
                new ApiKeyServiceClientCredentials(subscriptionKey),
                new System.Net.Http.DelegatingHandler[] { });

            // You must use the same region as you used to get your sub-
scription
            // keys. For example, if you got your subscription keys from
westus,
            // replace "Westcentralus" with "Westus".
            //
            // Free trial subscription keys are generated in the westcen-
tralus
            // region. If you use a free trial subscription key, you
shouldn't
            // need to change the region.

            // Specify the Azure region
            computerVision.AzureRegion = AzureRegions.Westcentralus;

            Console.WriteLine("Images being analyzed ...");
            var t1 = AnalyzeRemoteAsync(computerVision, remoteImageUrl);
            var t2 = AnalyzeLocalAsync(computerVision, localImagePath);

            Task.WhenAll(t1, t2).Wait(5000);
            Console.WriteLine("Press any key to exit");
            Console.ReadLine();
        }

        // Analyze a remote image
        private static async Task AnalyzeRemoteAsync(
            ComputerVisionAPI computerVision, string imageUrl)
        {
            if (!Uri.IsWellFormedUriString(imageUrl, UriKind.Absolute))
            {
                Console.WriteLine(
                    "\nInvalid remoteImageUrl:\n{0} \n", imageUrl);
                return;
            }

            ImageAnalysis analysis =
                await computerVision.AnalyzeImageAsync(imageUrl, features);
            DisplayResults(analysis, imageUrl);
        }

        // Analyze a local image
        private static async Task AnalyzeLocalAsync(
            ComputerVisionAPI computerVision, string imagePath)
```

```
        {
            if (!File.Exists(imagePath))
            {
                Console.WriteLine(
                    "\nUnable to open or read localImagePath:\n{0} \n",
imagePath);
                return;
            }

            using (Stream imageStream = File.OpenRead(imagePath))
            {
                ImageAnalysis analysis = await computerVision.AnalyzeImage-
InStreamAsync(
                    imageStream, features);
                DisplayResults(analysis, imagePath);
            }
        }

        // Display the most relevant caption for the image
        private static void DisplayResults(ImageAnalysis analysis, string
imageUri)
        {
            Console.WriteLine(imageUri);
            Console.WriteLine(analysis.Description.Captions[0].Text +
"\n");
        }
    }
}
```

AnalyzeImageAsync response

A successful response displays the most relevant caption for each image.

```
http://upload.wikimedia.org/wikipedia/commons/3/3c/Shaki_waterfall.jpg
a large waterfall over a rocky cliff
```

Generate a thumbnail with C#

The following information shows you how to generate a thumbnail from an image using the Computer Vision Windows client library.

Prerequisites

- To use Computer Vision, you need a subscription key; see **Obtaining Subscription Keys**[7].
- Any edition of Visual Studio 2015 or 2017.

7 https://docs.microsoft.com/en-us/azure/cognitive-services/Computer-vision/vision-api-how-to-topics/howtosubscribe

- The **Microsoft.Azure.CognitiveServices.Vision.ComputerVision**[8] client library NuGet package. It isn't necessary to download the package. Installation instructions are provided below.

GenerateThumbnailAsync method

The `GenerateThumbnailAsync` and `GenerateThumbnailInStreamAsync` methods wrap the **Get Thumbnail API**[9] for remote and local images, respectively. You can use these methods to generate a thumbnail of an image. You specify the height and width, which can differ from the aspect ratio of the input image. Computer Vision uses smart cropping to intelligently identify the region of interest and generate cropping coordinates based on that region.

To run the sample, do the following steps:

1. Create a new Visual C# Console App in Visual Studio.
2. Install the Computer Vision client library NuGet package.
   - On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
   - Click the **Browse** tab, and in the **Search** box type "Microsoft.Azure.CognitiveServices.Vision.ComputerVision".
   - Select **Microsoft.Azure.CognitiveServices.Vision.ComputerVision** when it displays, then click the checkbox next to your project name, and **Install**.
3. Replace `Program.cs` with the following code.
4. Replace `<Subscription Key>` with your valid subscription key.
5. Change `computerVision.AzureRegion = AzureRegions.Westcentralus` to the location where you obtained your subscription keys, if necessary.
6. Optionally, replace `<LocalImage>` with the path and file name of a local image (will be ignored if not set).
7. Optionally, set `remoteImageUrl` to a different image.
8. Optionally, set `writeThumbnailToDisk` to `true` to save the thumbnail to disk.
9. Run the program.

```
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;

using System;
using System.IO;
using System.Threading.Tasks;

namespace ImageThumbnail
{
    class Program
    {
        private const bool writeThumbnailToDisk = false;

        // subscriptionKey = "0123456789abcdef0123456789ABCDEF"
```

8 https://www.nuget.org/packages/Microsoft.Azure.CognitiveServices.Vision.ComputerVision
9 https://westus.dev.cognitive.microsoft.com/docs/services/5adf991815e1060e6355ad44/operations/56f91f2e778daf14a499e1fb

```
        private const string subscriptionKey = "<SubscriptionKey>";

        // localImagePath = @"C:\Documents\LocalImage.jpg"
        private const string localImagePath = @"<LocalImage>";

        private const string remoteImageUrl =
            "https://upload.wikimedia.org/wikipedia/commons/9/94/Blood-
hound_Puppy.jpg";

        private const int thumbnailWidth = 100;
        private const int thumbnailHeight = 100;

        static void Main(string[] args)
        {
            ComputerVisionAPI computerVision = new ComputerVisionAPI(
                new ApiKeyServiceClientCredentials(subscriptionKey),
                new System.Net.Http.DelegatingHandler[] { });

            // You must use the same region as you used to get your sub-
scription
            // keys. For example, if you got your subscription keys from
westus,
            // replace "Westcentralus" with "Westus".
            //
            // Free trial subscription keys are generated in the westcen-
tralus
            // region. If you use a free trial subscription key, you
shouldn't
            // need to change the region.

            // Specify the Azure region
            computerVision.AzureRegion = AzureRegions.Westcentralus;

            Console.WriteLine("Images being analyzed ...\n");
            var t1 = GetRemoteThumbnailAsync(computerVision, remoteIma-
geUrl);
            var t2 = GetLocalThumbnailAsnc(computerVision, localImagePath);

            Task.WhenAll(t1, t2).Wait(5000);
            Console.WriteLine("Press any key to exit");
            Console.ReadLine();
        }

        // Create a thumbnail from a remote image
        private static async Task GetRemoteThumbnailAsync(
            ComputerVisionAPI computerVision, string imageUrl)
        {
            if (!Uri.IsWellFormedUriString(imageUrl, UriKind.Absolute))
            {
                Console.WriteLine(
                    "\nInvalid remoteImageUrl:\n{0} \n", imageUrl);
```

```
                return;
            }

            Stream thumbnail = await computerVision.GenerateThumbnailAsync(
                thumbnailWidth, thumbnailHeight, imageUrl, true);

            string path = Environment.CurrentDirectory;
            string imageName = imageUrl.Substring(imageUrl.LastIndexOf('/')
+ 1);
            string thumbnailFilePath =
                path + "\\" + imageName.Insert(imageName.Length - 4, "_
thumb");

            // Save the thumbnail to the current working directory,
            // using the original name with the suffix "_thumb".
            SaveThumbnail(thumbnail, thumbnailFilePath);
        }

        // Create a thumbnail from a local image
        private static async Task GetLocalThumbnailAsnc(
            ComputerVisionAPI computerVision, string imagePath)
        {
            if (!File.Exists(imagePath))
            {
                Console.WriteLine(
                    "\nUnable to open or read localImagePath:\n{0} \n",
imagePath);
                return;
            }

            using (Stream imageStream = File.OpenRead(imagePath))
            {
                Stream thumbnail = await computerVision.GenerateThumbnail-
InStreamAsync(
                    thumbnailWidth, thumbnailHeight, imageStream, true);

                string thumbnailFilePath =
                    localImagePath.Insert(localImagePath.Length - 4, "_
thumb");

                // Save the thumbnail to the same folder as the original
image,
                // using the original name with the suffix "_thumb".
                SaveThumbnail(thumbnail, thumbnailFilePath);
            }
        }

        // Save the thumbnail locally.
        // NOTE: This will overwrite an existing file of the same name.
        private static void SaveThumbnail(Stream thumbnail, string thumb-
nailFilePath)
```

```
            {
                if (writeThumbnailToDisk)
                {
                    using (Stream file = File.Create(thumbnailFilePath))
                    {
                        thumbnail.CopyTo(file);
                    }
                }
                Console.WriteLine("Thumbnail {0} written to: {1}\n",
                    writeThumbnailToDisk ? "" : "NOT", thumbnailFilePath);
            }
        }
    }
```

GenerateThumbnailAsync response

A successful response saves the thumbnail for each image locally and displays the thumbnail's location, for example:

```
Thumbnail written to: C:\Documents\LocalImage_thumb.jpg

Thumbnail written to: ...\bin\Debug\Bloodhound_Puppy_thumb.jpg
```

Extract handwritten text with C#

The following information shows you how to extract handwritten text from an image using the Computer Vision Windows client library.

Prerequisites

- To use Computer Vision, you need a subscription key; see **Obtaining Subscription Keys**[10].
- Any edition of Visual Studio 2015 or 2017.
- The **Microsoft.Azure.CognitiveServices.Vision.ComputerVision**[11] client library NuGet package. It isn't necessary to download the package. Installation instructions are provided below.

AnalyzeImageAsync method

The `AnalyzeImageAsync` and `AnalyzeImageInStreamAsync` methods wrap the Analyze Image API for remote and local images, respectively. You can use these methods to extract visual features based on image content and choose which features to return, including:

- A detailed list of tags related to the image content.
- A description of image content in a complete sentence.
- The coordinates, gender, and age of any faces contained in the image.
- The ImageType (clip art or a line drawing).

10 https://docs.microsoft.com/en-us/azure/cognitive-services/Computer-vision/vision-api-how-to-topics/howtosubscribe
11 https://www.nuget.org/packages/Microsoft.Azure.CognitiveServices.Vision.ComputerVision

- The dominant color, the accent color, or whether an image is black & white.
- The category defined in this **taxonomy**[12].
- Does the image contain adult or sexually suggestive content?

To run the sample, do the following steps:

1. Create a new Visual C# Console App in Visual Studio.
2. Install the Computer Vision client library NuGet package.
   - On the menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
   - Click the **Browse** tab, and in the **Search** box type "Microsoft.Azure.CognitiveServices.Vision.ComputerVision".
   - Select **Microsoft.Azure.CognitiveServices.Vision.ComputerVision** when it displays, then click the checkbox next to your project name, and **Install**.
3. Replace `Program.cs` with the following code.
4. Replace `<Subscription Key>` with your valid subscription key.
5. Change `computerVision.AzureRegion = AzureRegions.Westcentralus` to the location where you obtained your subscription keys, if necessary.
6. Optionally, replace `<LocalImage>` with the path and file name of a local image (will be ignored if not set).
7. Optionally, set `remoteImageUrl` to a different image.
8. Run the program.

```
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;

using System;
using System.IO;
using System.Threading.Tasks;

namespace ImageHandText
{
    class Program
    {
        // subscriptionKey = "0123456789abcdef0123456789ABCDEF"
        private const string subscriptionKey = "<SubscriptionKey>";

        // localImagePath = @"C:\Documents\LocalImage.jpg"
        private const string localImagePath = @"<LocalImage>";

        private const string remoteImageUrl =
            "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/" +
            "Cursive_Writing_on_Notebook_paper.jpg/" +
            "800px-Cursive_Writing_on_Notebook_paper.jpg";

        private const int numberOfCharsInOperationId = 36;
```

12 https://docs.microsoft.com/en-us/azure/cognitive-services/Computer-vision/category-taxonomy

```
        static void Main(string[] args)
        {
            ComputerVisionAPI computerVision = new ComputerVisionAPI(
                new ApiKeyServiceClientCredentials(subscriptionKey),
                new System.Net.Http.DelegatingHandler[] { });

            // You must use the same region as you used to get your sub-
scription
            // keys. For example, if you got your subscription keys from
westus,
            // replace "Westcentralus" with "Westus".
            //
            // Free trial subscription keys are generated in the westcen-
tralus
            // region. If you use a free trial subscription key, you
shouldn't
            // need to change the region.

            // Specify the Azure region
            computerVision.AzureRegion = AzureRegions.Westcentralus;

            Console.WriteLine("Images being analyzed ...");
            var t1 = ExtractRemoteHandTextAsync(computerVision, remoteIma-
geUrl);
            var t2 = ExtractLocalHandTextAsync(computerVision, localImage-
Path);

            Task.WhenAll(t1, t2).Wait(5000);
            Console.WriteLine("Press any key to exit");
            Console.ReadLine();
        }

        // Recognize text from a remote image
        private static async Task ExtractRemoteHandTextAsync(
            ComputerVisionAPI computerVision, string imageUrl)
        {
            if (!Uri.IsWellFormedUriString(imageUrl, UriKind.Absolute))
            {
                Console.WriteLine(
                    "\nInvalid remoteImageUrl:\n{0} \n", imageUrl);
                return;
            }

            // Start the async process to recognize the text
            RecognizeTextHeaders textHeaders = await computerVision.Recog-
nizeTextAsync(
                    imageUrl, TextRecognitionMode.Handwritten);

            await GetTextAsync(computerVision, textHeaders.OperationLoca-
tion);
```

```
        }

        // Recognize text from a local image
        private static async Task ExtractLocalHandTextAsync(
            ComputerVisionAPI computerVision, string imagePath)
        {
            if (!File.Exists(imagePath))
            {
                Console.WriteLine(
                    "\nUnable to open or read localImagePath:\n{0} \n",
imagePath);
                return;
            }

            using (Stream imageStream = File.OpenRead(imagePath))
            {
                // Start the async process to recognize the text
                RecognizeTextInStreamHeaders textHeaders =
                    await computerVision.RecognizeTextInStreamAsync(
                        imageStream, TextRecognitionMode.Handwritten);

                await GetTextAsync(computerVision, textHeaders.OperationLo-
cation);
            }
        }

        // Retrieve the recognized text
        private static async Task GetTextAsync(
            ComputerVisionAPI computerVision, string operationLocation)
        {
            // Retrieve the URI where the recognized text will be
            // stored from the Operation-Location header
            string operationId = operationLocation.Substring(
                operationLocation.Length - numberOfCharsInOperationId);

            Console.WriteLine("\nCalling GetHandwritingRecognitionOpera-
tionResultAsync()");
            TextOperationResult result =
                await computerVision.GetTextOperationResultAsync(operation-
Id);

            // Wait for the operation to complete
            int i = 0;
            int maxRetries = 10;
            while ((result.Status == TextOperationStatusCodes.Running ||
                    result.Status == TextOperationStatusCodes.NotStarted)
&& i++ < maxRetries)
            {
                Console.WriteLine(
                    "Server status: {0}, waiting {1} seconds...", result.
Status, i);
```

```
                await Task.Delay(1000);

                result = await computerVision.GetTextOperationResultAsync(-
operationId);
            }

            // Display the results
            Console.WriteLine();
            var lines = result.RecognitionResult.Lines;
            foreach(Line line in lines)
            {
                Console.WriteLine(line.Text);
            }
            Console.WriteLine();
        }
    }
}
```

RecognizeTextAsync response

A successful response displays the lines of recognized text for each image.

```
Calling GetHandwritingRecognitionOperationResultAsync()

Calling GetHandwritingRecognitionOperationResultAsync()
Server status: Running, waiting 1 seconds...
Server status: Running, waiting 1 seconds...

dog
The quick brown fox jumps over the lazy
Pack my box with five dozen liquor jugs
```

Develop Solutions using Bing Web Search

Bing Web Search API overview

The Bing Web Search API provides an experience similar to Bing.com/search by returning search results that Bing determines are relevant to the user's query. The results may include Web pages, images, videos, news, and entities, along with related search queries, spelling corrections, time zones, unit conversion, translations, and calculations. The kinds of results you get are based on their relevance and also the tier of the Bing Search APIs to which you subscribe.

If you're building a search results page that displays any content that's relevant to the user's search query, call this API instead of calling the other content-specific Bing APIs. The only time you should need to call the content-specific APIs, such as the Image Search API or News Search API, is if you need answers from only that API. For example, if you're building an image-only search results page or a news-only search results page.

If Bing didn't find content from one of the content-specific APIs relevant enough, it would not include it in the search results. For example, the results could include webpages, news articles, and videos but not images. However, it's possible that if you called the Image Search API directly with the same query, it would return images.

If you don't need webpages but you do need answers from more than one of the other APIs, such as images and news, you'd still call this API. For example, if you only wanted Images and News, you'd call this API and set `responseFilter` query parameter to limit the results to only Images and News.

Search response

When you send Bing a search request, it sends back a response that contains a SearchResponse object in the body of the response. The object includes a field for each answer that Bing thought was relevant to the user's query term. The following shows an example of the response object if Bing returned all answers.

```
{
    "_type": "SearchResponse",
    "queryContext": {...},
    "webPages": {...},
    "images": {...},
    "relatedSearches": {...},
    "videos": {...},
    "news": {...},
    "spellSuggestion": {...},
    "computation": {...},
    "timeZone": {...},
    "rankingResponse": {...}
}, ...
```

Typically, Bing returns a subset of the answers. For example, if the query term was sailing dinghies, the response might include only `webPages`, `images`, and `rankingResponse`. Unless you've used `responseFilter` to filter out webpages, the response always includes the `webpages` and `rankingResponse` answers.

Answer	Description
Webpages	The webPages answer contains a list of links to webpages that Bing thought were relevant to the query. Each webpage in the list includes the page's name, url, display URL, short description of the content and the date Bing found the content.
Images	The images answer contains a list of images that Bing thought were relevant to the query. Each image in the list includes the URL of the image, its size, its dimensions, and its encoding format. The image object also includes the URL of a thumbnail of the image and the thumbnail's dimensions.
Related searches	The `relatedSearches` answer contains a list of the most popular related queries made by other users. Each query in the list includes a query string (`text`), a query string with hit highlighting characters (`displayText`), and a URL (`webSearchUrl`) to Bing's search results page for that query.
Videos	The videos answer contains a list of videos that Bing thought were relevant to the query. Each video in the list includes the URL of the video, its duration, its dimensions, and its encoding format. The video object also includes the URL of a thumbnail of the video and the thumbnail's dimensions.
News	The news answer contains a list of news articles that Bing thought were relevant to the query. Each news article in the list includes the article's name, description, and URL to the article on the host's website. If the article contains an image, the object includes a thumbnail of the image.
Computation	If the user enters a mathematical expression or a unit conversion query, the response may contain a Computation answer. The `computation` answer contains the normalized expression and its result.
TimeZone	If the user enters a time or date query, the response may contain a TimeZone answer. This answer supports implicit or explicit queries. An implicit query such as What time is it?, returns the local time of the user's location. An explicit query such as What time is it in Seattle?, returns the local time of Seattle, WA.

Develop a Bing Web Search query in C#

The Bing Web Search API provides a experience similar to Bing.com/Search by returning search results that Bing determines are relevant to the user's query. The results may include Web pages, images, videos, news, and entities, along with related search queries, spelling corrections, time zones, unit conversion,

translations, and calculations. The kinds of results you get are based on their relevance and the tier of the Bing Search APIs to which you subscribe.

This lesson includes a simple console application that performs a Bing Web Search API query and displays the returned raw search results, which are in JSON format. While this application is written in C#, the API is a RESTful Web service compatible with any programming language that can make HTTP requests and parse JSON.

The example program uses .NET Core classes only and runs on Windows using the .NET CLR or on Linux or macOS using Mono.

Prerequisites

You will need Visual Studio 2017 to get this code running on Windows. (The free Community Edition will work.)

You must have a **Cognitive Services API account**[13] with **Bing Search APIs**. The free trial is sufficient for this quickstart. You need the access key provided when you activate your free trial, or you may use a paid subscription key from your Azure dashboard.

Running the application

To run this application, follow these steps.

1. Create a new Console solution in Visual Studio.
2. Replace `Program.cs` with the provided code.
3. Replace the `accessKey` value with an access key valid for your subscription.
4. Run the program.

```
using System;
using System.Text;
using System.Net;
using System.IO;
using System.Collections.Generic;

namespace BingSearchApisQuickstart
{

    class Program
    {
        // **********************************************
        // *** Update or verify the following values. ***
        // **********************************************

        // Replace the accessKey string value with your valid access key.
        const string accessKey = "enter key here";

        // Verify the endpoint URI.  At this writing, only one endpoint is
used for Bing
        // search APIs.  In the future, regional endpoints may be availa-
ble.  If you
```

13 https://docs.microsoft.com/azure/cognitive-services/cognitive-services-apis-create-account

```
        // encounter unexpected authorization errors, double-check this
value against
        // the endpoint for your Bing Web search instance in your Azure
dashboard.
        const string uriBase = "https://api.cognitive.microsoft.com/bing/
v7.0/search";

        const string searchTerm = "Microsoft Cognitive Services";

        // Used to return search results including relevant headers
        struct SearchResult
        {
            public String jsonResult;
            public Dictionary<String, String> relevantHeaders;
        }

        static void Main()
        {
            Console.OutputEncoding = System.Text.Encoding.UTF8;

            if (accessKey.Length == 32)
            {
                Console.WriteLine("Searching the Web for: " + searchTerm);

                SearchResult result = BingWebSearch(searchTerm);

                Console.WriteLine("\nRelevant HTTP Headers:\n");
                foreach (var header in result.relevantHeaders)
                    Console.WriteLine(header.Key + ": " + header.Value);

                Console.WriteLine("\nJSON Response:\n");
                Console.WriteLine(JsonPrettyPrint(result.jsonResult));
            }
            else
            {
                Console.WriteLine("Invalid Bing Search API subscription
key!");
                Console.WriteLine("Please paste yours into the source
code.");
            }

            Console.Write("\nPress Enter to exit ");
            Console.ReadLine();
        }

        /// <summary>
        /// Performs a Bing Web search and return the results as a
SearchResult.
        /// </summary>
        static SearchResult BingWebSearch(string searchQuery)
        {
```

```
            // Construct the URI of the search request
            var uriQuery = uriBase + "?q=" + Uri.EscapeDataString(search-
Query);

            // Perform the Web request and get the response
            WebRequest request = HttpWebRequest.Create(uriQuery);
            request.Headers["Ocp-Apim-Subscription-Key"] = accessKey;
            HttpWebResponse response = (HttpWebResponse)request.GetRespon-
seAsync().Result;
            string json = new StreamReader(response.GetResponseStream()).
ReadToEnd();

            // Create result object for return
            var searchResult = new SearchResult()
            {
                jsonResult = json,
                relevantHeaders = new Dictionary<String, String>()
            };

            // Extract Bing HTTP headers
            foreach (String header in response.Headers)
            {
                if (header.StartsWith("BingAPIs-") || header.StartsWith("X-
MSEdge-"))
                    searchResult.relevantHeaders[header] = response.Head-
ers[header];
            }

            return searchResult;
        }

        /// <summary>
        /// Formats the given JSON string by adding line breaks and in-
dents.
        /// </summary>
        /// <param name="json">The raw JSON string to format.</param>
        /// <returns>The formatted JSON string.</returns>
        static string JsonPrettyPrint(string json)
        {
            if (string.IsNullOrEmpty(json))
                return string.Empty;

            json = json.Replace(Environment.NewLine, "").Replace("\t", "");

            StringBuilder sb = new StringBuilder();
            bool quote = false;
            bool ignore = false;
            char last = ' ';
            int offset = 0;
            int indentLength = 2;
```

```
            foreach (char ch in json)
            {
                switch (ch)
                {
                    case '"':
                        if (!ignore) quote = !quote;
                        break;
                    case '\\':
                        if (quote && last != '\\') ignore = true;
                        break;
                }

                if (quote)
                {
                    sb.Append(ch);
                    if (last == '\\' && ignore) ignore = false;
                }
                else
                {
                    switch (ch)
                    {
                        case '{':
                        case '[':
                            sb.Append(ch);
                            sb.Append(Environment.NewLine);
                            sb.Append(new string(' ', ++offset * indentLe-
ngth));
                            break;
                        case '}':
                        case ']':
                            sb.Append(Environment.NewLine);
                            sb.Append(new string(' ', --offset * indentLe-
ngth));
                            sb.Append(ch);
                            break;
                        case ',':
                            sb.Append(ch);
                            sb.Append(Environment.NewLine);
                            sb.Append(new string(' ', offset * indentLe-
ngth));
                            break;
                        case ':':
                            sb.Append(ch);
                            sb.Append(' ');
                            break;
                        default:
                            if (quote || ch != ' ') sb.Append(ch);
                            break;
                    }
                }
                last = ch;
```

```
            }

            return sb.ToString().Trim();
        }
    }
}
```

JSON response

A sample response follows. To limit the length of the JSON, only a single result is shown, and other parts of the response have been truncated.

```
{
  "_type": "SearchResponse",
  "queryContext": {
    "originalQuery": "Microsoft Cognitive Services"
  },
  "webPages": {
    "webSearchUrl": "https://www.bing.com/search?q=Microsoft+cognitive+ser-
vices",
    "totalEstimatedMatches": 22300000,
    "value": [
      {
        "id": "https://api.cognitive.microsoft.com/api/v7/#WebPages.0",
        "name": "Microsoft Cognitive Services",
        "url": "https://www.microsoft.com/cognitive-services",
        "displayUrl": "https://www.microsoft.com/cognitive-services",
        "snippet": "Knock down barriers between you and your ideas. Enable
natural and contextual interaction with tools that augment users' experi-
ences via the power of machine-based AI. Plug them in and bring your ideas
to life.",
        "deepLinks": [
          {
            "name": "Face API",
            "url": "https://azure.microsoft.com/services/cognitive-servic-
es/face/",
            "snippet": "Add facial recognition to your applications to
detect, identify, and verify faces using a Face API from Microsoft Azure.
... Cognitive Services; Face API;"
          },
          {
            "name": "Text Analytics",
            "url": "https://azure.microsoft.com/services/cognitive-servic-
es/text-analytics/",
            "snippet": "Cognitive Services; Text Analytics API; Text Ana-
lytics API . Detect sentiment, ... you agree that Microsoft may store it
and use it to improve Microsoft services, ..."
          },
          {
            "name": "Computer Vision API",
            "url": "https://azure.microsoft.com/services/cognitive-servic-
```

```
es/computer-vision/",
            "snippet": "Extract the data you need from images using optical
character recognition and image analytics with Computer Vision APIs from
Microsoft Azure."
          },
          {
            "name": "Emotion",
            "url": "https://www.microsoft.com/cognitive-services/en-us/
emotion-api",
            "snippet": "Cognitive Services Emotion API - microsoft.com"
          },
          {
            "name": "Bing Speech API",
            "url": "https://azure.microsoft.com/services/cognitive-servic-
es/speech/",
            "snippet": "Add speech recognition to your applications, in-
cluding text to speech, with a speech API from Microsoft Azure. ... Cogni-
tive Services; Bing Speech API;"
          },
          {
            "name": "Get Started for Free",
            "url": "https://azure.microsoft.com/services/cognitive-servic-
es/",
            "snippet": "Add vision, speech, language, and knowledge capa-
bilities to your applications using intelligence APIs and SDKs from Cogni-
tive Services."
          }
        ]
      }
    ]
  },
  "relatedSearches": {
    "id": "https://api.cognitive.microsoft.com/api/v7/#RelatedSearches",
    "value": [
      {
        "text": "microsoft bot framework",
        "displayText": "microsoft bot framework",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+bot+frame-
work"
      },
      {
        "text": "microsoft cognitive services youtube",
        "displayText": "microsoft cognitive services youtube",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+cogni-
tive+services+youtube"
      },
      {
        "text": "microsoft cognitive services search api",
        "displayText": "microsoft cognitive services search api",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+cogni-
tive+services+search+api"
```

```
      },
      {
        "text": "microsoft cognitive services news",
        "displayText": "microsoft cognitive services news",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+cogni-
tive+services+news"
      },
      {
        "text": "ms cognitive service",
        "displayText": "ms cognitive service",
        "webSearchUrl": "https://www.bing.com/search?q=ms+cognitive+ser-
vice"
      },
      {
        "text": "microsoft cognitive services text analytics",
        "displayText": "microsoft cognitive services text analytics",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+cogni-
tive+services+text+analytics"
      },
      {
        "text": "microsoft cognitive services toolkit",
        "displayText": "microsoft cognitive services toolkit",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+cogni-
tive+services+toolkit"
      },
      {
        "text": "microsoft cognitive services api",
        "displayText": "microsoft cognitive services api",
        "webSearchUrl": "https://www.bing.com/search?q=microsoft+cogni-
tive+services+api"
      }
    ]
  },
  "rankingResponse": {
    "mainline": {
      "items": [
        {
          "answerType": "WebPages",
          "resultIndex": 0,
          "value": {
            "id": "https://api.cognitive.microsoft.com/api/v7/#WebPages.0"
          }
        }
      ]
    },
    "sidebar": {
      "items": [
        {
          "answerType": "RelatedSearches",
          "value": {
            "id": "https://api.cognitive.microsoft.com/api/v7/#Related-
```

```
Searches"
          }
        }
      ]
    }
  }
}
```

Filtering the answers in the search response

When you query the web, Bing returns all content that it thinks is relevant to the search. For example, if the search query is "sailing+dinghies", the response might contain the following answers:

```
{
    "_type" : "SearchResponse",
    "webPages" : {
        "webSearchUrl" : "https:\/\/www.bing.com\/cr?IG=3A43C...",
        "totalEstimatedMatches" : 262000,
        "value" : [...]
    },
    "images" : {
        "id" : "https:\/\/api.cognitive.microsoft.com\/api\/v7\/#Images",
        "readLink" : "https:\/\/api.cognitive.microsoft.com\/api\/v7\/
images\/search?q=sail...",
        "webSearchUrl" : "https:\/\/www.bing.com\/cr?IG=3A43CA-
5CA6464E5D...",
        "isFamilyFriendly" : true,
        "value" : [...]
    },
    "rankingResponse" : {
        "mainline" : {
            "items" : [...]
        }
    }
}
```

If you're interested in specific types of content such as images, videos, and news, you can request only those answers by using the `responseFilter` query parameter. If Bing finds relevant content for the specified answers, Bing returns it. The response filter is a comma-delimited list of answers. The following shows how to use `responseFilter` to request images, videos, and news of sailing dinghies. When you encode the query string, the commas change to %2C.

```
GET https://api.cognitive.microsoft.com/bing/v7.0/search?q=sailing+din-
ghies&responseFilter=images%2Cvideos%2Cnews&mkt=en-us HTTP/1.1
Ocp-Apim-Subscription-Key: 123456789ABCDE
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0; Tri-
dent/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 822)
X-Search-ClientIP: 999.999.999.999
X-Search-Location:  47.60357;long:-122.3295;re:100
X-MSEdge-ClientID: <blobFromPriorResponseGoesHere>
```

```
Host: api.cognitive.microsoft.com
```

The following shows the response to the previous query. As you can see Bing didn't find relevant video and news results, so the response doesn't include them.

```
{
    "_type" : "SearchResponse",
    "images" : {
        "id" : "https:\/\/api.cognitive.microsoft.com\/api\/v7\/#Images",
        "readLink" : "https:\/\/api.cognitive.microsoft.com\/api\/v7\/
images\/search?q=sail...",
        "webSearchUrl" : "https:\/\/www.bing.com\/cr?IG=3AD-
78B183C56456C...",
        "isFamilyFriendly" : true,
        "value" : [...]
    },
    "rankingResponse" : {
        "mainline" : {
            "items" : [{
                "answerType" : "Images",
                "value" : {
                    "id" : "https:\/\/api.cognitive.microsoft.com\/api\/
v7\/#Images"
                }
            }]
        }
    }
}
```

Although Bing did not return video and news results in the previous response, it does not mean that video and news content does not exist. It simply means that the page didn't include them. However, if you page through more results, the subsequent pages would likely include them. Also, if you call the Video Search API and News Search API endpoints directly, the response would likely contain results.

You are discouraged from using `responseFilter` to get results from a single API. If you want content from a single Bing API, call that API directly. For example, to receive only images, send a request to the Image Search API endpoint, `https://api.cognitive.microsoft.com/bing/v7.0/images/search` or one of the other Images endpoints. Calling the single API is important not only for performance reasons but because the content-specific APIs offer richer results. For example, you can use filters that are not available to the Web Search API to filter the results.

To get search results from a specific domain, include the `site:` query operator in the query string.

```
https://api.cognitive.microsoft.com/bing/v7.0/search?q=sailing+din-
ghies+site:contososailing.com&mkt=en-us
```

Note: Depending on the query, if you use the `site:` query operator, there is the chance that the response may contain adult content regardless of the `safeSearch` setting. You should use `site:` only if you are aware of the content on the site and your scenario supports the possibility of adult content.

Limiting the number of answers in the response

Bing includes answers in the response based on ranking. For example, if you query *sailing+dinghies*, Bing returns `webpages`, `images`, `videos`, and `relatedSearches`.

```
{
    "_type" : "SearchResponse",
    "queryContext" : {
        "originalQuery" : "sailing dinghies"
    },
    "webPages" : {...},
    "images" : {...},
    "relatedSearches" : {...},
    "videos" : {...},
    "rankingResponse" : {...}
}
```

To limit the number of answers that Bing returns to the top two answers (webpages and images), set the `answerCount` query parameter to 2.

```
GET https://api.cognitive.microsoft.com/bing/v7.0/search?q=sailing+din-
ghies&answerCount=2&mkt=en-us HTTP/1.1
Ocp-Apim-Subscription-Key: 123456789ABCDE
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0; Tri-
dent/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 822)
X-Search-ClientIP: 999.999.999.999
X-Search-Location:  47.60357;long:-122.3295;re:100
X-MSEdge-ClientID: <blobFromPriorResponseGoesHere>
Host: api.cognitive.microsoft.com
```

The response includes only webPages and images.

```
{
    "_type" : "SearchResponse",
    "queryContext" : {
        "originalQuery" : "sailing dinghies"
    },
    "webPages" : {...},
    "images" : {...},
    "rankingResponse" : {...}
}
```

If you add the `responseFilter` query parameter to the previous query and set it to webpages and news, the response contains only webpages because news is not ranked.

```
{
    "_type" : "SearchResponse",
    "queryContext" : {
        "originalQuery" : "sailing dinghies"
    },
    "webPages" : {...},
    "rankingResponse" : {...}
```

```
}
```

Promoting answers that are not ranked

If the top ranked answers that Bing returns for a query are webpages, images, videos, and relatedSearches, the response would include those answers. If you set `answerCount` to two (2), Bing returns the top two ranked `answers`: webpages and images. If you want Bing to include images and videos in the response, specify the `promote` query parameter and set it to images and videos.

```
GET https://api.cognitive.microsoft.com/bing/v7.0/search?q=sailing+dinghies&answerCount=2&promote=images%2Cvideos&mkt=en-us HTTP/1.1
Ocp-Apim-Subscription-Key: 123456789ABCDE
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0; Trident/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 822)
X-Search-ClientIP: 999.999.999.999
X-Search-Location:  47.60357;long:-122.3295;re:100
X-MSEdge-ClientID: <blobFromPriorResponseGoesHere>
Host: api.cognitive.microsoft.com
```

The following is the response to the above request. Bing returns the top two answers, webpages and images, and promotes videos into the answer.

```
{
    "_type" : "SearchResponse",
    "queryContext" : {
        "originalQuery" : "sailiing dinghies"
    },
    "webPages" : {...},
    "images" : {...},
    "videos" : {...},
    "rankingResponse" : {...}
}
```

If you set `promote` to news, the response doesn't include the news answer because it is not a ranked answer—you can promote only ranked answers.

The answers that you want to promote do not count against the `answerCount` limit. For example, if the ranked answers are news, images, and videos, and you set `answerCount` to 1 and `promote` to news, the response contains news and images. Or, if the ranked answers are videos, images, and news, the response contains videos and news.

You may use `promote` only if you specify the `answerCount` query parameter.

Develop Solutions using Custom Speech Service

Custom Speech Service overview

Custom Speech Service is a cloud-based service that provides users with the ability to customize speech models for Speech-to-Text transcription. To use the Custom Speech Service, refer to the **Custom Speech Service Portal**[14].

The Custom Speech Service enables you to create customized language models and acoustic models tailored to your application and your users. By uploading your specific speech and/or text data to the Custom Speech Service, you can create custom models that can be used in conjunction with Microsoft's existing state-of-the-art speech models.

For example, if you're adding voice interaction to a mobile phone, tablet or PC app, you can create a custom language model that can be combined with Microsoft's acoustic model to create a speech-to-text endpoint designed especially for your app. If your application is designed for use in a particular environment or by a particular user population, you can also create and deploy a custom acoustic model with this service.

How do speech recognition systems work?

Speech recognition systems are composed of several components that work together. Two of the most important components are the acoustic model and the language model.

The acoustic model is a classifier that labels short fragments of audio into one of a number of phonemes, or sound units, in a given language. For example, the word "speech" is composed of four phonemes "s p iy ch". These classifications are made on the order of 100 times per second.

The language model is a probability distribution over sequences of words. The language model helps the system decide among sequences of words that sound similar, based on the likelihood of the word sequences themselves. For example, "recognize speech" and "wreck a nice beach" sound alike but the first hypothesis is far more likely to occur, and therefore will be assigned a higher score by the language model.

Both the acoustic and language models are statistical models learned from training data. As a result, they perform best when the speech they encounter when used in applications is similar to the data observed during training. The acoustic and language models in the Microsoft Speech-To-Text engine have been trained on an enormous collection of speech and text and provide state-of-the-art performance for the most common usage scenarios, such as interacting with Cortana on your smart phone, tablet or PC, searching the web by voice or dictating text messages to a friend.

Why use the Custom Speech Service?

While the Microsoft Speech-To-Text engine is world-class, it is targeted toward the scenarios described above. However, if you expect voice queries to your application to contain particular vocabulary items, such as product names or jargon that rarely occur in typical speech, it is likely that you can obtain improved performance by customizing the language model.

14 https://cris.ai/

For example, if you were building an app to search MSDN by voice, it's likely that terms like "object-oriented" or "namespace" or "dot net" will appear more frequently than in typical voice applications. Customizing the language model will enable the system to learn this.

Create a custom acoustic model

Creating a custom acoustic model is helpful if your application is designed for use in a particular environment, such as a noisy factory, or by a particular user population.

In this walkthrough, you learn how to:

- Prepare the data
- Import the acoustic data set
- Create the custom acoustic model

If you don't have a Cognitive Services account, create a **free account**[15] before you begin.

Prerequisites

Ensure that your Cognitive Services account is connected to a subscription by opening the **Cognitive Services Subscriptions**[16] page.

If no subscriptions are listed, you can either have Cognitive Services create an account for you by clicking the **Get free subscription** button. Or you can connect to a Custom Search Service subscription created in the Azure portal by clicking the **Connect existing subscription** button.

Prepare the data

To customize the acoustic model to a particular domain, a collection of speech data is required. This collection consists of a set of audio files of speech data, and a text file of transcriptions of each audio file. The audio data should be representative of the scenario in which you would like to use the recognizer.

For example:

- If you would like to better recognize speech in a noisy factory environment, the audio files should consist of people speaking in a noisy factory.
- If you are interested in optimizing performance for a single speaker, for example, you would like to transcribe all of FDR's Fireside Chats, then the audio files should consist of many examples of that speaker only.

An acoustic data set for customizing the acoustic model consists of two parts: (1) a set of audio files containing the speech data and (2) a file containing the transcriptions of all audio files.

Audio data recommendations

- All audio files in the data set should be stored in the WAV (RIFF) audio format.
- The audio must have a sampling rate of 8 kHz or 16 kHz and the sample values should be stored as uncompressed PCM 16-bit signed integers (shorts).
- Only single channel (mono) audio files are supported.

15 https://cris.ai/
16 https://cris.ai/Subscriptions

- The audio files must be between 100 ms and 1 minute in length. Each audio file should ideally start and end with at least 100ms of silence, and somewhere between 500ms and 1 second is common.
- If you have background noise in your data, it is recommended to also have some examples with longer segments of silence, for example, a few seconds, in your data, before and/or after the speech content.
- Each audio file should consist of a single utterance, for example, a single sentence for dictation, a single query, or a single turn of a dialog system.
- Each audio file to in the data set should have a unique filename and the extension "wav".
- The set of audio files should be placed in a single folder without subdirectories and the entire set of audio files should be packaged as a single ZIP file archive.

Note: Data imports via the web portal are currently limited to 2 GB, so this is the maximum size of an acoustic data set. This corresponds to approximately 17 hours of audio recorded at 16 kHz or 34 hours of audio recorded at 8 kHz. The main requirements for the audio data are summarized in the following table.

Property	Value
File Format	RIFF (WAV)
Sampling Rate	8000 Hz or 16000 Hz
Channels	1 (mono)
Sample Format	PCM, 16-bit integers
File Duration	0.1 seconds < duration < 60 seconds
Silence Collar	> 0.1 seconds
Archive Format	Zip
Maximum Archive Size	2 GB

Transcriptions

The transcriptions for all WAV files should be contained in a single plain-text file. Each line of the transcription file should have the name of one of the audio files, followed by the corresponding transcription. The file name and transcription should be separated by a tab (\t).

For example:

```
speech01.wav  speech recognition is awesome

speech02.wav  the quick brown fox jumped all over the place

speech03.wav  the lazy dog was not amused
```

The transcriptions will be text-normalized so they can be processed by the system. However, there are some important normalizations that must be done by the user prior to uploading the data to the Custom Speech Service. Consult the section on **transcription guidelines**[17] for the appropriate language when preparing your transcriptions.

The following steps are done using the **Custom Speech Service Portal**[18].

17 https://docs.microsoft.com/en-us/azure/cognitive-services/custom-speech-service/customspeech-how-to-topics/cognitive-services-custom-speech-transcription-guidelines

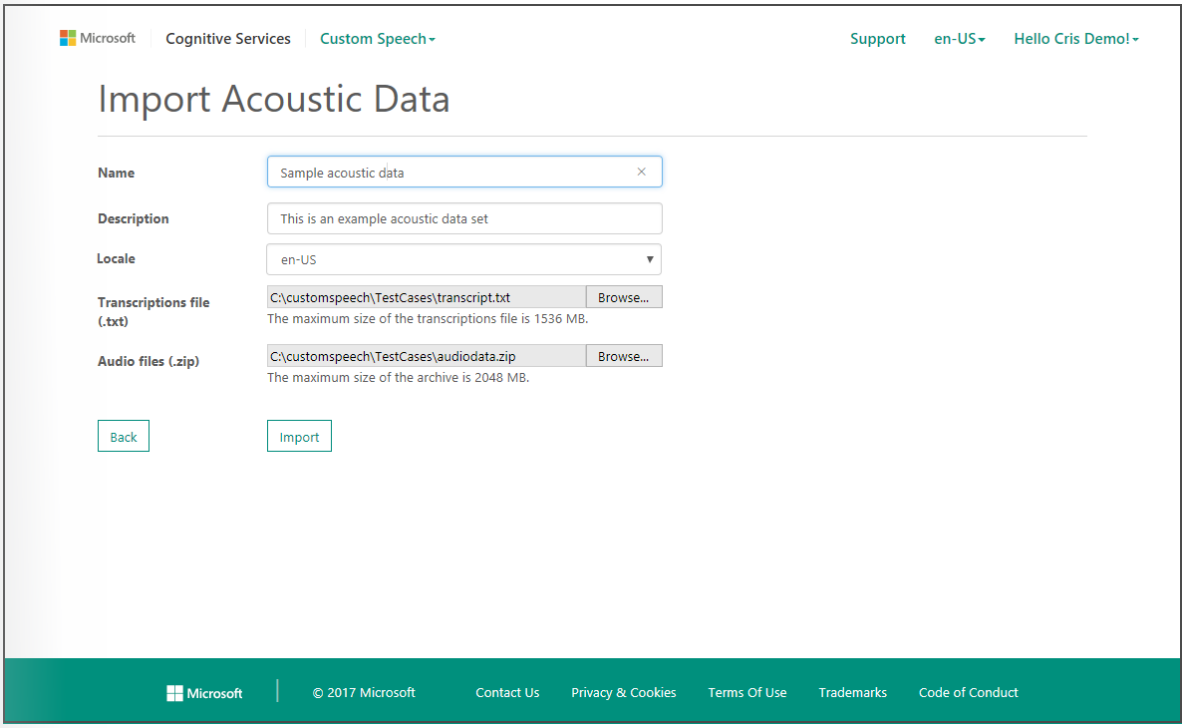
18 https://cris.ai/

Import the acoustic data set

Once the audio files and transcriptions have been prepared, they are ready to be imported to the service web portal.

To do so, first ensure you are signed into the Custom Speech Service Portal. Then click the "Custom Speech" drop-down menu on the top ribbon and select "Adaptation Data". If this is your first time uploading data to the Custom Speech Service, you will see an empty table called "Datasets".

Click the "Import" button in the "Acoustic Datasets" row, and the site displays a page for uploading a new data set.

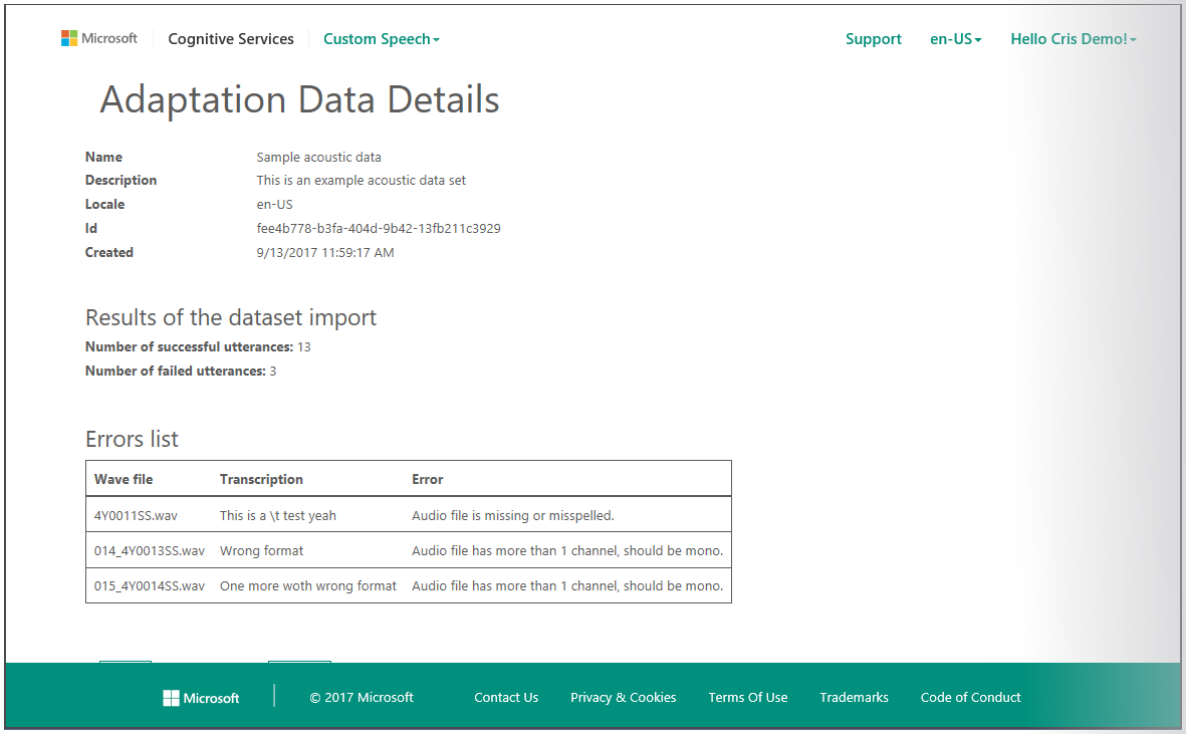


Enter a *Name* and *Description* in the appropriate text boxes. These are useful for keeping track of various data sets you upload. Next, click "Choose File" for the "Transcription File" and "WAV files" and select your plaint-text transcription file and zip archive of WAV files, respectively. When this is complete, click "Import" to upload your data. Your data will then be uploaded. For larger data sets, this may take several minutes.

When the upload is complete, you will return to the "Acoustic Datasets" table and will see an entry that corresponds to your acoustic data set. Notice that it has been assigned a unique id (GUID). The data will also have a status that reflects its current state. Its status will be "Waiting" while it is being queued for processing, "Processing" while it is going through validation, and "Complete" when the data is ready for use.

Data validation includes a series of checks on the audio files to verify the file format, length, and sampling rate, and on the transcription files to verify the file format and perform some text normalization.

When the status is "Complete", you can click "Details" to see the acoustic data verification report. The number of utterances that passed and failed verification will be shown, along with details about the failed utterances. In the example below, two WAV files failed verification because of improper audio format (in this data set, one had an incorrect sampling rate and one was the incorrect file format).

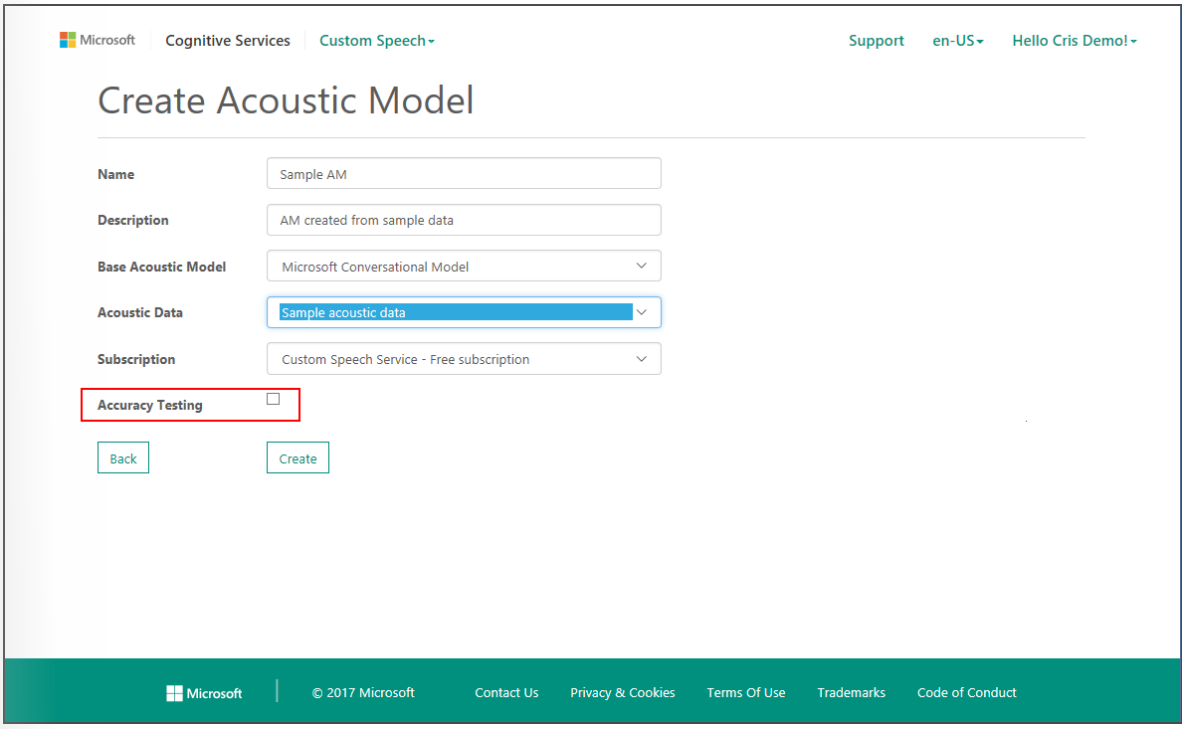


At some point, if you would like to change the Name or Description of the data set, you can click the "Edit" link and change these entries. Note that you cannot modify the audio files or transcriptions.

Once the status of your acoustic data set is "Complete", it can be used to create a custom acoustic model. To do so, click "Acoustic Models" in the "Custom Speech" drop-down menu. You will see a table called "Your models" that lists all of your custom acoustic models. This table will be empty if this is your first use. The current locale is shown in the table title. Currently, acoustic models can be created for US English only.

To create a new model, click "Create New" under the table title. As before, enter a name and description to help you identify this model. For example, the "Description"" field can be used to record which starting model and acoustic data set were used to create the model. Next, select a "Base Acoustic Model" from the drop-down menu. The base model is the model that is the starting point for your customization. There are two base acoustic models to choose from. The Microsoft Search and Dictation AM is appropriate for speech directed at an application, such as commands, search queries, or dictation. The Microsoft Conversational model is appropriate for recognizing speech spoken in a conversational style. This type of speech is typically directed at another person and occurs in call center or meetings. Note that latency for partial results in Conversational models is higher than in Search and Dictation models.

Next, select the acoustic data you wish to use to perform the customization using the drop-down menu.

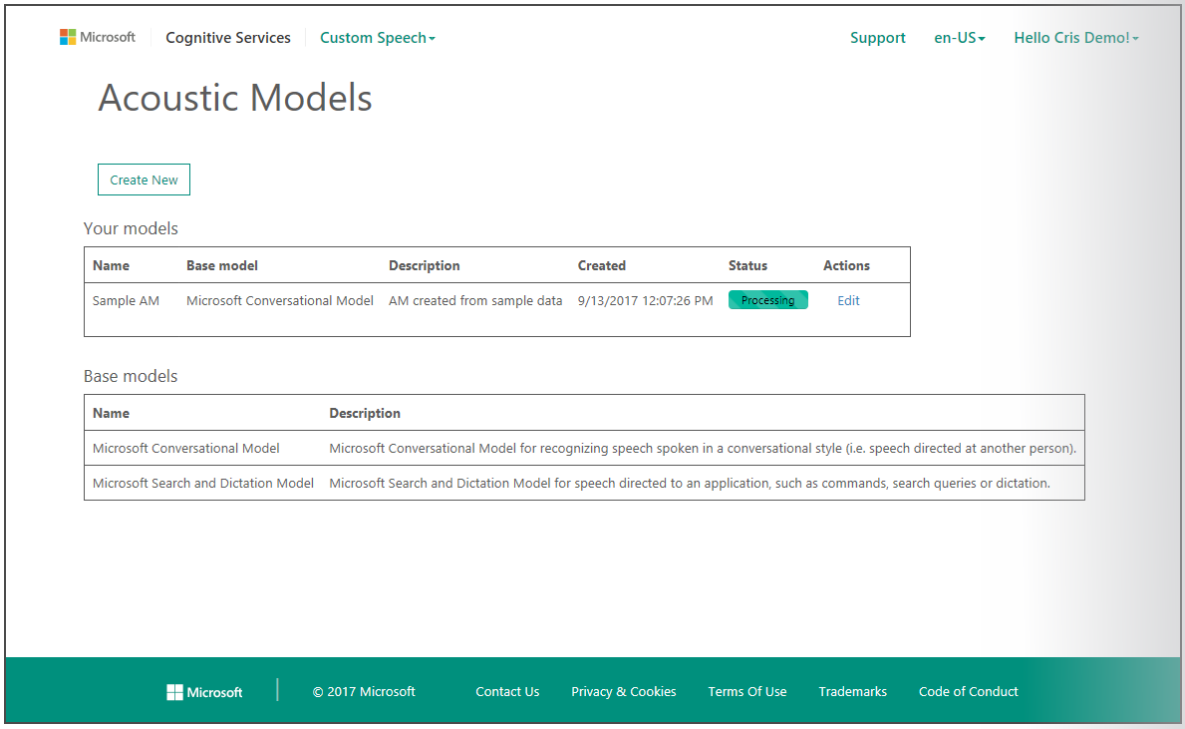


You can optionally choose to perform offline testing of your new model when the processing is complete. This will run a speech-to-text evaluation on a specified acoustic data set using the customized acoustic model and report the results. To perform this testing, select the "Accuracy Testing" check box. Then select a language model from the drop-down menu. If you have not created any custom language models, only the base language models will be in the drop-down list. See the description of the base language models in the guide and select the one that is most appropriate.

Finally, select the acoustic data set you would like to use to evaluate the custom model. If you perform accuracy testing, it is important to select an acoustic data that is different from the one used for the model creation to get a realistic sense of the model's performance. Also note that offline testing is limited to 1000 utterances. If the acoustic dataset for testing is larger than that, only the first 1000 utterances will be evaluated.

When you are ready to start running the customization process, press "Create".

You will now see a new entry in the acoustic models table corresponding to this new model. The status of the process is reflected in the table. The status states are "Waiting", "Processing" and "Complete".



Create a custom language model

In this lesson, you create a custom language model for text queries or utterances you expect users to say or type in an application. You can then use this custom language model in conjunction with existing state-of-the-art speech models from Microsoft to add voice interaction to your application.

In this lesson, you learn how to:

- Prepare the data
- Import the language data set
- Create the custom language model

If you don't have a Cognitive Services account, create a **free account**[19] before you begin.

Prerequisites

Ensure that your Cognitive Services account is connected to a subscription by opening the **Cognitive Services Subscriptions**[20] page.

If no subscriptions are listed, you can either have Cognitive Services create an account for you by clicking the **Get free subscription** button. Or you can connect to a Custom Search Service subscription created in the Azure portal by clicking the **Connect existing subscription** button.

19 https://cris.ai/
20 https://cris.ai/Subscriptions

Prepare the data

In order to create a custom language model for your application, you need to provide a list of example utterances to the system, for example:

- "He has had urticaria for the past week."
- "The patient had a well-healed herniorrhaphy scar."

The sentences do not need to be complete sentences or grammatically correct, and should accurately reflect the spoken input you expect the system to encounter in deployment. These examples should reflect both the style and content of the task the users will perform with your application.

The language model data should be written in plain-text file using either the US-ASCII or UTF-8, depending on the locale. For en-US, both encodings are supported. For zh-CN, only UTF-8 is supported (BOM is optional). The text file should contain one example (sentence, utterance, or query) per line.

If you wish some sentences to have a higher weight (importance), you can add it several times to your data. A good number of repetitions is between 10 - 100. If you normalize it to 100, you can weight sentence relative to this easily.

The main requirements for the language data are summarized in the following table.

Property	Value
Text Encoding	en-US: US-ACSII or UTF-8 or zh-CN: UTF-8
# of Utterances per line	1
Maximum File Size	200 MB
Remarks	avoid repeating characters more often than 4 times, for example 'aaaaa'
Remarks	no special characters like '\t' or any other UTF-8 character above U+00A1 in **Unicode characters table** *(http://www.utf8-chartable.de/)*
Remarks	URIs will also be rejected since there is no unique way to pronounce a URI

When the text is imported, it is text-normalized so it can be processed by the system. However, there are some important normalizations that must be done by the user prior to uploading the data. See the Transcription guidelines to determine appropriate language when preparing your language data.

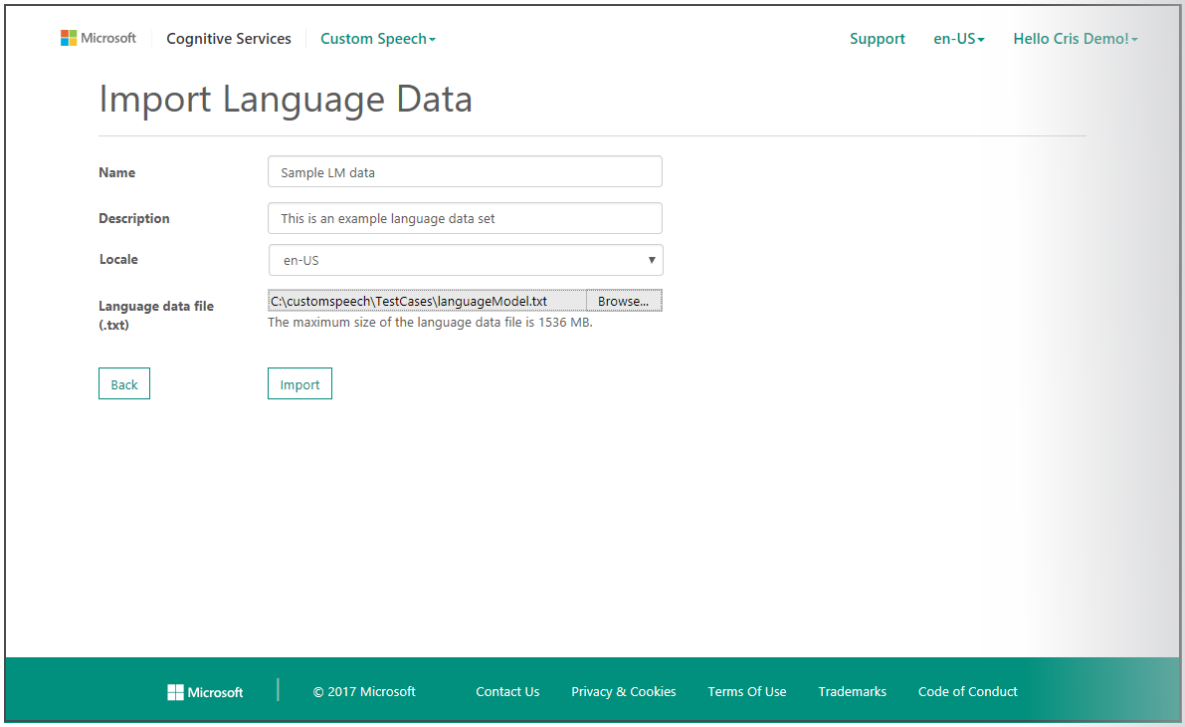
Import the language data set

Click the "Import" button in the "Acoustic Datasets" row, and the site displays a page for uploading a new data set.

When you are ready to import your language data set, log into the **Custom Speech Service Portal**[21]. Then click the "Custom Speech" drop-down menu on the top ribbon and select "Adaptation Data". If this is your first time uploading data to the Custom Speech Service, you will see an empty table called "Datasets".

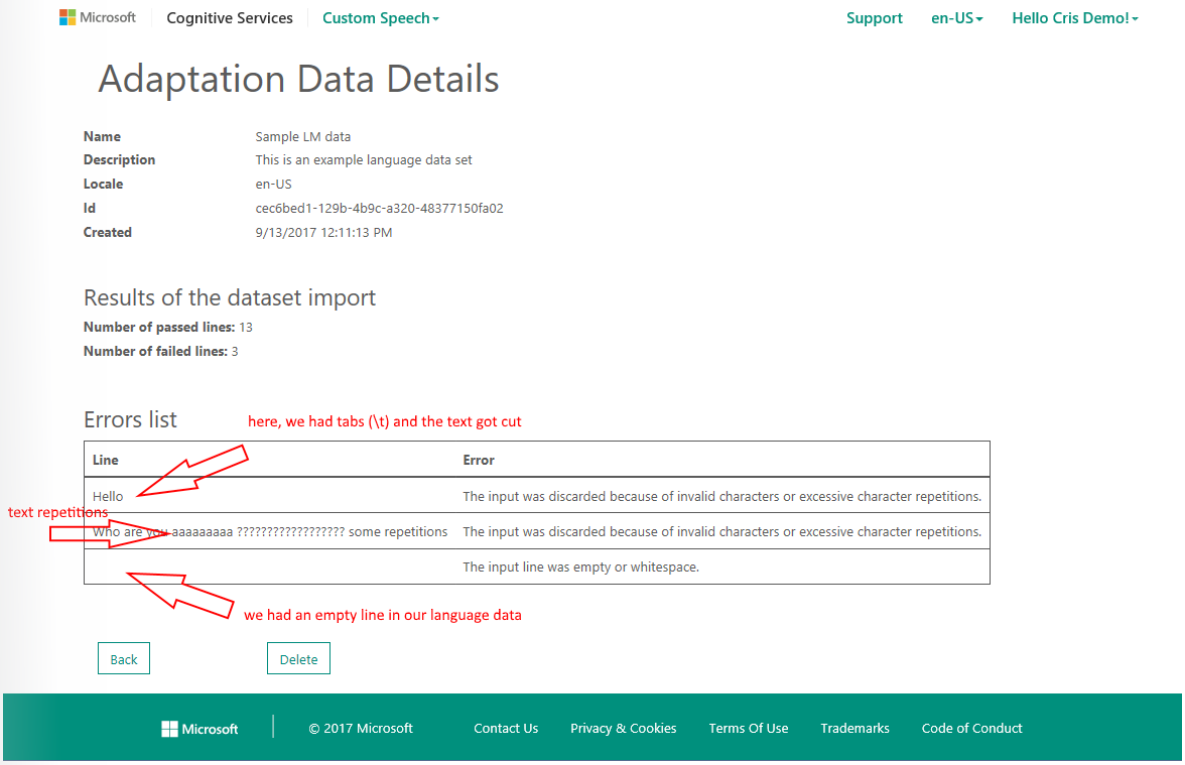
To import a new data set, click the "Import" button in the "Language Datasets" row, and the site displays a page for uploading a new data set. Enter a Name and Description to help you identify the data set in the future. Next, use the "Choose File" button to locate the text file of language data. After that, click "Import" and the data set will be uploaded. Depending on the size of the data set, this may take several minutes.

21 https://cris.ai/

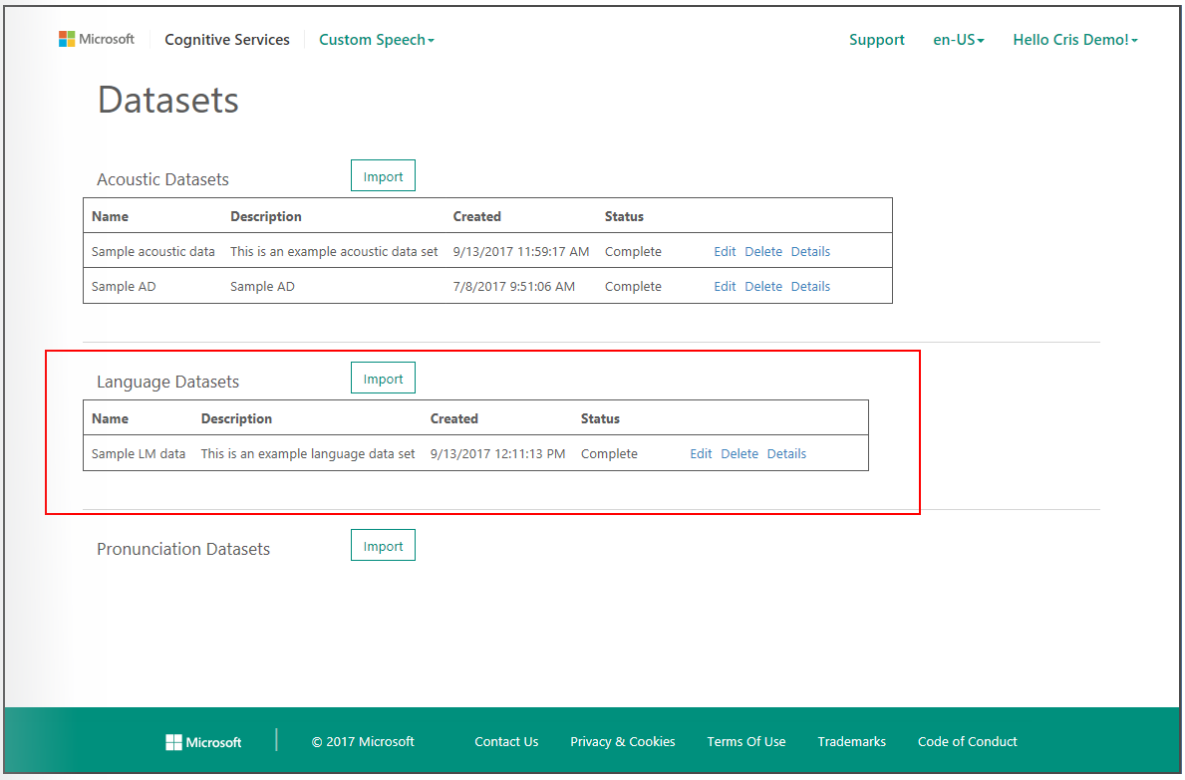


When the import is complete, you will return to the language data table and will see an entry that corresponds to your language data set. Notice that it has been assigned a unique id (GUID). The data will also have a status that reflects its current state. Its status will be "Waiting" while it is being queued for processing, "Processing" while it is going through validation, and "Complete" when the data is ready for use. Data validation performs a series of checks on the text in the file and some text normalization of the data.

When the status is "Complete", you can click "View Report" to see the language data verification report. The number of utterances that passed and failed verification are shown, along with details about the failed utterances. In the example below, two examples failed verification because of improper characters (in this data set, the first had two emoticons and the second had several characters outside of the ASCII printable character set).



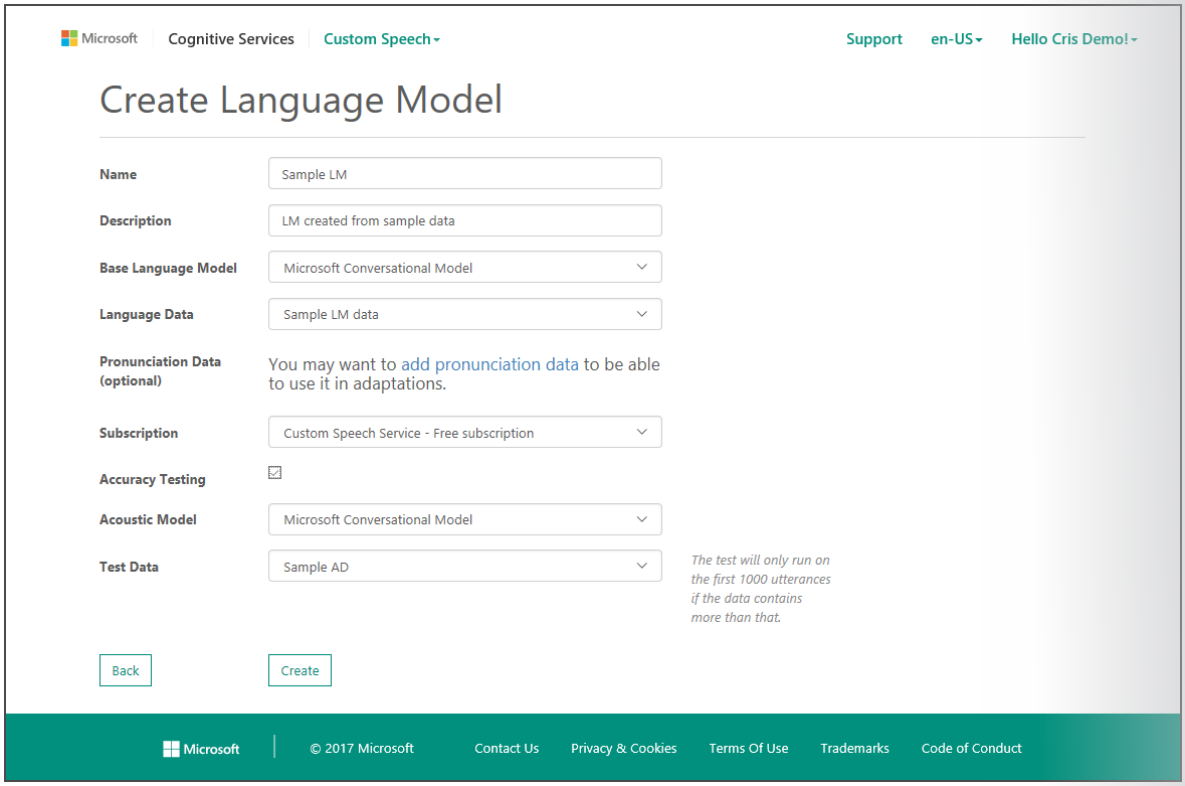
When the status of the language data set is "Complete", it can be used to create a custom language model.



Once your language data is ready, click “Language Models” from the “Menu” drop-down menu to start the process of custom language model creation. This page contains a table called “Language Models” with your current custom language models. If you have not yet created any custom language models, the table will be empty. The current locale is reflected in the table title. If you would like to create a language model for a different language, click on “Change Locale”. Additional information on supported languages can be found in the section on **Changing Locale**[22]. To create a new model, click the “Create New” link below the table title.

On the “Create Language Model” page, enter a "Name" and “Description” to help you keep track of pertinent information about this model, such as the data set used. Next, select the “Base Language Model” from the drop-down menu. This model will be the starting point for your customization. There are two base language models to choose from. The Microsoft Search and Dictation LM is appropriate for speech directed at an application, such as commands, search queries, or dictation. The Microsoft Conversational LM is appropriate for recognizing speech spoken in a conversational style. This type of speech is typically directed at another person and occurs in call centers or meetings.

After you have specified the base language model, select the language data set you wish to use for the customization using the “Language Data” drop-down menu



As with the acoustic model creation, you can optionally choose to perform offline testing of your new model when the processing is complete. Because this is an evaluation of the speech-to-text performance, offline testing requires an acoustic data set.

To perform offline testing of your language model, select the check box next to “Offline Testing”. Then select an acoustic model from the drop-down menu. If you have not created any custom acoustic models, the Microsoft base acoustic models will be the only model in the menu. In case you have picked a

22 https://docs.microsoft.com/en-us/azure/cognitive-services/custom-speech-service/customspeech-how-to-topics/cognitive-services-custom-speech-change-locale

conversational LM base model, you need to use a conversational AM here. In case you use a search and dictate LM model, you have to select a search and dictate AM model.

Finally, select the acoustic data set you would like to use to perform the evaluation.

When you are ready to start processing, press “Create”. This will return you to the table of language models. There will be a new entry in the table corresponding to this model. The status reflects the model’s state and will go through several states including “Waiting”, “Processing”, and “Complete”.

When the model has reached the “Complete” state, it can be deployed to an endpoint. Clicking on “View Result” will show the results of offline testing, if performed.

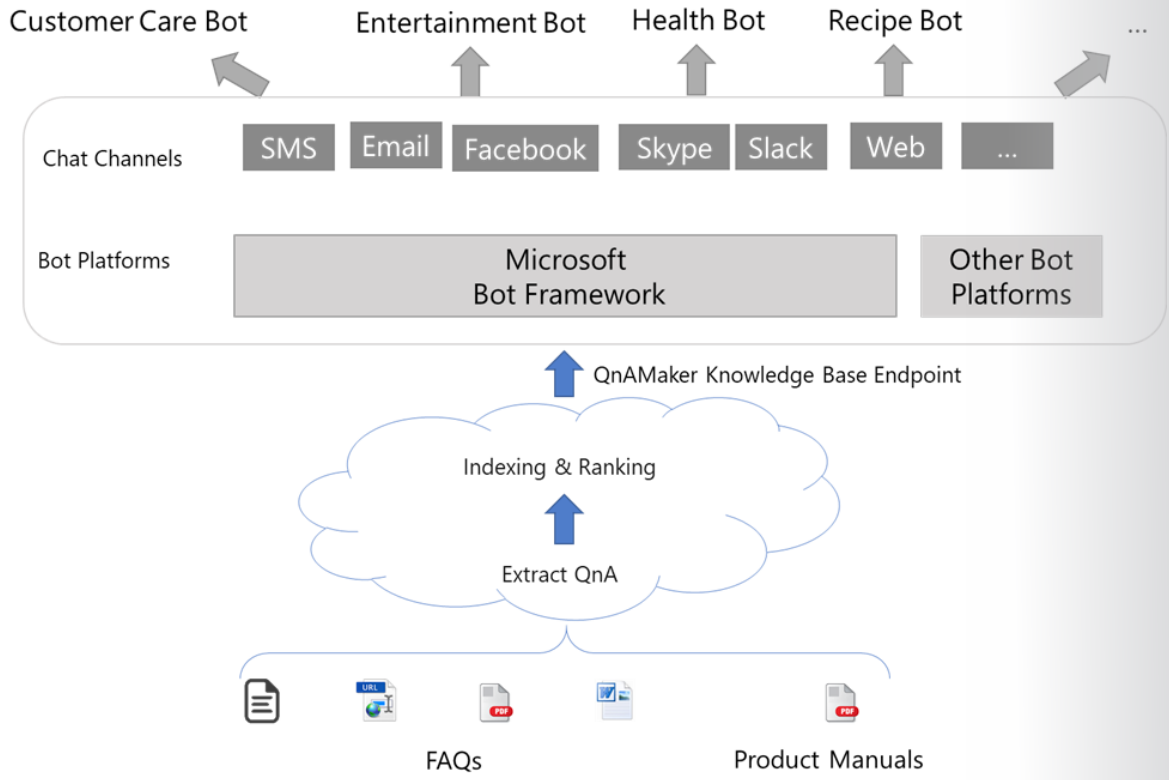
If you would like to change the “Name” or "Description" of the model at some point, you can use the “Edit” link in the appropriate row of the language models table.

Develop Solutions using QnA Maker

QnA Maker overview

QnA Maker enables you to power a question and answer service from your semi-structured content like FAQ (Frequently Asked Questions) documents or URLs and product manuals. You can build a model of questions and answers that is flexible to user queries, providing responses that you'll train a bot to use in a natural, conversational way.

An easy-to-use graphical user interface enables you to create, manage, train and get your service up and running without any developer experience.



Create a new knowledge base in C#

This walkthrough walks you through creating a sample QnA maker knowledge base, programmatically, that will appear in your Azure Dashboard of your Cognitive Services API account.

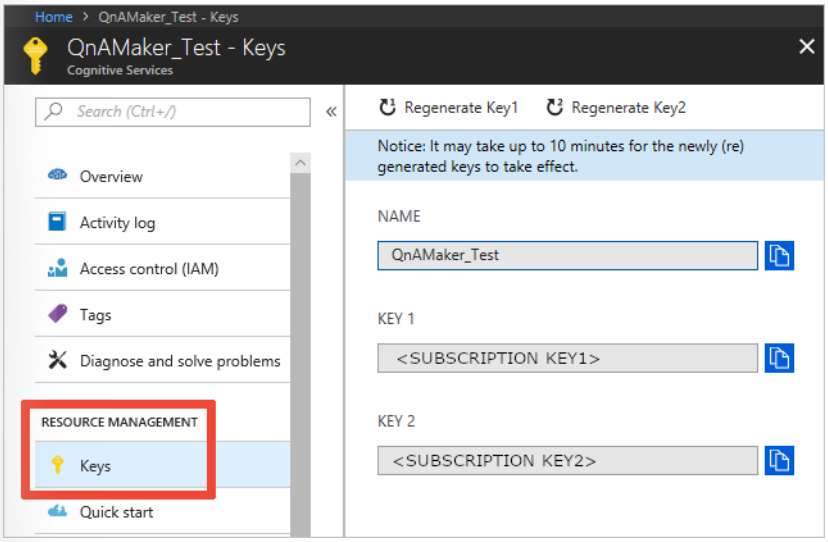
Two sample FAQ URLs are given below ('urls' in the string kb). QnA Maker automatically extracts questions and answers from semi-structured content, like FAQs, as explained more in this **data sources**[23] document. You may also use your own FAQ URLs in this quickstart.

Prerequisites

If your preferred IDE is Visual Studio, you'll need Visual Studio 2017 to run this code sample on Windows. (The free Community Edition will work.)

23 https://docs.microsoft.com/en-us/azure/cognitive-services/QnAMaker/concepts/data-sources-supported

You must have a Cognitive Services API account with **QnA Maker** chosen as your resource. You'll need a paid subscription key from your new API account in your Azure dashboard. To retrieve your key, select **Keys** under **Resource Management** in your dashboard. Either key will work for this quickstart.



Create knowledge base

The following code creates a new knowledge base, using the **Create**[24] method.

1. Create a new .NET Framework C# console application in your preferred IDE.
2. Add the code provided below.
3. Replace the `key` value with your valid subscription key.
4. Run the program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

// NOTE: Install the Newtonsoft.Json NuGet package.
using Newtonsoft.Json;

namespace QnAMaker
{
    class Program
    {
        // Represents the various elements used to create HTTP request URIs
        // for QnA Maker operations.
        static string host = "https://westus.api.cognitive.microsoft.com";
```

24 https://westus.dev.cognitive.microsoft.com/docs/services/5a93fcf85b4ccd136866eb37/operations/5ac266295b4ccd1554da75ff

```
        static string service = "/qnamaker/v4.0";
        static string method = "/knowledgebases/create";

        // NOTE: Replace this value with a valid QnA Maker subscription
key.
        static string key = "YOUR SUBSCRIPTION KEY HERE";

        /// <summary>
        /// Defines the data source used to create the knowledge base.
        /// The data source includes a QnA pair, with metadata,
        /// the URL for the QnA Maker FAQ article, and
        /// the URL for the Azure Bot Service FAQ article.
        /// </summary>
        static string kb = @"
{
  'name': 'QnA Maker FAQ',
  'qnaList': [
    {
      'id': 0,
      'answer': 'You can use our REST APIs to manage your knowledge base.
See here for details: https://westus.dev.cognitive.microsoft.com/docs/
services/58994a073d9e04097c7ba6fe/operations/58994a073d9e041ad42d9baa',
      'source': 'Custom Editorial',
      'questions': [
        'How do I programmatically update my knowledge base?'
      ],
      'metadata': [
        {
          'name': 'category',
          'value': 'api'
        }
      ]
    }
  ],
  'urls': [
    'https://docs.microsoft.com/en-in/azure/cognitive-services/qnamaker/
faqs',
    'https://docs.microsoft.com/en-us/bot-framework/resources-bot-frame-
work-faq'
  ],
  'files': []
}
";

        /// <summary>
        /// Represents the HTTP response returned by an HTTP request.
        /// </summary>
        public struct Response
        {
            public HttpResponseHeaders headers;
            public string response;
```

```
            public Response(HttpResponseHeaders headers, string response)
            {
                this.headers = headers;
                this.response = response;
            }
        }

        /// <summary>
        /// Formats and indents JSON for display.
        /// </summary>
        /// <param name="s">The JSON to format and indent.</param>
        /// <returns>A string containing formatted and indented JSON.</
returns>
        static string PrettyPrint(string s)
        {
            return JsonConvert.SerializeObject(JsonConvert.DeserializeOb-
ject(s), Formatting.Indented);
        }

        /// <summary>
        /// Asynchronously sends a POST HTTP request.
        /// </summary>
        /// <param name="uri">The URI of the HTTP request.</param>
        /// <param name="body">The body of the HTTP request.</param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        async static Task<Response> Post(string uri, string body)
        {
            using (var client = new HttpClient())
            using (var request = new HttpRequestMessage())
            {
                request.Method = HttpMethod.Post;
                request.RequestUri = new Uri(uri);
                request.Content = new StringContent(body, Encoding.UTF8,
"application/json");
                request.Headers.Add("Ocp-Apim-Subscription-Key", key);

                var response = await client.SendAsync(request);
                var responseBody = await response.Content.ReadAsStrin-
gAsync();
                return new Response(response.Headers, responseBody);
            }
        }

        /// <summary>
        /// Asynchronously sends a GET HTTP request.
        /// </summary>
        /// <param name="uri">The URI of the HTTP request.</param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
```

```
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        async static Task<Response> Get(string uri)
        {
            using (var client = new HttpClient())
            using (var request = new HttpRequestMessage())
            {
                request.Method = HttpMethod.Get;
                request.RequestUri = new Uri(uri);
                request.Headers.Add("Ocp-Apim-Subscription-Key", key);

                var response = await client.SendAsync(request);
                var responseBody = await response.Content.ReadAsStrin-
gAsync();
                return new Response(response.Headers, responseBody);
            }
        }

        /// <summary>
        /// Creates a knowledge base.
        /// </summary>
        /// <param name="kb">The data source for the knowledge base.</
param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        /// <remarks>The method constructs the URI to create a knowledge
base in QnA Maker, and then
        /// asynchronously invokes the <see cref="QnAMaker.Program.
Post(string, string)"/> method
        /// to send the HTTP request.</remarks>
        async static Task<Response> PostCreateKB(string kb)
        {
            // Builds the HTTP request URI.
            string uri = host + service + method;

            // Writes the HTTP request URI to the console, for display
purposes.
            Console.WriteLine("Calling " + uri + ".");

            // Asynchronously invokes the Post(string, string) method,
using the
            // HTTP request URI and the specified data source.
            return await Post(uri, kb);
        }

        /// <summary>
        /// Gets the status of the specified QnA Maker operation.
        /// </summary>
        /// <param name="operation">The QnA Maker operation to check.</
param>
```

```
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        /// <remarks>The method constructs the URI to get the status of a
QnA Maker operation, and
        /// then asynchronously invokes the <see cref="QnAMaker.Program.
Get(string)"/> method
        /// to send the HTTP request.</remarks>
        async static Task<Response> GetStatus(string operation)
        {
            // Builds the HTTP request URI.
            string uri = host + service + operation;

            // Writes the HTTP request URI to the console, for display
purposes.
            Console.WriteLine("Calling " + uri + ".");

            // Asynchronously invokes the Get(string) method, using the
            // HTTP request URI.
            return await Get(uri);
        }

        /// <summary>
        /// Creates a knowledge base, periodically checking status
        /// until the knowledge base is created.
        /// </summary>
        async static void CreateKB()
        {
            try
            {
                // Starts the QnA Maker operation to create the knowledge
base.
                var response = await PostCreateKB(kb);

                // Retrieves the operation ID, so the operation's status
can be
                // checked periodically.
                var operation = response.headers.GetValues("Location").
First();

                // Displays the JSON in the HTTP response returned by the
                // PostCreateKB(string) method.
                Console.WriteLine(PrettyPrint(response.response));

                // Iteratively gets the state of the operation creating the
                // knowledge base. Once the operation state is set to
something other
                // than "Running" or "NotStarted", the loop ends.
                var done = false;
                while (true != done)
                {
```

```
                    // Gets the status of the operation.
                    response = await GetStatus(operation);

                    // Displays the JSON in the HTTP response returned by
the
                    // GetStatus(string) method.
                    Console.WriteLine(PrettyPrint(response.response));

                    // Deserialize the JSON into key-value pairs, to re-
trieve the
                    // state of the operation.
                    var fields = JsonConvert.DeserializeObject<Diction-
ary<string, string>>(response.response);

                    // Gets and checks the state of the operation.
                    String state = fields["operationState"];
                    if (state.CompareTo("Running") == 0 || state.Compare-
To("NotStarted") == 0)
                    {
                        // QnA Maker is still creating the knowledge base.
The thread is
                        // paused for a number of seconds equal to the
Retry-After header value,
                        // and then the loop continues.
                        var wait = response.headers.GetValues("Retry-Af-
ter").First();
                        Console.WriteLine("Waiting " + wait + " sec-
onds...");
                        Thread.Sleep(Int32.Parse(wait) * 1000);
                    }
                    else
                    {
                        // QnA Maker has completed creating the knowledge
base.
                        done = true;
                    }
                }
            }
            catch
            {
                // An error occurred while creating the knowledge base.
Ensure that
                // you included your QnA Maker subscription key where
directed in the sample.
                Console.WriteLine("An error occurred while creating the
knowledge base.");
            }
            finally
            {
                Console.WriteLine("Press any key to continue.");
            }
```

```
        }

        static void Main(string[] args)
        {
            // Invoke the CreateKB() method to create a knowledge base,
periodically
            // checking the status of the QnA Maker operation until the
            // knowledge base is created.
            CreateKB();

            // The console waits for a key to be pressed before closing.
            Console.ReadLine();
        }
    }
}
```

Understand what QnA Maker returns

A successful response is returned in JSON, as shown in the following example. Your results may differ slightly. If the final call returns a "Succeeded" state... your knowledge base was created successfully. To troubleshoot refer to the Get Operation Details of the **QnA Maker API**[25].

```
Calling https://westus.api.cognitive.microsoft.com/qnamaker/v4.0/knowledge-
bases/create.
{
  "operationState": "NotStarted",
  "createdTimestamp": "2018-06-25T10:30:15Z",
  "lastActionTimestamp": "2018-06-25T10:30:15Z",
  "userId": "0d85ec291c204197a70cfec51725cd22",
  "operationId": "d9d40918-01bd-49f4-88b4-129fbc434c94"
}
Calling https://westus.api.cognitive.microsoft.com/qnamaker/v4.0/opera-
tions/d9d40918-01bd-49f4-88b4-129fbc434c94.
{
  "operationState": "Running",
  "createdTimestamp": "2018-06-25T10:30:15Z",
  "lastActionTimestamp": "2018-06-25T10:30:15Z",
  "userId": "0d85ec291c184197a70cfeb51025cd22",
  "operationId": "d9d40918-01bd-49f4-88b4-129fbc434c94"
}
Waiting 30 seconds...
Calling https://westus.api.cognitive.microsoft.com/qnamaker/v4.0/opera-
tions/d9d40918-01bd-49f4-88b4-129fbc434c94.
{
  "operationState": "Running",
  "createdTimestamp": "2018-06-25T10:30:15Z",
  "lastActionTimestamp": "2018-06-25T10:30:15Z",
  "userId": "0d85ec221c284197a70gfeb51725cd22",
```

25 https://westus.dev.cognitive.microsoft.com/docs/services/5a93fcf85b4ccd136866eb37/operations/operations_getoperationdetails

```
  "operationId": "d9d40918-01bd-49f4-88b4-129fbc434c94"
}
Waiting 30 seconds...
Calling https://westus.api.cognitive.microsoft.com/qnamaker/v4.0/opera-
tions/d9d40918-01bd-49f4-88b4-129fbc434c94.
{
  "operationState": "Succeeded",
  "createdTimestamp": "2018-06-25T10:30:15Z",
  "lastActionTimestamp": "2018-06-25T10:30:51Z",
  "resourceLocation": "/knowledgebases/1d9eb2a1-de2a-4709-91b2-f6ea8afb-
6fb9",
  "userId": "0d85ec294c284197a70cfeb51775cd22",
  "operationId": "d9d40918-01bd-49f4-88b4-129fbc434c94"
}
Press any key to continue.
```

Once your knowledge base is created, you can view it in your QnA Maker Portal, **My knowledge bases**[26] page. Select your knowledge base name, for example QnA Maker FAQ, to view.

Update a knowledge base in C#

The following code updates an existing knowledge base, using the **Update**[27] method.

1. Create a new .NET Framework C# console application in your favorite IDE.
2. Add the code provided below.
3. Replace the `key` value with a valid subscription key.
4. Replace the `kb` value with a valid knowledge base ID. Find this value by going to one of your **QnA Maker knowledge bases**[28]. Select the knowledge base you want to update. Once on that page, find the 'kdid=' in the URL as shown below. Use its value for your code sample.

5.
6. Run the program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

// NOTE: Install the Newtonsoft.Json NuGet package.
using Newtonsoft.Json;

namespace QnAMaker
```

26 https://www.qnamaker.ai/Home/MyServices
27 https://westus.dev.cognitive.microsoft.com/docs/services/5a93fcf85b4ccd136866eb37/operations/5ac266295b4ccd1554da7600
28 https://www.qnamaker.ai/Home/MyServices

```
{
    class Program
    {
        // Represents the various elements used to create HTTP request URIs
        // for QnA Maker operations.
        static string host = "https://westus.api.cognitive.microsoft.com";
        static string service = "/qnamaker/v4.0";
        static string method = "/knowledgebases/";

        // NOTE: Replace this with a valid subscription key.
        static string key = "ADD KEY HERE";

        // NOTE: Replace this with a valid knowledge base ID.
        static string kb = "ADD ID HERE";

        /// <summary>
        /// Defines the data source used to update the knowledge base.
        /// This JSON schema is based on your existing knowledge base.
        /// In the 'update' object, the existing name is changed.
        /// </summary>
        static string new_kb = @"
        {
          'add': {
            'qnaList': [
              {
                'id': 1,
                'answer': 'You can change the default message if you use
the QnAMakerDialog. See this for details: https://docs.botframework.com/
en-us/azure-bot-service/templates/qnamaker/#navtitle',
                'source': 'Custom Editorial',
                'questions': [
                  'How can I change the default message from QnA Maker?'
                ],
                'metadata': []
              }
            ],
            'urls': []
          },
          'update' : {
            'name' : 'New KB Name'
          },
          'delete': {
            'ids': [
              0
            ]
          }
        }
        ";
        /// <summary>
        /// Represents the HTTP response returned by an HTTP request.
        /// </summary>
```

```
        public struct Response
        {
            public HttpResponseHeaders headers;
            public string response;

            public Response(HttpResponseHeaders headers, string response)
            {
                this.headers = headers;
                this.response = response;
            }
        }

        /// <summary>
        /// Formats and indents JSON for display.
        /// </summary>
        /// <param name="s">The JSON to format and indent.</param>
        /// <returns>A string containing formatted and indented JSON.</
returns>
        static string PrettyPrint(string s)
        {
            return JsonConvert.SerializeObject(JsonConvert.DeserializeOb-
ject(s), Formatting.Indented);
        }

        /// <summary>
        /// Asynchronously sends a PATCH HTTP request.
        /// </summary>
        /// <param name="uri">The URI of the HTTP request.</param>
        /// <param name="body">The body of the HTTP request.</param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        async static Task<Response> Patch(string uri, string body)
        {
            using (var client = new HttpClient())
            using (var request = new HttpRequestMessage())
            {
                request.Method = new HttpMethod("PATCH");
                request.RequestUri = new Uri(uri);
                request.Content = new StringContent(body, Encoding.UTF8,
"application/json");
                request.Headers.Add("Ocp-Apim-Subscription-Key", key);

                var response = await client.SendAsync(request);
                var responseBody = await response.Content.ReadAsStrin-
gAsync();
                return new Response(response.Headers, responseBody);
            }
        }

        /// <summary>
```

```
        /// Asynchronously sends a GET HTTP request.
        /// </summary>
        /// <param name="uri">The URI of the HTTP request.</param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        async static Task<Response> Get(string uri)
        {
            using (var client = new HttpClient())
            using (var request = new HttpRequestMessage())
            {
                request.Method = HttpMethod.Get;
                request.RequestUri = new Uri(uri);
                request.Headers.Add("Ocp-Apim-Subscription-Key", key);

                var response = await client.SendAsync(request);
                var responseBody = await response.Content.ReadAsStrin-
gAsync();
                return new Response(response.Headers, responseBody);
            }
        }

        /// <summary>
        /// Updates a knowledge base.
        /// </summary>
        /// <param name="kb">The ID for the existing knowledge base.</
param>
        /// <param name="new_kb">The new data source for the updated knowl-
edge base.</param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
        /// object that represents the HTTP response."</returns>
        /// <remarks>Constructs the URI to update a knowledge base in QnA
Maker,
        /// then asynchronously invokes the <see cref="QnAMaker.Program.
Patch(string, string)"/>
        /// method to send the HTTP request.</remarks>
        async static Task<Response> PostUpdateKB(string kb, string new_kb)
        {
            string uri = host + service + method + kb;
            Console.WriteLine("Calling " + uri + ".");
            return await Patch(uri, new_kb);
        }

        /// <summary>
        /// Gets the status of the specified QnA Maker operation.
        /// </summary>
        /// <param name="operation">The QnA Maker operation to check.</
param>
        /// <returns>A <see cref="System.Threading.Tasks.Task{TResult}
(QnAMaker.Program.Response)"/>
```

```
        /// object that represents the HTTP response."</returns>
        /// <remarks>Constructs the URI to get the status of a QnA Maker
        /// operation, then asynchronously invokes the <see cref="QnAMaker.
Program.Get(string)"/>
        /// method to send the HTTP request.</remarks>
        async static Task<Response> GetStatus(string operation)
        {
            string uri = host + service + operation;
            Console.WriteLine("Calling " + uri + ".");
            return await Get(uri);
        }

        /// <summary>
        /// Updates a knowledge base, periodically checking status
        /// until the knowledge base is updated.
        /// </summary>
        async static void UpdateKB(string kb, string new_kb)
        {
            try
            {
                // Starts the QnA Maker operation to update the knowledge
base.
                var response = await PostUpdateKB(kb, new_kb);

                // Retrieves the operation ID, so the operation's status
can be
                // checked periodically.
                var operation = response.headers.GetValues("Location").
First();

                // Displays the JSON in the HTTP response returned by the
                // PostUpdateKB(string, string) method.
                Console.WriteLine(PrettyPrint(response.response));

                // Iteratively gets the state of the operation updating the
                // knowledge base. Once the operation state is something
other
                // than "Running" or "NotStarted", the loop ends.
                var done = false;
                while (true != done)
                {
                    // Gets the status of the operation.
                    response = await GetStatus(operation);
                    // Displays the JSON in the HTTP response returned by
the
                    // GetStatus(string) method.
                    Console.WriteLine(PrettyPrint(response.response));

                    // Deserialize the JSON into key-value pairs, to re-
trieve the
                    // state of the operation.
```

```
                    var fields = JsonConvert.DeserializeObject<Diction-
ary<string, string>>(response.response);

                    // Gets and checks the state of the operation.
                    String state = fields["operationState"];
                    if (state.CompareTo("Running") == 0 || state.Compare-
To("NotStarted") == 0)
                    {
                        // QnA Maker is still updating the knowledge base.
The thread is
                        // paused for a number of seconds equal to the
Retry-After
                        // header value, and then the loop continues.
                        var wait = response.headers.GetValues("Retry-Af-
ter").First();
                        Console.WriteLine("Waiting " + wait + " sec-
onds...");
                        Thread.Sleep(Int32.Parse(wait) * 1000);
                    }
                    else
                    {
                        // QnA Maker has completed updating the knowledge
base.
                        done = true;
                    }
                }
            }
            catch
            {
                // An error occurred while updating the knowledge base.
Ensure that
                // you included your QnA Maker subscription key and knowl-
edge base ID
                // where directed in the sample.
                Console.WriteLine("An error occurred while updating the
knowledge base.");
            }
            finally
            {
                Console.WriteLine("Press any key to continue.");
            }
        }

        static void Main(string[] args)
        {
            // Invoke the UpdateKB() method to update a knowledge base,
periodically
            // checking the status of the QnA Maker operation until the
            // knowledge base is updated.
            UpdateKB(kb, new_kb);
```

```
            // The console waits for a key to be pressed before closing.
            Console.ReadLine();
        }
    }
}
```

Understand what QnA Maker returns

A successful response is returned in JSON, as shown in the following example. Your results may differ slightly. If the final call returns a "Succeeded" state... your knowledge base was updated successfully. To troubleshoot, refer to the **Update Knowledgebase**[29] response codes of the QnA Maker API.

```
{
  "operationState": "NotStarted",
  "createdTimestamp": "2018-04-13T01:49:48Z",
  "lastActionTimestamp": "2018-04-13T01:49:48Z",
  "userId": "2280ef5917bb4ebfa1aae41fb1cebb4a",
  "operationId": "5156f64e-e31d-4638-ad7c-a2bdd7f41658"
}
...
{
  "operationState": "Succeeded",
  "createdTimestamp": "2018-04-13T01:49:48Z",
  "lastActionTimestamp": "2018-04-13T01:49:50Z",
  "resourceLocation": "/knowledgebases/140a46f3-b248-4f1b-9349-614bf-
d6e5563",
  "userId": "2280ef5917bb4ebfa1aae41fb1cebb4a",
  "operationId": "5156f64e-e31d-4638-ad7c-a2bdd7f41658"
}
Press any key to continue.
```

Once your knowledge base is updated, you can view it on your QnA Maker Portal, **My knowledge bases**[30] page. Notice that your knowledge base name has changed, for example QnA Maker FAQ (or the name of your pre-existing knowledge base) is now New KB Name.

Publish a knowledge base in C#

The following code publishes an existing knowledge base, using the Publish method.

1. Create a new C# project in your favorite IDE.
2. Add the code provided below.
3. Replace the `key` value with an access key valid for your subscription.
4. Run the program.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

29 https://westus.dev.cognitive.microsoft.com/docs/services/5a93fcf85b4ccd136866eb37/operations/5ac266295b4ccd1554da7600
30 https://www.qnamaker.ai/Home/MyServices

```
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

// NOTE: Install the Newtonsoft.Json NuGet package.
using Newtonsoft.Json;

namespace QnAMaker
{
    class Program
    {
        static string host = "https://westus.api.cognitive.microsoft.com";
        static string service = "/qnamaker/v4.0";
        static string method = "/knowledgebases/";

        // NOTE: Replace this with a valid subscription key.
        static string key = "ENTER KEY HERE";

        // NOTE: Replace this with a valid knowledge base ID.
        static string kb = "ENTER ID HERE";

        static string PrettyPrint(string s)
        {
            return JsonConvert.SerializeObject(JsonConvert.DeserializeOb-
ject(s), Formatting.Indented);
        }

        async static Task<string> Post(string uri)
        {
            using (var client = new HttpClient())
            using (var request = new HttpRequestMessage())
            {
                request.Method = HttpMethod.Post;
                request.RequestUri = new Uri(uri);
                request.Headers.Add("Ocp-Apim-Subscription-Key", key);

                var response = await client.SendAsync(request);
                if (response.IsSuccessStatusCode)
                {
                    return "{'result' : 'Success.'}";
                }
                else
                {
                    return await response.Content.ReadAsStringAsync();
                }
            }
        }

        async static void PublishKB()
```

```
        {
            var uri = host + service + method + kb;
            Console.WriteLine("Calling " + uri + ".");
            var response = await Post(uri);
            Console.WriteLine(PrettyPrint(response));
            Console.WriteLine("Press any key to continue.");
        }

        static void Main(string[] args)
        {
            PublishKB();
            Console.ReadLine();
        }
    }
}
```

The publish a knowledge base response

A successful response is returned in JSON, as shown in the following example:

```
{
  "result": "Success."
}
```

Working with the Azure IoT Hub

Azure IoT Hub overview

IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and secure communications between millions of IoT devices and a cloud-hosted solution backend. You can connect virtually any device to IoT Hub.

IoT Hub supports communications both from the device to the cloud and from the cloud to the device. IoT Hub supports multiple messaging patterns such as device-to-cloud telemetry, file upload from devices, and request-reply methods to control your devices from the cloud. IoT Hub monitoring helps you maintain the health of your solution by tracking events such as device creation, device failures, and device connections.

IoT Hub's capabilities help you build scalable, full-featured IoT solutions such as managing industrial equipment used in manufacturing, tracking valuable assets in healthcare, and monitoring office building usage.

Secure your communications

IoT Hub gives you a secure communication channel for your devices to send data.

- Per-device authentication enables each device to connect securely to IoT Hub and for each device to be managed securely.
- You have complete control over device access and can control connections at the per-device level.
- The IoT Hub Device Provisioning Service automatically provisions devices to the right IoT hub when the device first boots up.
- Multiple authentication types support a variety of device capabilities:
  - SAS token-based authentication to quickly get started with your IoT solution.
  - Individual X.509 certificate authentication for secure, standards-based authentication.
  - X.509 CA authentication for simple, standards-based enrollment.

Route device data

Built-in message routing functionality gives you flexibility to set up automatic rules-based message fan-out:

- Use message routing to control where your hub sends device telemetry.
- There is no additional cost to route messages to multiple endpoints.
- No-code routing rules take the place of custom message dispatcher code.

Integrate with other services

You can integrate IoT Hub with other Azure services to build complete, end-to-end solutions. For example, use:

- Azure Event Grid to enable your business to react quickly to critical events in a reliable, scalable, and secure manner.

- Azure Logic Apps to automate business processes.
- Azure Machine Learning to add machine learning and AI models to your solution.
- Azure Stream Analytics to run real-time analytic computations on the data streaming from your devices.

Configure and control your devices

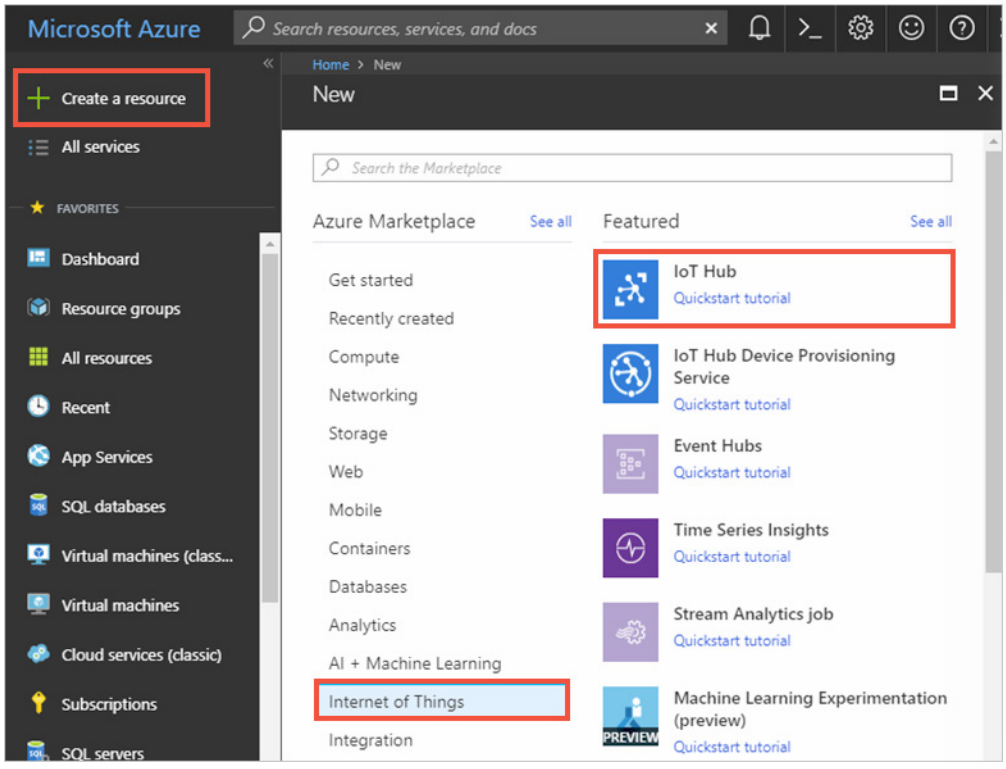
You can manage your devices connected to IoT Hub with an array of built-in functionality.

- Store, synchronize, and query device metadata and state information for all your devices.
- Set device state either per-device or based on common characteristics of devices.
- Automatically respond to a device-reported state change with message routing integration.

Create an IoT hub

The first step is to use the Azure portal to create an IoT hub in your subscription. The IoT hub enables you to ingest high volumes of telemetry into the cloud from many devices. The hub then enables one or more back-end services running in the cloud to read and process that telemetry.

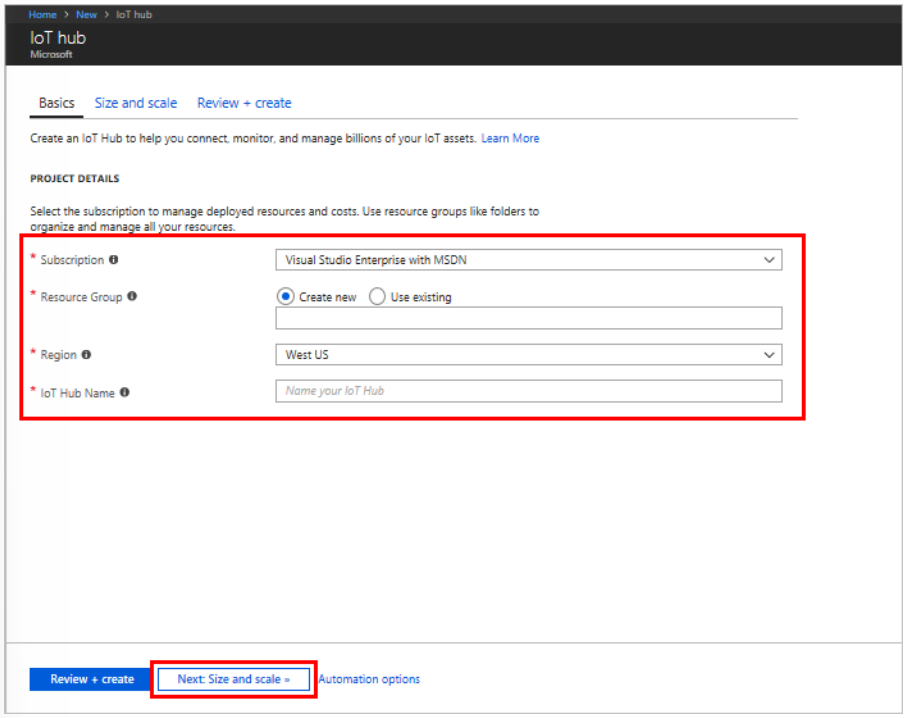
1. Sign in to the **Azure portal**[31].
2. Select **Create a resource** > **Internet of Things** > **IoT Hub**.



3.
4. In the IoT hub pane, enter the following information for your IoT hub:
   - **Subscription:** Choose the subscription that you want to use to create this IoT hub.

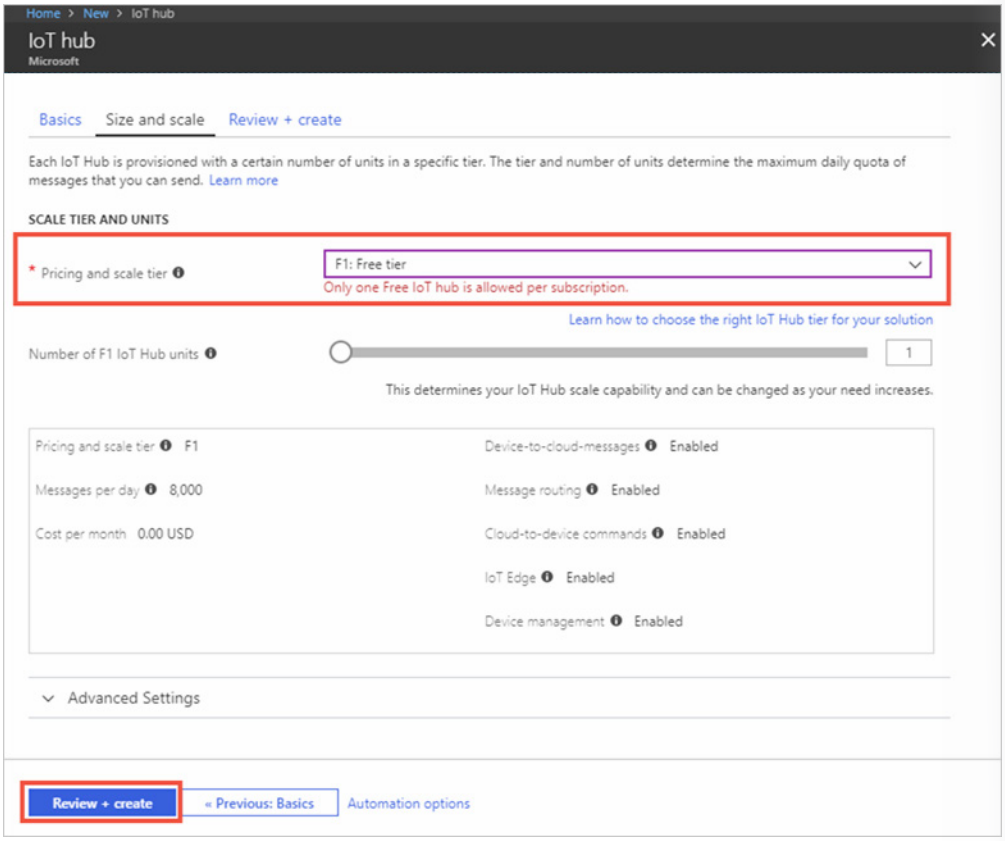
31 http://portal.azure.com/

- **Resource group:** Create a resource group to contain the IoT hub or use an existing one. By putting all related resources in a group together, such as **TestResources**, you can manage them all together. For example, deleting the resource group deletes all resources contained in that group.
- **Region:** Select the closest location to your devices.
- **Name:** Create a unique name for your IoT hub. If the name you enter is available, a green check mark appears.
- **Important:** The IoT hub will be publicly discoverable as a DNS endpoint, so make sure to avoid any sensitive information while naming it.



5.

6. Select **Next: Size and scale** to continue creating your IoT hub.
7. Choose your **Pricing and scale tier**. For this walkthrough, select the **F1 - Free tier** if it's still available on your subscription.



8.

9. Select **Review + create**.

10. Review your IoT hub information, then click **Create**. Your IoT hub might take a few minutes to create. You can monitor the progress in the Notifications pane.

Register a device

A device must be registered with your IoT hub before it can connect. In this walkthrough, you use the Azure CLI to register a simulated device.

1. Add the IoT Hub CLI extension and create the device identity. Replace `{YourIoTHubName}` with the name you chose for your IoT hub:

```
az extension add --name azure-cli-iot-ext
az iot hub device-identity create --hub-name {YourIoTHubName} --device-id
MyDotnetDevice
```

1. If you choose a different name for your device, update the device name in the sample applications before you run them.

2. Run the following command to get the *device connection string* for the device you just registered:

```
az iot hub device-identity show-connection-string --hub-name {YourIoTHub-
Name} --device-id MyDotnetDevice --output table
```

3. Make a note of the device connection string, which looks like `Hostname=...=`. You use this value later in the walkthrough.

4. You also need the *Event Hubs-compatible endpoint*, *Event Hubs-compatible path*, and *iothubowner primary key* from your IoT hub to enable the back-end application to connect to your IoT hub and retrieve the messages. The following commands retrieve these values for your IoT hub:

```
az iot hub show --query properties.eventHubEndpoints.events.endpoint --name
{YourIoTHubName}

az iot hub show --query properties.eventHubEndpoints.events.path --name
{YourIoTHubName}

az iot hub policy show --name iothubowner --query primaryKey --hub-name
{your IoT Hub name}
```

Review Questions

Module 5 Review Questions

Computer Vision API v2 overview

You are designing an application that will use facial recognition to identify repeat customers for a retail store. The solution will recommend products and services to the customers.

The application must be able to process a large number of images. You plan to implement the Computer Vision API v2.

What criteria must the images meet to use Computer Vision API v2?

Suggested Answer ↓

The Computer Vision API supports images of various file types up to 4MB in size. Images must be at least 50 x 50 pixels.

Analyze an image with C#

You plan to implement Computer Vision API v2.

What are the prerequisites for Computer Vision API v2?

Suggested Answer ↓

To use the Computer Vision API, you must procure a subscription key. You can use Visual Studio 2015 or 2017 to develop solutions that use the API. You must import the Microsoft.Azure.CognitiveServices.Vision.ComputerVision client library NuGet package into your Visual Studio solution.

QnA Maker

You are designing a solution that provides users answer to questions about common health issues. You plan to use QnA Maker.

What types of data sources can you use as input for QnA Maker? What type of developer experience is required to implement QnA Maker.

Suggested Answer ↓

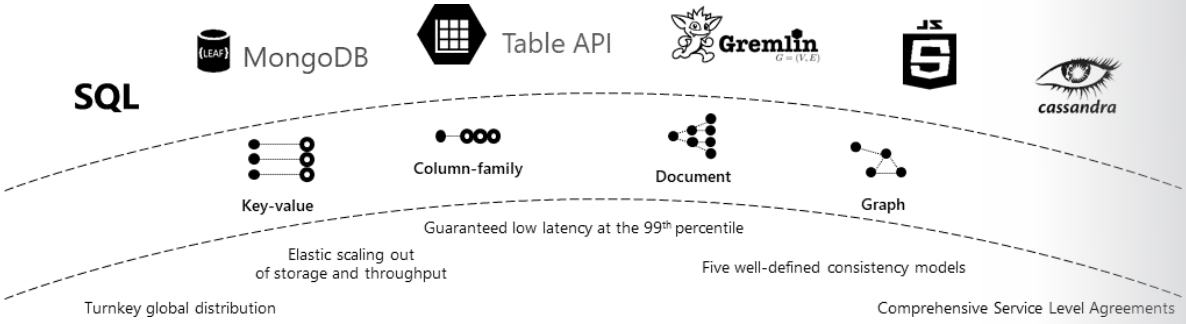
QnA Maker is a service that allows you to create a question and answer service. You can build out the service without any developer experience. You create a knowledgebase and publish it.

Module 6 Module Develop for Azure Storage

Develop Solutions that use Azure Cosmos DB Storage

Azure Cosmos DB

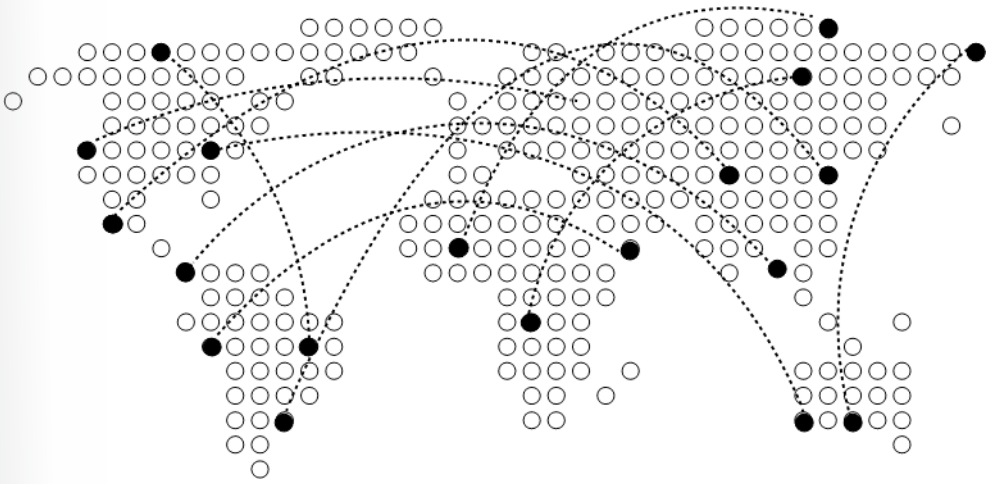
Microsoft Azure Cosmos DB is a database service native to Azure that focuses on providing a high-performance database regardless of your selected API or data model. Azure Cosmos DB offers multiple APIs and models that can be used interchangeably for various application scenarios.



Core Functionality

Global replication

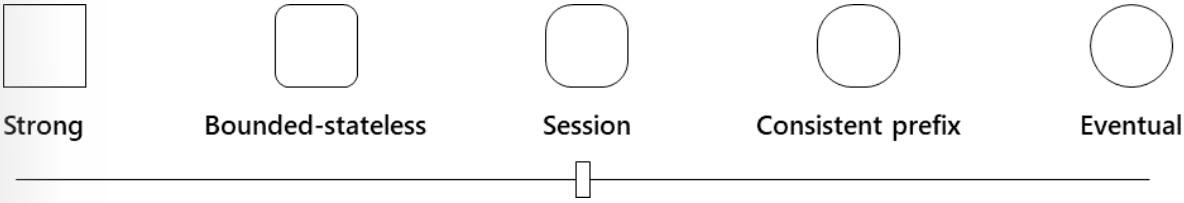
Azure Cosmos DB has a feature referred to as *turnkey global distribution* that automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure.



Consistency levels

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

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



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

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

Consistency Level	Description
Bounded Stateless	This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.
Session	This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.
Consistent Prefix	This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.
Eventual	This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary. This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize it's transactions.

Choose the Right Consistency Level for your Application

Distributed databases relying on replication for high availability, low latency or both, make the fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Most commercially available distributed databases ask developers to choose between the two extreme consistency models: strong consistency and eventual consistency. Azure Cosmos DB allows developers to choose among the five well-defined consistency models: strong, bounded staleness, session, consistent prefix, and eventual. Each of these consistency models is well-defined, intuitive and can be used for specific real-world scenarios. Each of the five consistency models provide availability and performance tradeoffs and are backed by comprehensive SLAs. The following simple considerations will help you make the right choice in many common scenarios.

SQL API and Table API

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

- For many real-world scenarios, session consistency is optimal and it's the recommended option.
- If your application requires strong consistency, it is recommended that you use bounded staleness consistency level.
- If you need stricter consistency guarantees than the ones provided by session consistency and single-digit-millisecond latency for writes, it is recommended that you use bounded staleness consistency level.

- If your application requires eventual consistency, it is recommended that you use consistent prefix consistency level.
- If you need less strict consistency guarantees than the ones provided by session consistency, it is recommended that you use consistent prefix consistency level.
- If you need the highest availability and lowest latency, then use eventual consistency level.

Consistency guarantees in practice

You may get stronger consistency guarantees in practice. Consistency guarantees for a read operation correspond to the freshness and ordering of the database state that you request. Read-consistency is tied to the ordering and propagation of the write/update operations.

- When the consistency level is set to **bounded staleness**, Cosmos DB guarantees that the clients always read the value of a previous write, with a lag bounded by the staleness window.
- When the consistency level is set to **strong**, the staleness window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the write operation.
- For the remaining three consistency levels, the staleness window is largely dependent on your workload. For example, if there are no write operations on the database, a read operation with **eventual**, **session**, or **consistent** prefix consistency levels is likely to yield the same results as a read operation with strong consistency level.

If your Cosmos DB account is configured with a consistency level other than the strong consistency, you can find out the probability that your clients may get strong and consistent reads for your workloads by looking at the Probabilistic Bounded Staleness (PBS) metric. This metric is exposed in the Azure portal.

Probabilistic bounded staleness shows how eventual is your eventual consistency. This metric provides an insight into how often you can get a stronger consistency than the consistency level that you have currently configured on your Cosmos DB account. In other words, you can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

Consistency Levels and Azure Cosmos DB APIs

Five consistency models offered by Azure Cosmos DB are natively supported by the Azure Cosmos DB SQL API. When you use Azure Cosmos DB, the SQL API is the default.

Azure Cosmos DB also provides native support for wire protocol-compatible APIs for popular databases. Databases include MongoDB, Apache Cassandra, Gremlin, and Azure Table storage. These databases don't offer precisely defined consistency models or SLA-backed guarantees for consistency levels. They typically provide only a subset of the five consistency models offered by Azure Cosmos DB. For the SQL API, Gremlin API, and Table API, the default consistency level configured on the Azure Cosmos DB account is used.

The following sections show the mapping between the data consistency requested by an OSS client driver for Apache Cassandra 4.x and MongoDB 3.4. This document also shows the corresponding Azure Cosmos DB consistency levels for Apache Cassandra and MongoDB.

Mapping between Apache Cassandra and Azure Cosmos DB consistency levels

This table shows the "read consistency" mapping between the Apache Cassandra 4.x client and the default consistency level in Azure Cosmos DB. The table shows multi-region and single-region deployments.

Apache Cassandra 4.x	Azure Cosmos DB (multi-region)	Azure Cosmos DB (single region)
ONE, TWO, THREE	Consistent prefix	Consistent prefix
LOCAL_ONE	Consistent prefix	Consistent prefix
QUORUM, ALL, SERIAL	Bounded staleness is the default. Strong is in private preview.	Strong
LOCAL_QUORUM	Bounded staleness	Strong
LOCAL_SERIAL	Bounded staleness	Strong

Mapping between MongoDB 3.4 and Azure Cosmos DB consistency levels

The following table shows the "read concerns" mapping between MongoDB 3.4 and the default consistency level in Azure Cosmos DB. The table shows multi-region and single-region deployments.

MongoDB 3.4	Azure Cosmos DB (multi-region)	Azure Cosmos DB (single region)
Linearizable	Strong	Strong
Majority	Bounded staleness	Strong
Local	Consistent prefix	Consistent prefix

Azure Cosmos DB Supported APIs

Today, Azure Cosmos DB can be accessed by using five different APIs. The underlying data structure in Azure Cosmos DB is a data model based on atom record sequences that enabled Azure Cosmos DB to support multiple data models. Because of the flexible nature of atom record sequences, Azure Cosmos DB will be able to support many more models and APIs over time.

MongoDB API

The MongoDB API in Azure Cosmos DB acts as a massively scalable MongoDB service powered by the Azure Cosmos DB platform. It is compatible with existing MongoDB libraries, drivers, tools, and applications.

Table API

The Table API in Azure Cosmos DB is a key-value database service built to provide premium capabilities (for example, automatic indexing, guaranteed low latency, and global distribution) to existing Azure Table storage applications without making any app changes.

Gremlin API

The Gremlin API in Azure Cosmos DB is a fully managed, horizontally scalable graph database service that makes it easy to build and run applications that work with highly connected datasets supporting Open Graph APIs (based on the Apache TinkerPop specification, Apache Gremlin).

Apache Cassandra API

The Cassandra API in Azure Cosmos DB is a globally distributed Apache Cassandra service powered by the Azure Cosmos DB platform. Compatible with existing Apache Cassandra libraries, drivers, tools, and applications.

SQL API

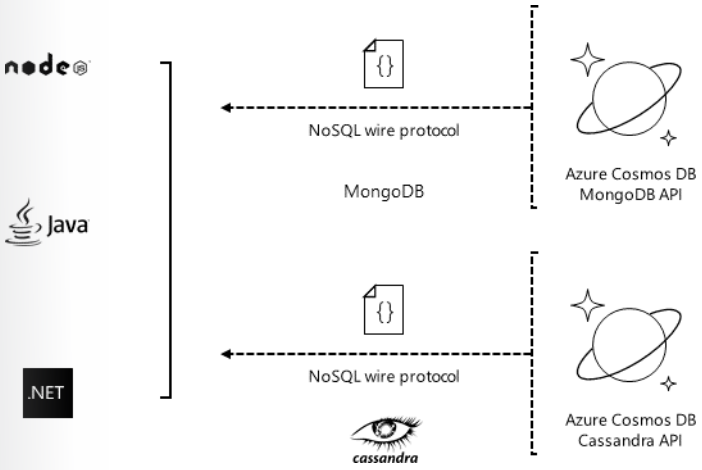
The SQL API in Azure Cosmos DB is a JavaScript and JavaScript Object Notation (JSON) native API based on the Azure Cosmos DB database engine. The SQL API also provides query capabilities rooted in the familiar SQL query language. Using SQL, you can query for documents based on their identifiers or make deeper queries based on properties of the document, complex objects, or even the existence of specific properties. The SQL API supports the execution of JavaScript logic within the database in the form of stored procedures, triggers, and user-defined functions.

Migrating from NoSQL

Many NoSQL database engines are simple to get started with, but they provide problems as you scale, including:

- Tedious set-up and maintenance requirements for a multiple-server database cluster
- Expensive and complex high-availability solutions
- Challenges in achieving end-to-end security, including encryption at rest and in flight
- Required resource overprovisioning and unpredictable costs to achieve scale

Azure Cosmos DB has a MongoDB API and a Cassandra API to provide a NoSQL service offering for two of the most popular NoSQL database platforms. Both APIs are protocol compatible with the Cassandra API supporting CQLv4 and the MongoDB API supporting MongoDB v5. Many applications can be "lifted and shifted" to Azure Cosmos DB without the need to rewrite code.



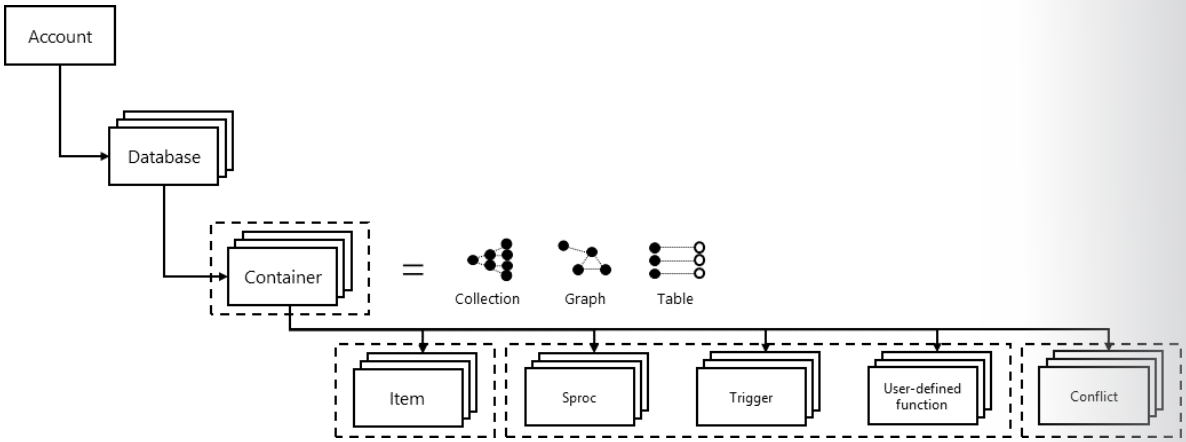
To achieve a successful migration, it is important to keep a few tips in mind:

- Instead of writing custom code, you should use native tools, such as the Cassandra shell, mongodump, and mongoexport.
- Azure Cosmos DB containers should be allocated prior to the migration with the appropriate throughput levels set. Many of the tools will create containers for you with default settings that are not ideal.

- Prior to migrating, you should increase the container's throughput to at least 1,000 Request Units (RUs) per second so that the import tools are not throttled. The throughput can be reverted back to the typical values after the import is complete.

Resource Hierarchy

The JSON documents stored in the Azure Cosmos DB SQL API are managed through a well-defined hierarchy of database resources. The Azure Cosmos DB hierarchical resource model consists of sets of resources under a database account, each addressable via a logical and stable URI. A set of resources is referred to as a feed.



Resource	Description
Account	A database account is associated with a set of databases and a fixed amount of large object (blob) storage for attachments. You can create one or more database accounts by using your Azure subscription. For more information, visit the pricing page.
Database	A database is a logical container of document storage partitioned across collections. It is also a users container.
Collection (container)	A collection is a container of JSON documents and the associated JavaScript application logic. Collections can span one or more partitions or servers and can scale to handle practically unlimited volumes of storage or throughput.
Document (item)	User-defined (arbitrary) JSON content. By default, no schema needs to be defined nor do secondary indexes need to be provided for all the documents added to a collection.
Stored procedure (sproc)	Application logic written in JavaScript that is registered with a collection and executed within the database engine as a transaction.
Trigger	Application logic written in JavaScript executed before or after either an insert, replace, or delete operation.

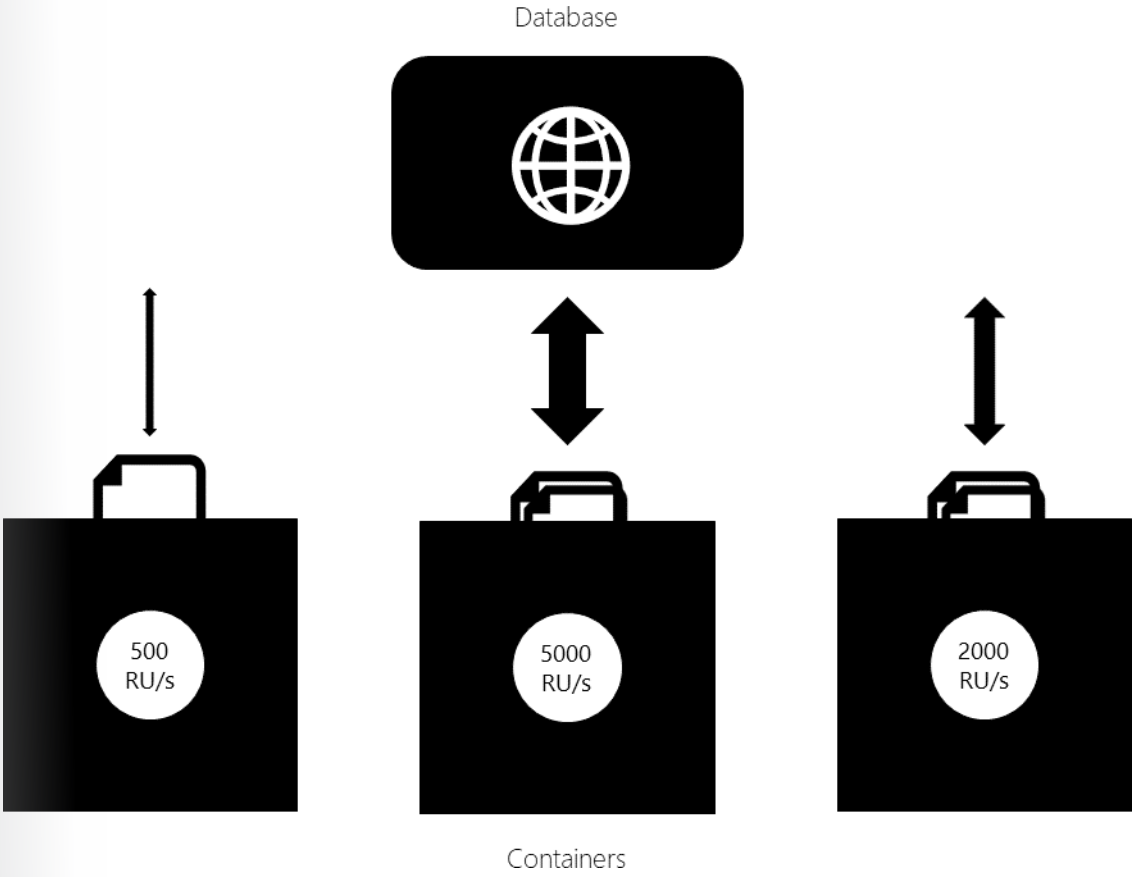
Resource	Description
User-defined function	Application logic written in JavaScript. User-defined functions enable you to model a custom query operator and thereby extend the core SQL API query language.

Collections

In the Azure Cosmos DB SQL API, databases are essentially containers for collections. Collections are where you place individual documents. A collection is intrinsically elastic—it automatically grows and shrinks as you add or remove documents.

Each collection is assigned a throughput value, and that value dictates the maximum throughput for that collection and its corresponding documents. Alternatively, you can assign the throughput at the database level and share the throughput values among the collections in the database. If you have a set of documents that needs throughput beyond the limits of an individual collection, you can distribute the documents among multiple collections. Each collection has its own distinct throughput level.

If a particular collection is seeing spikes in throughput, you can manage its throughput level in isolation by increasing or decreasing the value. This change to the throughput level of a particular collection will not cause side effects for the other collections. This allows you to adjust to meet the performance needs of any workload in isolation.



You can also scale workloads across collections, if you have a workload that needs to be partitioned, you can scale that workload by distributing its associated documents across multiple collections. The SQL API

for Azure Cosmos DB includes a client-side partition resolver that allows you to manage transactions and point them in code to the correct partition based on a partition key field.

Collection types

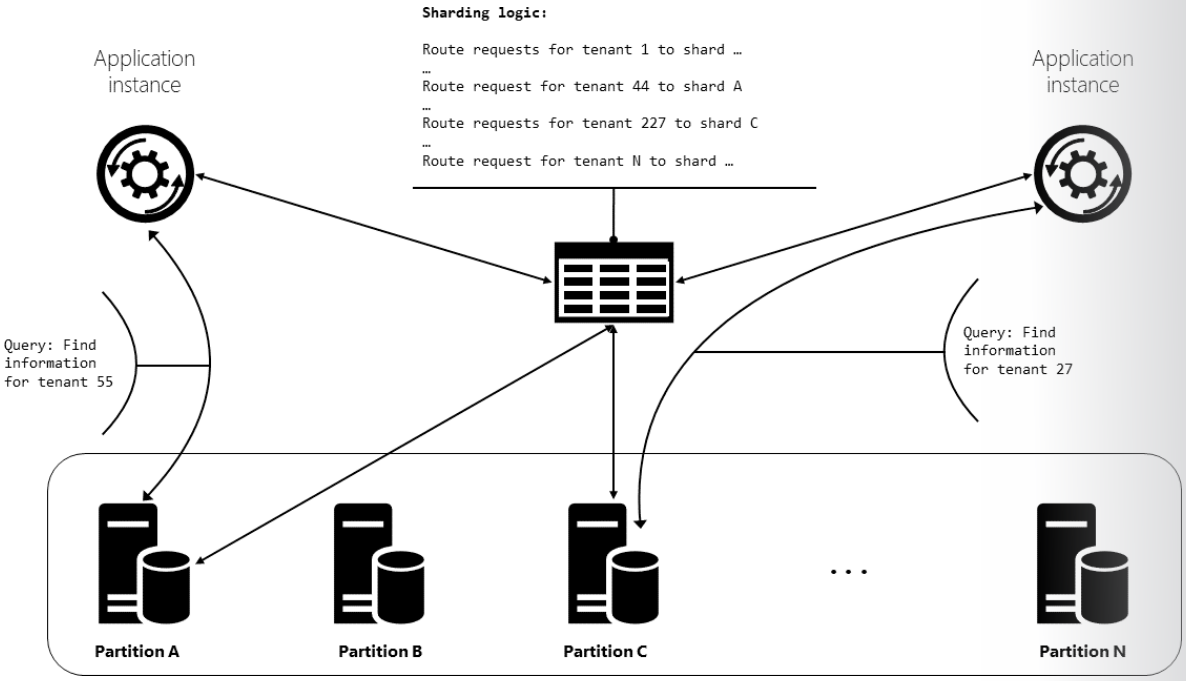
Azure Cosmos DB containers can be created as fixed or unlimited in the Azure portal. Fixed-size containers have a maximum limit of 10 GB and a 10,000 RU/s throughput. To create a container as unlimited, you must specify a partition key and a minimum throughput of 1,000 RU/s. Azure Cosmos DB containers can also be configured to share throughput among the containers in a database.

If you created a fixed container with no partition key or a throughput less than 1,000 RU/s, the container will not automatically scale. To migrate the data from a fixed container to an unlimited container, you need to use the data migration tool or the Change Feed library.

Partitioning

Azure Cosmos DB provides containers for storing data called collections (for documents), graphs, or tables. *Containers* are logical resources and can span one or more physical partitions or servers. The number of partitions is determined by Azure Cosmos DB based on the storage size and throughput provisioned for a container or set of containers.

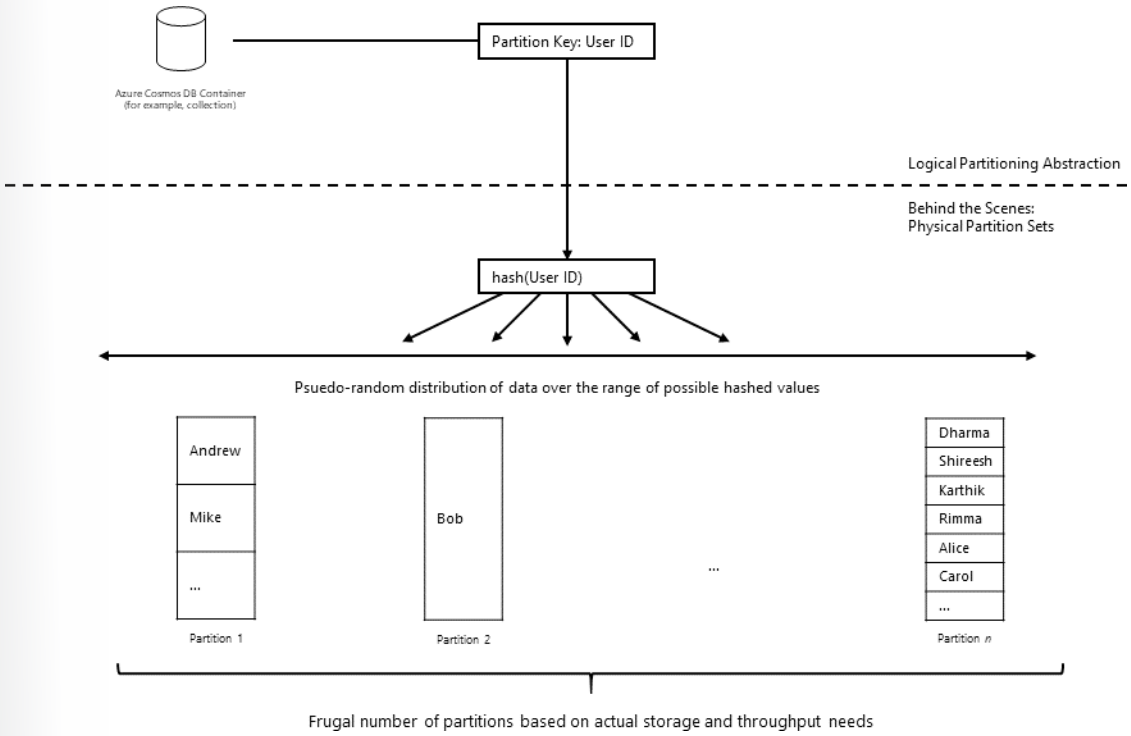
If you are already familiar with the sharding pattern, the idea of dynamic partitioning is not very different.



A *physical partition* is a fixed amount of reserved solid-state drive (SSD) backend storage combined with a variable amount of compute resources (CPU and memory). Each physical partition is replicated for high availability. A physical partition is an internal concept of Azure Cosmos DB, and physical partitions are transient. Azure Cosmos DB will automatically scale the number of physical partitions based on your workload.

A *logical partition* is a partition within a physical partition that stores all the data associated with a single partition key value. Partition ranges can be dynamically subdivided to seamlessly grow the database as the application grows while simultaneously maintaining high availability. When a container meets the

partitioning prerequisites, partitioning is completely transparent to your application. Azure Cosmos DB handles distributing data across physical and logical partitions and routing query requests to the right partition.



Manage Collections and Documents by using the Microsoft .NET SDK

To get started with the Azure Cosmos DB SQL API, you will need the **Microsoft.Azure.DocumentDB.Core**[1] package from NuGet.

First, you will need to add the following `using` directives to the top of your class file:

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

Then, you can create a `DocumentClient` instance by using the endpoint from your Azure Cosmos DB account and one of your keys:

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

To reference any resource in the software development kit (SDK), you will need a URI. The `UriFactory` class contains a series of static helper methods that can create URIs for common Azure Cosmos DB resources. In this example, we will create a URI for a collection:

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

1 https://www.nuget.org/packages/Microsoft.Azure.DocumentDB.Core

Now, you can use the `CreateDocumentAsync` method of the `DocumentClient` class to insert a C# object into the collection. You can use any C# type you want for your documents, because the SDK doesn't require a specific base type:

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

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

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

```
var query = client.CreateDocumentQuery<Family>(
    collectionUri,
    new SqlQuerySpec()
    {
          QueryText = "SELECT * FROM f WHERE (f.surname = @lastName)",
          Parameters = new SqlParameterCollection()
          {
                new SqlParameter("@lastName", "Andt")
          }
    },
    DefaultOptions
);

var families = query.ToList();
```

Alternatively, you can use the language-integrated query (LINQ) feature of C# with the SDK. The LINQ expressions will be automatically translated into the appropriate SQL query:

```
var query = client.CreateDocumentQuery<Family>(collectionUri)
    .Where(d => d.Surname = "Andt")
    .Select(d => new { Name = d.Id, City = d.Address?.City)
    .AsDocumentQuery();

var families = query.ToList();
```

Develop Solutions that use a Relational Database

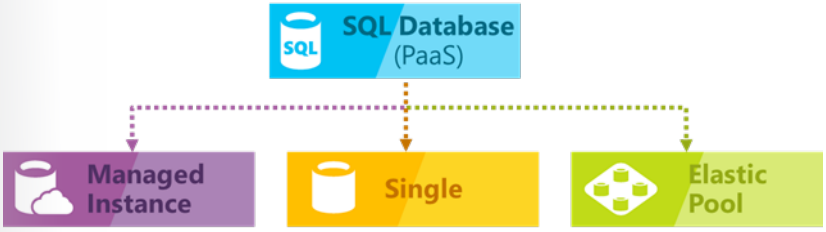
The Azure SQL Database Service

SQL Database is a general-purpose relational database managed service in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML. SQL Database delivers dynamically scalable performance within two different purchasing models: a vCore-based purchasing model and a DTU-based purchasing model. SQL Database also provides options such as columnstore indexes for extreme analytic analysis and reporting, and in-memory OLTP for extreme transactional processing. Microsoft handles all patching and updating of the SQL code base seamlessly and abstracts away all management of the underlying infrastructure.

Azure SQL Database provides the following deployment options for an Azure SQL database:

- As a single database with its own set of resources managed via a logical server
- As a pooled database in an elastic pool with a shared set of resources managed via a logical server
- As a part of a collection of databases known as a managed instance that contains system and user databases and sharing a set of resources

The following illustration shows these deployment options:



SQL Database shares its code base with the Microsoft SQL Server database engine. With Microsoft's cloud-first strategy, the newest capabilities of SQL Server are released first to SQL Database, and then to SQL Server itself. This approach provides you with the newest SQL Server capabilities with no overhead for patching or upgrading - and with these new features tested across millions of databases.

Choosing the Right SQL Server Option in Azure

In Azure, you can have your SQL Server workloads running in a hosted infrastructure (IaaS) or running as a hosted service (PaaS). The key question that you need to ask when deciding between PaaS or IaaS is do you want to manage your database, apply patches, take backups, or you want to delegate these operations to Azure? Depending on the answer, you have the following options:

- **Azure SQL Database:** A fully-managed SQL database engine, based on the latest stable Enterprise Edition of SQL Server. This is a relational database-as-a-service (DBaaS) hosted in the Azure cloud that falls into the industry category of Platform-as-a-Service (PaaS). SQL database is built on standardized hardware and software that is owned, hosted, and maintained by Microsoft. With SQL Database, you can use built-in features and functionality that require extensive configuration in SQL Server. When using SQL Database, you pay-as-you-go with options to scale up or out for greater power with no

interruption. SQL Database has additional features that are not available in SQL Server, such as built-in intelligence and management. Azure SQL Database offers several deployment options:

- You can deploy a single database to a logical server. A logical server containing single and pooled databases offers most of database-scoped features of SQL Server. This option is optimized for modern application development of new cloud-born applications.
- You can deploy to a Azure SQL Database Managed Instances. With Azure SQL Database Managed Instance, Azure SQL Database offers shared resources for databases and additional instance-scoped features. Azure SQL Database Managed Instance supports database migration from on-premises with minimal to no database change. This option provides all of the PaaS benefits of Azure SQL Database but adds capabilities that were previously only available in SQL VMs. This includes a native virtual network (VNet) and near 100% compatibility with on-premises SQL Server.

- **SQL Server on Azure Virtual Machines** falls into the industry category Infrastructure-as-a-Service (IaaS) and allows you to run SQL Server inside a fully-managed virtual machine in the Azure cloud. SQL Server virtual machines also run on standardized hardware that is owned, hosted, and maintained by Microsoft. When using SQL Server on a VM, you can either pay-as you-go for a SQL Server license already included in a SQL Server image or easily use an existing license. You can also stop or resume the VM as needed.SQL Server installed and hosted in the cloud on Windows Server or Linux virtual machines (VMs) running on Azure, also known as an infrastructure as a service (IaaS). SQL Server on Azure virtual machines is a good option for migrating on-premises SQL Server databases and applications without any database change. All recent versions and editions of SQL Server are available for installation in an IaaS virtual machine. The most significant difference from SQL Database is that SQL Server VMs allow full control over the database engine. You can choose when maintenance/patching will start, to change the recovery model to simple or bulk logged to enable faster load less log, to pause or start engine when needed, and you can fully customize the SQL Server database engine. With this additional control comes with added responsibility to manage the virtual machines.

The main differences between these options are listed in the following table:

SQL Server on VM	Azure SQL Database (Managed Instance)	Azure SQL Database (Logical server)
You have full control over the SQL Server engine. Up to 99.95% availability. Full parity with the matching version of on-premises SQL Server. Fixed, well-known database engine version. Easy migration from SQL Server on-premises. Private IP address within Azure VNet. You have ability to deploy application or services on the host where SQL Server is placed.	High compatibility with SQL Server on-premises. 99.99% availability guaranteed. Built-in backups, patching, recovery. Latest stable Database Engine version. Easy migration from SQL Server. Private IP address within Azure VNet. Built-in advanced intelligence and security. Online change of resources (CPU/ storage).	The most commonly used SQL Server features are available. 99.99% availability guaranteed. Built-in backups, patching, recovery. Latest stable Database Engine version. Ability to assign necessary resources (CPU/ storage) to individual databases. Built-in advanced intelligence and security. Online change of resources (CPU/storage).
You need to manage your backups and patches. You need to implement your own High-Availability solution. There is a downtime while changing the resources(CPU/ storage)	There is still some minimal number of SQL Server features that are not available. No guaranteed exact maintenance time (but nearly transparent). Compatibility with the SQL Server version can be achieved only using database compatibility levels.	Migration from SQL Server might be hard. Some SQL Server features are not available. No guaranteed exact maintenance time (but nearly transparent). Compatibility with the SQL Server version can be achieved only using database compatibility levels. Private IP address cannot be assigned (you can limit the access using firewall rules).

Copy a transactionally Consistent Copy of an Azure SQL Database

Azure SQL Database provides several methods for creating a transactionally consistent copy of an existing Azure SQL database on either the same server or a different server. You can copy a SQL database by using the Azure portal, PowerShell, or T-SQL.

Overview

A database copy is a snapshot of the source database as of the time of the copy request. You can select the same server or a different server, its service tier and compute size, or a different compute size within the same service tier (edition). After the copy is complete, it becomes a fully functional, independent database. At this point, you can upgrade or downgrade it to any edition. The logins, users, and permissions can be managed independently.

Note: Automated database backups are used when you create a database copy.

Logins in the database copy

When you copy a database to the same logical server, the same logins can be used on both databases. The security principal you use to copy the database becomes the database owner on the new database. All database users, their permissions, and their security identifiers (SIDs) are copied to the database copy.

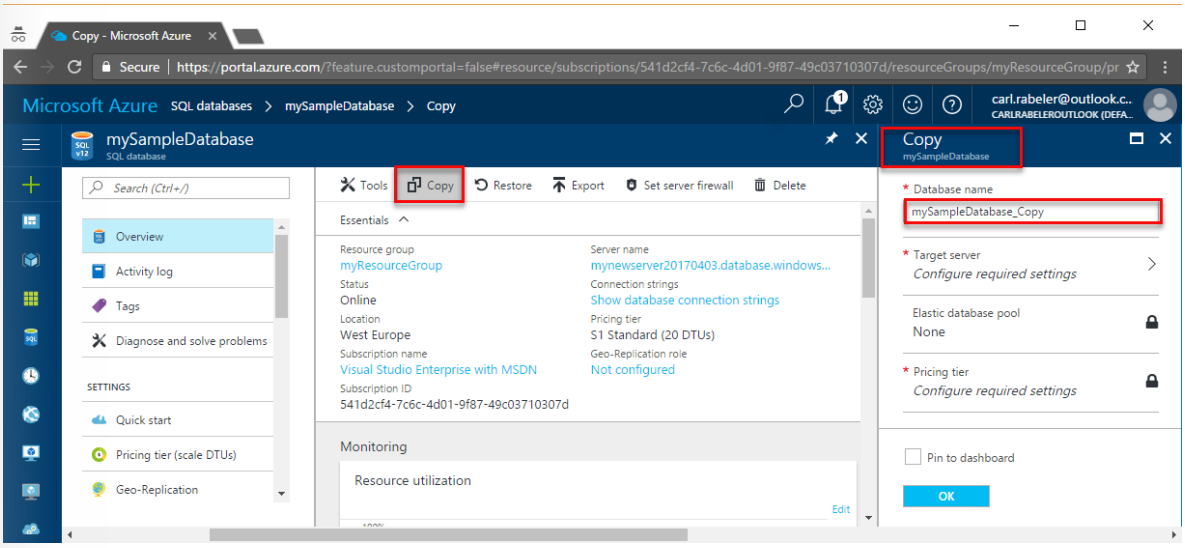
When you copy a database to a different logical server, the security principal on the new server becomes the database owner on the new database. If you use contained database users for data access, ensure that both the primary and secondary databases always have the same user credentials, so that after the copy is complete you can immediately access it with the same credentials.

If you use Azure Active Directory, you can completely eliminate the need for managing credentials in the copy. However, when you copy the database to a new server, the login-based access might not work, because the logins do not exist on the new server. To learn about managing logins when you copy a database to a different logical server, see How to manage Azure SQL database security after disaster recovery.

After the copying succeeds and before other users are remapped, only the login that initiated the copying, the database owner, can log in to the new database.

Copy a database by using the Azure portal

To copy a database by using the Azure portal, open the page for your database, and then click **Copy**.



Copy a database by using PowerShell

To copy a database by using PowerShell, use the `New-AzureRmSqlDatabaseCopy` cmdlet.

```
New-AzureRmSqlDatabaseCopy -ResourceGroupName "myResourceGroup" `
    -ServerName $sourceserver `
    -DatabaseName "MySampleDatabase" `
    -CopyResourceGroupName "myResourceGroup" `
    -CopyServerName $targetserver `
    -CopyDatabaseName "CopyOfMySampleDatabase"
```

Resolve logins

After the new database is online on the destination server, use the `ALTER USER` statement to remap the users from the new database to logins on the destination server. To resolve orphaned users, see Troubleshoot Orphaned Users.

All users in the new database retain the permissions that they had in the source database. The user who initiated the database copy becomes the database owner of the new database and is assigned a new security identifier (SID). After the copying succeeds and before other users are remapped, only the login that initiated the copying, the database owner, can log in to the new database.

Entity Framework

Entity Framework is an object-relational mapper library for Microsoft .NET that is designed to reduce the impedance mismatch between the relational and object-oriented worlds. The goal of the library is to enable developers to interact with data stored in relational databases by using strongly-typed .NET objects that represent the application's domain and to eliminate the need for a large portion of the data access “plumbing” code that they usually need to write to access data in a database.

Entity Framework Core and Entity Framework

Entity Framework Core (EF Core) is a recent rewrite of the entire Entity Framework library to target .NET Standard. Entity Framework Core can be used with .NET Framework applications and .NET Core applications. Entity Framework Core was built to be more lightweight and agile than the full Entity Framework by dropping many of the earlier features from Entity Framework and implementing new, modern, and extensible features at an agile pace. For new applications, we recommend considering using Entity Framework Core over Entity Framework.

Note: The examples in this section will assume that you are using Entity Framework Core.

Entity Framework Providers

The Entity Framework provider model allows Entity Framework to be used with different types of database servers. For example, one provider can be plugged in to allow Entity Framework to be used against Microsoft SQL Server, whereas another provider can be plugged in to allow Entity Framework to be used against Oracle Database. There are many current providers in the market for databases, including:

- SQL Server
- SQLite
- PostgreSQL
- MySQL
- MariaDB
- MyCAT Server
- SQL Server Compact
- Firebird
- DB2
- Informix
- Oracle
- Microsoft Access

Note: If a provider you need is not available, you can certainly write a provider yourself, although it should not be considered a trivial undertaking.

Entity Framework Core also ships with an InMemory provider. This database provider allows Entity Framework Core to be used with an in-memory database. The InMemory provider is useful when you want to test components by using something that approximates connecting to the real database without the overhead of actual database operations.

Note: EF Core database providers do not have to be relational databases. InMemory is designed to be a general-purpose database for testing and is not designed to mimic a relational database.

SQL Server provider

This database provider allows Entity Framework Core to be used with Microsoft SQL Server (including Microsoft Azure SQL Database). The provider is maintained as an open-source project as part of the Entity Framework Core repository on GitHub (**https://github.com/aspnet/EntityFrameworkCore**).

Many of the examples you will find online assume that you are using the SQL Server provider with Entity Framework Core. It is important to remember that Entity Framework Core has a provider model that

abstracts the underlying database away from the actual database access logic. The code samples you see can be used with many of the database providers.

MySQL and PostgreSQL providers

The MySQL team maintains a database provider for both Entity Framework and Entity Framework Core as part of the MySQL Connector for .NET library. Along with the MySQL team, other third-party groups have written providers for MySQL. Two of the MySQL providers are:

- MySql.Data.EntityFrameworkCore (**https://www.nuget.org/packages/MySql.Data.EntityFrameworkCore**).
- Pomelo.EntityFrameworkCore.MySql (**https://www.nuget.org/packages/Pomelo.EntityFrameworkCore.MySql/**).

There are multiple third-party organizations that have written .NET libraries to access PostgreSQL. Many of them have rewritten their Entity Framework providers to support Entity Framework Core. These libraries include:

- Npgsql.EntityFrameworkCore.PostgreSQL (**https://www.nuget.org/packages/Npgsql.EntityFrameworkCore.PostgreSQL/**).
- Devart.Data.PostgreSql.EFCore (**https://www.nuget.org/packages/Devart.Data.PostgreSql.EFCore/**).

These providers allow you to use Entity Framework Core with a MySQL or PostgreSQL database in the same manner as you would use the library with a SQL database.

Modeling a Database by using Entity Framework Core

To use Entity Framework to query, insert, update, and delete data using .NET objects, you first need to create a model that maps the entities and relationships defined in your model to tables in a database.

Entity Framework uses a set of conventions to build a model based on the shapes of your entity classes. You can specify additional configuration to supplement and override what was discovered by convention. The conventions can be applied to a model targeting any data store and when targeting any relational database. Providers might also enable a configuration that is specific to a particular data store.

First, let's look at how we can model a database with a single table named **Blogs**.

BlogId	Url	Description
1	/first-post	This is my first post on this platform
2	/follow-up-post	NULL

If we want to use plain-old CLR objects (POCOs), such as existing domain objects, to model this table, we would have a class that looks like this:

```
public class Blog
{
    public int BlogId { get; set; }

    public string Url { get; set; }

    public string Description { get; set; }
```

```
}
```

Logically, our database has a table that is a collection of these blog instances. Without knowing anything about Entity Framework, we would probably create a class that looks like this:

```
public class BlogDatabase
{
    public IEnumerable<Blog> Blogs { get; set; }
}
```

Now, we need to find a way to mark these classes in C# as models of our database. Including a type in the model means that Entity Framework has metadata about that type and will attempt to read and write instances from and to the database. There are two methods for modeling a database: the fluent API or data annotations.

Fluent API

You can override the `OnModelCreating` method in your derived context class and use the ModelBuilder API to configure your model. This is the most powerful method of configuration and allows the configuration to be specified without modifying your entity classes. The fluent API configuration has the highest precedence and will override conventions and data annotations:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
           .HasKey(c => c.BlogId)
           .Property(b => b.Url)
           .IsRequired()
           .Property(b => b.Description);
}
```

Data annotations

You can apply attributes (known as data annotations) to your classes and properties. data annotations will override conventions but will be overwritten by a fluent API configuration (if it exists):

```
public class Blog
{
    [Key]
    public int BlogId { get; set; }

    [Required]
    public string Url { get; set; }

    public string Description { get; set; }
}
```

DbContext implementation

After you have a model, the primary class your application interacts with is `System.Data.Entity.DbContext` (often referred to as the *context class*). You can use a `DbContext` class associated to a model to:

- Write and execute queries.
- Materialize query results as entity objects.
- Track changes that are made to those objects.
- Persist object changes back on the database.
- Bind objects in memory to UI controls.

The recommended way to work with the context is to define a class that derives from `DbContext` and exposes `DbSet` properties that represent collections of the specified entities in the context:

```
public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
}
```

By convention, types that are exposed in `DbSet` properties on your context are included in your model. In addition, types that are mentioned in the `OnModelCreating` method are also included. Finally, any types that are found by recursively exploring the navigation properties of discovered types are also included in the model.

Querying Databases by using Entity Framework Core

The `DbSet<>` generic class includes methods that will allow you to query your database by using language-integrated query (LINQ).

If you are already familiar with the LINQ syntax, you can perform many queries in Entity Framework Core without the need to learn too much. This is because the `DbSet<>` generic class implements the `IEnumerable<>` interface, giving you access to many of the existing LINQ queries.

For example, you can load all the data from a table by enumerating the collection with a call to the `ToList` method:

```
List<Blog> allblogs = context.Blogs.ToList();
```

You can use LINQ methods such as the `Where` method to filter your resulting list:

```
IEnumerable<Blog> someblogs = context.Blogs
    .Where(b => b.Url.Contains("dotnet"))
```

You can also use the `Single` method to get a single instance that matches a specific filter:

```
Blog specificblog = context.Blogs
    .Single(b => b.BlogId == 1);
```

When you call LINQ operators, you are simply building up an in-memory representation of the query. The query is sent to the database only when the results are consumed (enumerated). The most common operations that result in the query being sent to the database are:

- Iterating the results in a **for** loop.
- Using an operator such as *ToList*, *ToArray*, *Single*, or *Count*.
- Data binding the results of a query to a UI.

Although Entity Framework does help protect against SQL injection attacks, it does not do any general validation of input. Therefore, if values being passed to APIs, used in LINQ queries, assigned to entity properties, and so on come from an untrusted source, the appropriate validation per your application requirements should be performed. This includes any user input used to dynamically construct queries. Even when using LINQ, if you are accepting user input to build expressions, you need to make sure that only intended expressions can be constructed.

Develop Solutions that use Microsoft Azure Blob Storage

Introduction to Azure Blob Storage

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that does not adhere to a particular data model or definition, such as text or binary data.

About Blob storage

Blob storage is designed for:

- Serving images or documents directly to a browser.
- Storing files for distributed access.
- Streaming video and audio.
- Writing to log files.
- Storing data for backup and restore, disaster recovery, and archiving.
- Storing data for analysis by an on-premises or Azure-hosted service.

Users or client applications can access objects in Blob storage via HTTP/HTTPS, from anywhere in the world. Objects in Blob storage are accessible via the Azure Storage REST API, Azure PowerShell, Azure CLI, or an Azure Storage client library. Client libraries are available for a variety of languages, including .NET, Java, Node.js, Python, Go, PHP, and Ruby.

About Azure Data Lake Storage Gen2

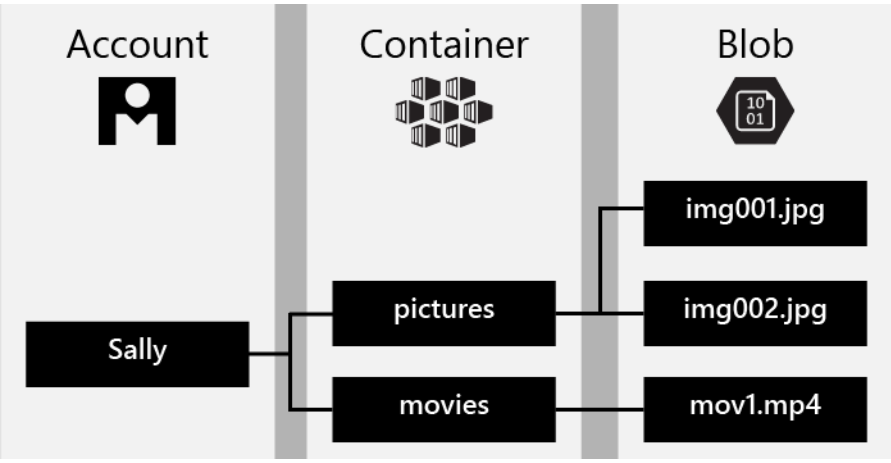
Blob storage supports Azure Data Lake Storage Gen2, Microsoft's enterprise big data analytics solution for the cloud. Azure Data Lake Storage Gen2 offers a hierarchical file system as well as the advantages of Blob storage, including low-cost, tiered storage; high availability; strong consistency; and disaster recovery capabilities.

Blob storage resources

Blob storage offers three types of resources:

- The **storage account**.
- A **container** in the storage account
- A **blob** in a container

The following diagram shows the relationship between these resources.



Storage accounts

A storage account provides a unique namespace in Azure for your data. Every object that you store in Azure Storage has an address that includes your unique account name. The combination of the account name and the Azure Storage service endpoint forms the endpoints for your storage account.

For example, if your storage account is named *mystorageaccount*, then the default endpoint for Blob storage is:

```
http://mystorageaccount.blob.core.windows.net
```

To learn more about storage accounts, see **Azure storage account overview**[2].

Containers

A container organizes a set of blobs, similar to a directory in a file system. A storage account can include an unlimited number of containers, and a container can store an unlimited number of blobs.

Note: The container name must be lowercase.

Blobs

Azure Storage supports three types of blobs:

- **Block blobs** store text and binary data, up to about 4.7 TB. Block blobs are made up of blocks of data that can be managed individually.
- **Append blobs** are made up of blocks like block blobs, but are optimized for append operations. Append blobs are ideal for scenarios such as logging data from virtual machines.
- **Page blobs** store random access files up to 8 TB in size. Page blobs store the virtual hard drive (VHD) files serve as disks for Azure virtual machines.

2 https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview?toc=%2fazure%2fstorage%2fblobs%2ftoc.json

Move data to Blob storage

A number of solutions exist for migrating existing data to Blob storage:

- **AzCopy** is an easy-to-use command-line tool for Windows and Linux that copies data to and from Blob storage, across containers, or across storage accounts.
- The **Azure Storage Data Movement library** is a .NET library for moving data between Azure Storage services. The AzCopy utility is built with the Data Movement library.
- **Azure Data Factory** supports copying data to and from Blob storage by using the account key, shared access signature, service principal, or managed identities for Azure resources authentications.
- **Blobfuse** is a virtual file system driver for Azure Blob storage. You can use blobfuse to access your existing block blob data in your Storage account through the Linux file system.
- **Azure Data Box Disk** is a service for transferring on-premises data to Blob storage when large datasets or network constraints make uploading data over the wire unrealistic. You can use Azure Data Box Disk to request solid-state disks (SSDs) from Microsoft. You can then copy your data to those disks and ship them back to Microsoft to be uploaded into Blob storage.
- The **Azure Import/Export service** provides a way to export large amounts of data from your storage account to hard drives that you provide and that Microsoft then ships back to you with your data.

Azure Blob Storage Tiers

Overview

Azure storage offers different storage tiers which allow you to store Blob object data in the most cost-effective manner. The available tiers include:

- **Hot storage**: is optimized for storing data that is accessed frequently.
- **Cool storage** is optimized for storing data that is infrequently accessed and stored for at least 30 days.
- **Archive storage** is optimized for storing data that is rarely accessed and stored for at least 180 days with flexible latency requirements (on the order of hours).

The following considerations accompany the different storage tiers:

- The Archive storage tier is only available at the blob level and not at the storage account level.
- Data in the Cool storage tier can tolerate slightly lower availability, but still requires high durability and similar time-to-access and throughput characteristics as Hot data. For Cool data, a slightly lower availability SLA and higher access costs compared to Hot data are acceptable trade-offs for lower storage costs.
- Archive storage is offline and offers the lowest storage costs but also the highest access costs.
- Only the Hot and Cool storage tiers can be set at the account level. Currently the Archive tier cannot be set at the account level.
- Hot, Cool, and Archive tiers can be set at the object level.

Data stored in the cloud grows at an exponential pace. To manage costs for your expanding storage needs, it's helpful to organize your data based on attributes like frequency-of-access and planned retention period to optimize costs. Data stored in the cloud can be different in terms of how it is generated, processed, and accessed over its lifetime. Some data is actively accessed and modified throughout its

lifetime. Some data is accessed frequently early in its lifetime, with access dropping drastically as the data ages. Some data remains idle in the cloud and is rarely, if ever, accessed once stored.

Each of these data access scenarios benefits from a different storage tier that is optimized for a particular access pattern. With Hot, Cool, and Archive storage tiers, Azure Blob storage addresses this need for differentiated storage tiers with separate pricing models.

Storage accounts that support tiering

You may only tier your object storage data to Hot, Cool, or Archive in Blob storage or General Purpose v2 (GPv2) accounts. General Purpose v1 (GPv1) accounts do not support tiering. However, customers can easily convert their existing GPv1 or Blob storage accounts to GPv2 accounts through a simple one-click process in the Azure portal. GPv2 provides a new pricing structure for blobs, files, and queues, and access to a variety of other new storage features as well. Furthermore, going forward some new features and prices cuts will only be offered in GPv2 accounts. Therefore, customers should evaluate using GPv2 accounts but only use them after reviewing the pricing for all services as some workloads can be more expensive on GPv2 than GPv1.

Blob storage and GPv2 accounts expose the **Access Tier** attribute at the account level, which allows you to specify the default storage tier as Hot or Cool for any blob in the storage account that does not have an explicit tier set at the object level. For objects with the tier set at the object level, the account tier will not apply. The Archive tier can only be applied at the object level. You can switch between these storage tiers at any time.

Hot access tier

Hot storage has higher storage costs than Cool and Archive storage, but the lowest access costs. Example usage scenarios for the Hot storage tier include:

- Data that is in active use or expected to be accessed (read from and written to) frequently.
- Data that is staged for processing and eventual migration to the Cool storage tier.

Cool access tier

Cool storage tier has lower storage costs and higher access costs compared to Hot storage. This tier is intended for data that will remain in the Cool tier for at least 30 days. Example usage scenarios for the Cool storage tier include:

- Short-term backup and disaster recovery datasets.
- Older media content not viewed frequently anymore but is expected to be available immediately when accessed.
- Large data sets that need to be stored cost effectively while more data is being gathered for future processing. (*For example*, long-term storage of scientific data, raw telemetry data from a manufacturing facility)

Archive access tier

Archive storage has the lowest storage cost and higher data retrieval costs compared to Hot and Cool storage. This tier is intended for data that can tolerate several hours of retrieval latency and will remain in the Archive tier for at least 180 days.

While a blob is in Archive storage, it is offline and cannot be read (except the metadata, which is online and available), copied, overwritten, or modified. Nor can you take snapshots of a blob in Archive storage. However, you may use existing operations to delete, list, get blob properties/metadata, or change the tier of your blob.

Example usage scenarios for the Archive storage tier include:

- Long-term backup, secondary backup, and archival datasets
- Original (raw) data that must be preserved, even after it has been processed into final usable form. (For *example*, Raw media files after transcoding into other formats)
- Compliance and archival data that needs to be stored for a long time and is hardly ever accessed. (*For example*, Security camera footage, old X-Rays/MRIs for healthcare organizations, audio recordings, and transcripts of customer calls for financial services)

Blob rehydration

To read data in Archive storage, you must first change the tier of the blob to Hot or Cool. This process is known as rehydration and can take up to 15 hours to complete. Large blob sizes are recommended for optimal performance. Rehydrating several small blobs concurrently may add additional time.

During rehydration, you may check the **Archive Status** blob property to confirm if the tier has changed. The status reads “rehydrate-pending-to-hot” or "rehydrate-pending-to-cool" depending on the destination tier. Upon completion, the Archive status property is removed, and the **Access Tier** blob property reflects the new Hot or Cool tier.

Comparison of the storage tiers

The following table shows a comparison of the Hot, Cool, and Archive storage tiers.

	Hot storage tier	Cool storage tier	Archive storage tier
Availability	99.9%	99%	N/A
Availability			
(RA-GRS reads)	99.99%	99.9%	N/A
Usage charges	Higher storage costs, lower access and transaction costs	Lower storage costs, higher access and transaction costs	Lowest storage costs, highest access and transaction costs
Minimum object size	N/A	N/A	N/A
Minimum storage duration	N/A	30 days (GPv2 only)	180 days
Latency			
(Time to first byte)	milliseconds	milliseconds	< 15 hrs
Scalability and performance targets	Same as general-purpose storage accounts	Same as general-purpose storage accounts	Same as general-purpose storage accounts

Understanding Block Blobs, Append Blobs, and Page Blobs

he storage service offers three types of blobs, *block blobs*, *append blobs*, and *page blobs*. You specify the blob type when you create the blob. Once the blob has been created, its type cannot be changed, and it

can be updated only by using operations appropriate for that blob type, i.e., writing a block or list of blocks to a block blob, appending blocks to a append blob, and writing pages to a page blob.

All blobs reflect committed changes immediately. Each version of the blob has a unique tag, called an *ETag*, that you can use with access conditions to assure you only change a specific instance of the blob.

Any blob can be leased for exclusive write access. When a blob is leased, only calls that include the current lease ID can modify the blob or (for block blobs) its blocks. Any blob can be duplicated in a snapshot.

Note: Blobs in the *Azure storage emulator* are limited to a maximum size of 2 GB.

About Block Blobs

Block blobs let you upload large blobs efficiently. Block blobs are comprised of blocks, each of which is identified by a block ID. You create or modify a block blob by writing a set of blocks and committing them by their block IDs. Each block can be a different size, up to a maximum of 100 MB (4 MB for requests using REST versions before 2016-05-31), and a block blob can include up to 50,000 blocks. The maximum size of a block blob is therefore slightly more than 4.75 TB (100 MB X 50,000 blocks). For REST versions before 2016-05-31, the maximum size of a block blob is a little more than 195 GB (4 MB X 50,000 blocks). If you are writing a block blob that is no more than 256 MB (64 MB for requests using REST versions before 2016-05-31) in size, you can upload it in its entirety with a single write operation.

Storage clients default to a 128 MB maximum single blob upload, settable using the `SingleBlobUploadThresholdInBytes` property of the `BlobRequestOptions` object. When a block blob upload is larger than the value in this property, storage clients break the file into blocks. You can set the number of threads used to upload the blocks in parallel on a per-request basis using the `ParallelOperationThreadCount` property of the `BlobRequestOptions` object.

When you upload a block to a blob in your storage account, it is associated with the specified block blob, but it does not become part of the blob until you commit a list of blocks that includes the new block's ID. New blocks remain in an uncommitted state until they are specifically committed or discarded. Writing a block does not update the last modified time of an existing blob.

Block blobs include features that help you manage large files over networks. With a block blob, you can upload multiple blocks in parallel to decrease upload time. Each block can include an MD5 hash to verify the transfer, so you can track upload progress and re-send blocks as needed. You can upload blocks in any order, and determine their sequence in the final block list commitment step. You can also upload a new block to replace an existing uncommitted block of the same block ID. You have one week to commit blocks to a blob before they are discarded. All uncommitted blocks are also discarded when a block list commitment operation occurs but does not include them.

You can modify an existing block blob by inserting, replacing, or deleting existing blocks. After uploading the block or blocks that have changed, you can commit a new version of the blob by committing the new blocks with the existing blocks you want to keep using a single commit operation. To insert the same range of bytes in two different locations of the committed blob, you can commit the same block in two places within the same commit operation. For any commit operation, if any block is not found, the entire commitment operation fails with an error, and the blob is not modified. Any block commitment overwrites the blob's existing properties and metadata, and discards all uncommitted blocks.

Block IDs are strings of equal length within a blob. Block client code usually uses base-64 encoding to normalize strings into equal lengths. When using base-64 encoding, the pre-encoded string must be 64 bytes or less. Block ID values can be duplicated in different blobs. A blob can have up to 100,000 uncommitted blocks, but their total size cannot exceed 200,000 MB.

If you write a block for a blob that does not exist, a new block blob is created, with a length of zero bytes. This blob will appear in blob lists that include uncommitted blobs. If you don't commit any block to this blob, it and its uncommitted blocks will be discarded one week after the last successful block upload. All uncommitted blocks are also discarded when a new blob of the same name is created using a single step (rather than the two-step block upload-then-commit process).

About Page Blobs

Page blobs are a collection of 512-byte pages optimized for random read and write operations. To create a page blob, you initialize the page blob and specify the maximum size the page blob will grow. To add or update the contents of a page blob, you write a page or pages by specifying an offset and a range that align to 512-byte page boundaries. A write to a page blob can overwrite just one page, some pages, or up to 4 MB of the page blob. Writes to page blobs happen in-place and are immediately committed to the blob. The maximum size for a page blob is 8 TB.

About Append Blobs

An append blob is comprised of blocks and is optimized for append operations. When you modify an append blob, blocks are added to the end of the blob only, via the `Append Block` operation. Updating or deleting of existing blocks is not supported. Unlike a block blob, an append blob does not expose its block IDs.

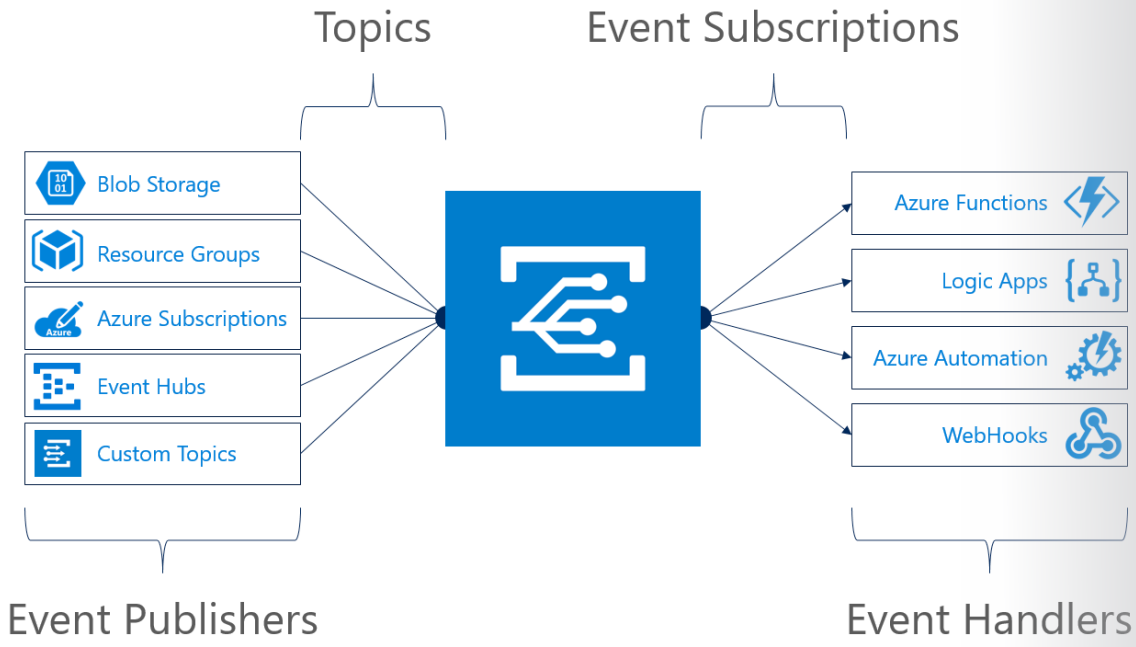
Each block in an append blob can be a different size, up to a maximum of 4 MB, and an append blob can include up to 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GB (4 MB X 50,000 blocks).

Reacting to Blob Storage Events

Azure Storage events allow applications to react to the creation and deletion of blobs using modern serverless architectures. It does so without the need for complicated code or expensive and inefficient polling services. Instead, events are pushed through Azure Event Grid to subscribers such as Azure Functions, Azure Logic Apps, or even to your own custom http listener, and you only pay for what you use.

Blob storage events are reliably sent to the Event grid service which provides reliable delivery services to your applications through rich retry policies and dead-letter delivery.

Common Blob storage event scenarios include image or video processing, search indexing, or any file-oriented workflow. Asynchronous file uploads are a great fit for events. When changes are infrequent, but your scenario requires immediate responsiveness, event-based architecture can be especially efficient.



Blob storage accounts

Blob storage events are available in general-purpose v2 storage accounts and Blob storage accounts. General-purpose v2 storage accounts support all features for all storage services, including Blobs, Files, Queues, and Tables. A Blob storage account is a specialized storage account for storing your unstructured data as blobs (objects) in Azure Storage. Blob storage accounts are like general-purpose storage accounts and share all the great durability, availability, scalability, and performance features that you use today including 100% API consistency for block blobs and append blobs.

Available Blob storage events

Event grid uses event subscriptions to route event messages to subscribers. Blob storage event subscriptions can include two types of events:

Event Name	Description
`Microsoft.Storage.BlobCreated`	Fired when a blob is created or replaced through the `PutBlob`, `PutBlockList`, or `CopyBlob` operations
`Microsoft.Storage.BlobDeleted`	Fired when a blob is deleted through a `DeleteBlob` operation

Event Schema

Blob storage events contain all the information you need to respond to changes in your data. You can identify a Blob storage event because the eventType property starts with "Microsoft.Storage". Additional information about the usage of Event Grid event properties is documented in **Event Grid event schema**[3].

3 https://docs.microsoft.com/en-us/azure/event-grid/event-schema

Filtering events

Blob event subscriptions can be filtered based on the event type and by the container name and blob name of the object that was created or deleted. Filters can be applied to event subscriptions either during the creation of the event subscription or at a later time. Subject filters in Event Grid work based on "begins with" and "ends with" matches, so that events with a matching subject are delivered to the subscriber. The subject of Blob storage events uses the format:

```
/blobServices/default/containers/<containername>/blobs/<blobname>
```

To match all events for a storage account, you can leave the subject filters empty. To match events from blobs created in a set of containers *sharing* a prefix, use a `subjectBeginsWith` filter like:

```
/blobServices/default/containers/containerprefix
```

To match events from blobs created in *specific* container, use a `subjectBeginsWith` filter like:

```
/blobServices/default/containers/containername/
```

To match events from blobs created in specific container sharing a blob name prefix, use a `subjectBeginsWith` filter like:

```
/blobServices/default/containers/containername/blobs/blobprefix
```

To match events from blobs created in specific container sharing a blob suffix, use a `subjectEndsWith` filter like ".log" or ".jpg".

Practices for consuming events

Applications that handle Blob storage events should follow a few recommended practices:

- As multiple subscriptions can be configured to route events to the same event handler, it is important not to assume events are from a particular source, but to check the topic of the message to ensure that it comes from the storage account you are expecting.
- Similarly, check that the eventType is one you are prepared to process, and do not assume that all events you receive will be the types you expect.
- As messages can arrive out of order and after some delay, use the etag fields to understand if your information about objects is still up-to-date. Also, use the sequencer fields to understand the order of events on any particular object.
- Use the blobType field to understand what type of operations are allowed on the blob, and which client library types you should use to access the blob. Valid values are either `BlockBlob` or `PageBlob`.
- Use the url field with the `CloudBlockBlob` and `CloudAppendBlob` constructors to access the blob.
- Ignore fields you don't understand. This practice will help keep you resilient to new features that might be added in the future.

Shared Access Signatures

A Shared Access Signature (SAS) is a URI that grants restricted access rights to containers, binary large objects (blobs), queues, and tables for a specific time interval. By providing a client with a Shared Access

Signature, you can enable them to access resources in your storage account without sharing your account key with them.

The Shared Access Signature URI query parameters incorporate all of the information necessary to grant controlled access to a storage resource. The URI query parameters specify the time interval over which the Shared Access Signature is valid, the permissions that it grants, the resource that is to be made available, and the signature that the storage services should use to authenticate the request.

Here is an example of a SAS URI that provides read and write permissions to a blob. The table breaks down each part of the URI to understand how it contributes to the SAS:

```
https://myaccount.blob.core.windows.net/sascontainer/sasblob.txt?sv=2012-02
-12&st=2013-04-29T22%3A18%3A26Z&se=2013-04-30T02%3A23%3A26Z&s-
r=b&sp=rw&sig=Z%2FRHIX5Xcg0Mq2rqI3OlWTjEg2tYkboXr1P9ZUXDtkk%3
```

Component	Content	Description
Blob URI	https://myaccount.blob.core.windows.net/sascontainer/sasblob.txt	The address of the blob. Note that using HTTPS is highly recommended.
Storage services version	sv=2012-02-12	For Azure Storage services version 2012-02-12 and later, this parameter indicates the version to use.
Start time	st=2013-04-29T22%3A18%3A26Z	Specified in an International Organization for Standardization (ISO) 8061 format. If you want the SAS to be valid immediately, omit the start time.
Expiration time	se=2013-04-30T02%3A23%3A26Z	Specified in an ISO 8061 format.
Resource	sr=b	The resource is a blob.
Permissions	sp=rw	The permissions granted by the SAS include Read (r) and Write (w).
Signature	sig=Z%2FRHIX5Xcg0Mq2rqI3Ol-WTjEg2tYkboXr1P9ZUXDtkk%3D	Used to authenticate access to the blob. The signature is a HMAC function computed over a string to sign and a key by using the SHA256 algorithm and then encoded by using Base64 encoding.

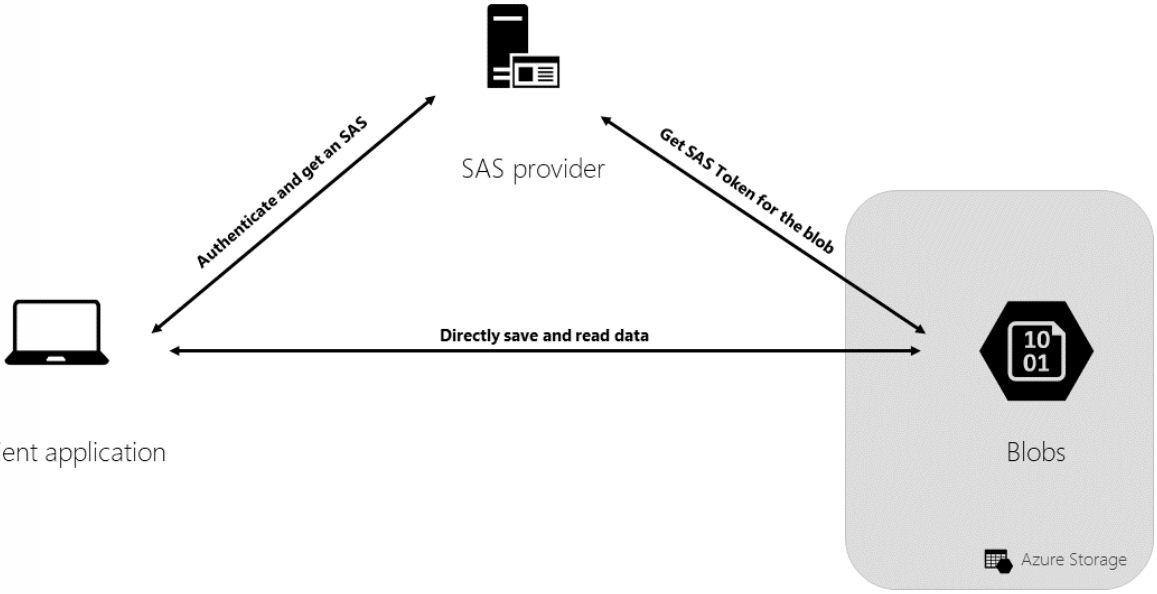
Valet key pattern that uses Shared Access Signatures

A common scenario where an SAS is useful is a service where users read and write their own data to your storage account. In a scenario where a storage account stores user data, there are two typical design patterns:

- Clients upload and download data via a front-end proxy service, which performs authentication. This front-end proxy service has the advantage of allowing the validation of business rules, but for large

amounts of data or high-volume transactions, creating a service that can scale to match demand might be expensive or difficult.

- Using the valet key pattern, a lightweight service authenticates the client as needed and then generates an SAS. After the client receives the SAS, they can access storage account resources directly with the permissions defined by the SAS and for the interval allowed by the SAS. The SAS mitigates the need for routing all data through the front-end proxy service.



Stored access policies

A Shared Access Signature can take one of two forms:

- **An ad hoc SAS.** When you create an ad hoc SAS, the start time, expiration time, and permissions for the SAS are all specified on the SAS URI (or implied in the case where the start time is omitted). This type of SAS can be created on a container, blob, table, or queue.
- **An SAS with a stored access policy.** A stored access policy is defined on a resource container—a blob container, table, or queue—and can be used to manage constraints for one or more Shared Access Signatures. When you associate an SAS with a stored access policy, the SAS inherits the constraints—the start time, expiration time, and permissions—defined for the stored access policy.

The difference between the two forms is important for one key scenario: revocation. An SAS is a URL, so anyone who obtains the SAS can use it regardless of who requested it to begin with. If an SAS is published publicly, it can be used by anyone in the world. Stored access policies give you the option to revoke permissions without having to regenerate the storage account keys. Set the expiration on these to be a very long time (or infinite), and make sure that it is regularly updated to move it further into the future.

Setting and Retrieving Properties and Metadata for Blob Resources by using REST

Containers and blobs support custom metadata, represented as HTTP headers. Metadata headers can be set on a request that creates a new container or blob resource, or on a request that explicitly creates a property on an existing resource.

Metadata Header Format

Metadata headers are name/value pairs. The format for the header is:

```
x-ms-meta-name:string-value
```

Beginning with version 2009-09-19, metadata names must adhere to the naming rules for C# identifiers.

Names are case-insensitive. Note that metadata names preserve the case with which they were created, but are case-insensitive when set or read. If two or more metadata headers with the same name are submitted for a resource, the Blob service returns status code 400 (Bad Request).

The metadata consists of name/value pairs. The total size of all metadata pairs can be up to 8KB in size.

Metadata name/value pairs are valid HTTP headers, and so they adhere to all restrictions governing HTTP headers.

Operations on Metadata

Metadata on a blob or container resource can be retrieved or set directly, without returning or altering the content of the resource.

Note that metadata values can only be read or written in full; partial updates are not supported. Setting metadata on a resource overwrites any existing metadata values for that resource.

Retrieving Properties and Metadata

The GET and HEAD operations both retrieve metadata headers for the specified container or blob. These operations return headers only; they do not return a response body. The URI syntax for retrieving metadata headers on a container is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer?restype=container
```

The URI syntax for retrieving metadata headers on a blob is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metadata
```

Setting Metadata Headers

The PUT operation sets metadata headers on the specified container or blob, overwriting any existing metadata on the resource. Calling PUT without any headers on the request clears all existing metadata on the resource.

The URI syntax for setting metadata headers on a container is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer?comp=metadata-?restype=container
```

The URI syntax for setting metadata headers on a blob is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metada-
ta
```

Standard HTTP Properties for Containers and Blobs

Containers and blobs also support certain standard HTTP properties. Properties and metadata are both represented as standard HTTP headers; the difference between them is in the naming of the headers. Metadata headers are named with the header prefix `x-ms-meta-` and a custom name. Property headers use standard HTTP header names, as specified in the Header Field Definitions section 14 of the HTTP/1.1 protocol specification.

The standard HTTP headers supported on containers include:

- `ETag`
- `Last-Modified`

The standard HTTP headers supported on blobs include:

- `ETag`
- `Last-Modified`
- `Content-Length`
- `Content-Type`
- `Content-MD5`
- `Content-Encoding`
- `Content-Language`
- `Cache-Control`
- `Origin`
- `Range`

Manipulating Blob Container Properties in .NET

The `CloudStorageAccount` class contains the `CreateCloudBlobClient` method that gives you programmatic access to a client that manages your file shares:

```
CloudBlobClient client = storageAccount.CreateCloudBlobClient();
```

To reference a specific blob container, you can use the `GetContainerReference` method of the `CloudBlobClient` class:

```
CloudBlobContainer container = client.GetContainerReference("images");
```

After you have a reference to the container, you can ensure that the container exists. This will create the container if it does not already exist in the Azure storage account:

```
container.CreateIfNotExists();
```

With a hydrated reference, you can perform actions such as fetching the properties (metadata) of the container by using the `FetchAttributesAsync` method of the `CloudBlobContainer` class:

```
await container.FetchAttributesAsync();
```

After the method is invoked, the local variable is hydrated with values for various container metadata. This metadata can be accessed by using the `Properties` property of the `CloudBlobContainer` class, which is of type `BlobContainerProperties`:

```
container.Properties
```

This class has properties that can be set to change the container, including (but not limited to) those in the following table.

Property	Description
`ETag`	This is a standard HTTP header that gives a value that is unchanged unless a property of the container is changed. This value can be used to implement optimistic concurrency with the blob containers.
`LastModified`	This property indicates when the container was last modified.
`PublicAccess`	This property indicates the level of public access that is allowed on the container. Valid values include Blob, Container, Off, and Unknown.
`HasImmutabilityPolicy`	This property indicates whether the container has an immutability policy. An immutability policy will help ensure that blobs are stored for a minimum amount of retention time.
`HasLegalHold`	This property indicates whether the container has an active legal hold. A legal hold will help ensure that blobs remain unchanged until the hold is removed.

Manipulating Blob Container Metadata in .NET

Using the existing **CloudBlobContainer** variable (named *container*), you can set and retrieve custom metadata for the container instance. This metadata is hydrated when you call the `FetchAttributes` or `FetchAttributesAsync` method on your blob or container to populate the Metadata collection.

The following code example sets metadata on a container. In this example, we use the collection's Add method to set a metadata value:

```
container.Metadata.Add("docType", "textDocuments");
```

In the next example, we set the metadata value by using implicit key/value syntax:

```
container.Metadata["category"] = "guidance";
```

To persist the newly set metadata, you must call the SetMetadataAsync method of the `CloudBlobContainer` class:

```
await container.SetMetadataAsync();
```

Review Questions

Module 6 - Review Questions

Shared Access Signature

A Shared Access Signature (SAS) is a URI that grants restricted access rights to containers, binary large objects (blobs), queues, and tables for a specific time interval. By providing a client with a Shared Access Signature, you can enable them to access resources in your storage account without sharing your account key with them. What are the two forms an SAS can take?

> Click to see suggested answer

- **An ad hoc SAS.** When you create an ad hoc SAS, the start time, expiration time, and permissions for the SAS are all specified on the SAS URI (or implied in the case where the start time is omitted). This type of SAS can be created on a container, blob, table, or queue.
- **An SAS with a stored access policy.** A stored access policy is defined on a resource container—a blob container, table, or queue—and can be used to manage constraints for one or more Shared Access Signatures. When you associate an SAS with a stored access policy, the SAS inherits the constraints—the start time, expiration time, and permissions—defined for the stored access policy.

Copying Blobs between containers

Like with Azure Files, you can use AzCopy to copy blobs between storage containers. By default, does AzCopy copy data synchronously or asynchronously?

> Click to see suggested answer

By default, AzCopy copies data between two storage endpoints asynchronously. Therefore, the copy operation runs in the background by using spare bandwidth capacity that has no Service Level Agreement (SLA) in terms of how fast a blob is copied, and AzCopy periodically checks the copy status until the copying has completed or failed.

Azure SQL database

SQL Database is a general-purpose relational database managed service in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML. What are the three deployment options for Azure SQL database?

> Click to see suggested answer

Azure SQL Database provides the following deployment options for an Azure SQL database:

- As a single database with its own set of resources managed via a logical server
- As a pooled database in an elastic pool with a shared set of resources managed via a logical server
- As a part of a collection of databases known as a managed instance that contains system and user databases and sharing a set of resources

DbContext implementation

To use Entity Framework to query, insert, update, and delete data using .NET objects, you first need to create a model that maps the entities and relationships defined in your model to tables in a database. After you have a model, the primary class your application interacts with is `System.Data.Entity.DbContext` (often referred to as the context class). You can use a `DbContext` class associated to a model to (name as many as you can):

> Click to see suggested answer

- Write and execute queries.
- Materialize query results as entity objects.
- Track changes that are made to those objects.
- Persist object changes back on the database.
- Bind objects in memory to UI controls.

The recommended way to work with the context is to define a class that derives from `DbContext` and exposes `DbSet` properties that represent collections of the specified entities in the context.

Azure Cosmos DB Core functionality

Azure Cosmos DB has a feature referred to as turnkey global distribution that automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure. Can you name, and describe, the different consistency levels?

> Click to see suggested answer

Consistency Level	Description
Strong	When a write operation is performed on your primary database, the write operation is replicated to the replica instances. The write operation is committed (and visible) on the primary only after it has been committed and confirmed by all replicas.
Bounded Stateless	This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.
Session	This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.

Consistency Level	Description
Consistent Prefix	This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.
Eventual	This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary. This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize it's transactions.

Partitioning

Azure Cosmos DB provides containers for storing data called collections (for documents), graphs, or tables. Containers are logical resources and can span one or more physical partitions or servers. Can you describe the differences between physical and logical partitions?

> Click to see suggested answer

A *physical partition* is a fixed amount of reserved solid-state drive (SSD) backend storage combined with a variable amount of compute resources (CPU and memory). Each physical partition is replicated for high availability.

A *logical partition* is a partition within a physical partition that stores all the data associated with a single partition key value. Partition ranges can be dynamically subdivided to seamlessly grow the database as the application grows while simultaneously maintaining high availability.