Exam AZ-204: Developing Solutions for Microsoft Azure Master Cheat Sheet

References

Microsoft Certifications - AZ-204

Microsoft Learning Labs - AZ-204

Topics:

1. Create serverless applications

1. Choose the best Azure service to automate your business processes
2. Create serverless logic with Azure Functions
3. Execute an Azure Function with triggers
4. Chain Azure Functions together using input and output bindings
5. Create a long-running serverless workflow with Durable Functions
6. Develop, test, and publish Azure Functions by using Azure Functions Core Tools
7. Develop, test, and deploy an Azure Function with Visual Studio
8. Monitor GitHub events by using a webhook with Azure Functions
9. Enable automatic updates in a web application using Azure Functions and SignalR Service

2. Connect your services together

1. Choose a messaging model in Azure to loosely connect your services
2. Implement message-based communication workflows with Azure Service Bus
3. Communicate between applications with Azure Queue storage
4. Enable reliable messaging for Big Data applications using Azure Event Hubs

3. Work with relational data in Azure

1. Provision an Azure SQL database to store application data
2. Create an Azure Database for PostgreSQL server
3. Scale multiple Azure SQL Databases with SQL elastic pools
4. Secure your Azure SQL Database
5. Develop and configure an ASP.NET application that queries an Azure SQL database

4. Store data in Azure

1. Choose a data storage approach in Azure
2. Create an Azure Storage account
3. Connect an app to Azure Storage
4. Secure your Azure Storage account
5. Store application data with Azure Blob storage

5. Deploy a website with Azure virtual machines

1. Introduction to Azure virtual machines
2. Create a Linux virtual machine in Azure
3. Create a Windows virtual machine in Azure
4. Build and run a web application with the MEAN stack on an Azure Linux virtual machine

1. Create Serverless Applications

1. Choose the best Azure service to automate your business processes

Business processes modeled in software are often called **workflows**. Azure includes four different technologies that you can use to build and implement workflows that integrate multiple systems

- Logic Apps
- Microsoft Power Automate
- WebJobs
- Azure Functions

These four technologies have the following similarities

- Accept Inputs
- Run Actions
- Include Condition
- Produce Output
- Triggered on a schedule or other external events

Design-First Approach
It includes user interface in which you can design the workflow and includes the following technologies:

- **Logic Apps**
  A service that you can use to automate, orchestrate and integrate disparate components of a distributed application. You can use Logic Apps Designer to define the workflow. Alternatively, you may prefer to code the workflow in JSON notations using the code-view.
  A connector is a Logic Apps component that provides an interface to an external service. Logic Apps provides hundreds of pre-built connectors that you can use to create your apps. If you have an unusual or unique system that you want to call from a Logic Apps, you can create your own connector if your system exposes a REST API.

- **Microsoft Power Automate**
  A service that you can use to create workflows even when you have no IT Pro experience. You can create workflows that integrate and orchestrate many different components by using the web or mobile app.
  Under the hood, Microsoft Power Automate is built on Logic Apps. This fact means that Power Automate supports the same range of connectors and actions. You can also use custom connectors in Microsoft Power Automate.

Design-First technologies compared

	Microsoft Power Automate	**Logic Apps**
Intended users	Office workers and business analysts	Developers and IT pros
Intended scenarios	GUI only. Browser and mobile app	Browser and Visual Studio designer. Code editing is possible
Application Lifecycle Management	Power Automate includes testing and production environments	Logic Apps source code can be included in Azure DevOps and source code management systems

If you choose a design-first approach, the workflow is visualized in an easy-to-understand design surface. In addition, that diagram is not a separate document, but a picture of the process as it is implemented. The benefit is that there's no possibility that the diagram is not updated when the process is changed.

Code-First Approach
When you need more control over the performance of your workflow or need to write custom code as part of your business process, the following technologies would help:

- **Azure WebJobs**
  WebJobs are a part of the Azure App Service that you can use to run a program or script automatically. The two kinds are:
    - Continous
    - Triggered

You can create a Webjob by using Shell Scripts (Windows, Powershell, Bash) or by writing a program in PHP, Python, Node.js, or Java. You can also program a WebJob by using the .NET Framework or the .NET Core Framework and a .NET language such as C# or VB.NET.
Along with C# .NET you can use the WebJobs SDK which includes a range of classes, such as JobHostConfiguration and HostBuilder, which reduce the amount of code required to interact with the Azure App Service.

- **Azure Functions** An Azure Function is a simple way for you to run small pieces of code in the cloud, without having to worry about the infrastructure required to host that code. You can write the Function in C#, Java, JavaScript, PowerShell, Python, or any of the languages that are listed in the Supported languages in Azure Functions article. In addition, with the consumption plan option, you only pay for the time when the code runs. Azure automatically scales your function in response to the demand from users.

  Following is the list of function triggers:

  - HTTPTrigger
  - TimerTrigger
  - BlobTrigger
  - CosmosDBTrigger

  Azure Functions can integrate with many different services both within Azure and from third parties. These services can trigger your function, or send data input to your function, or receive data output from your function.

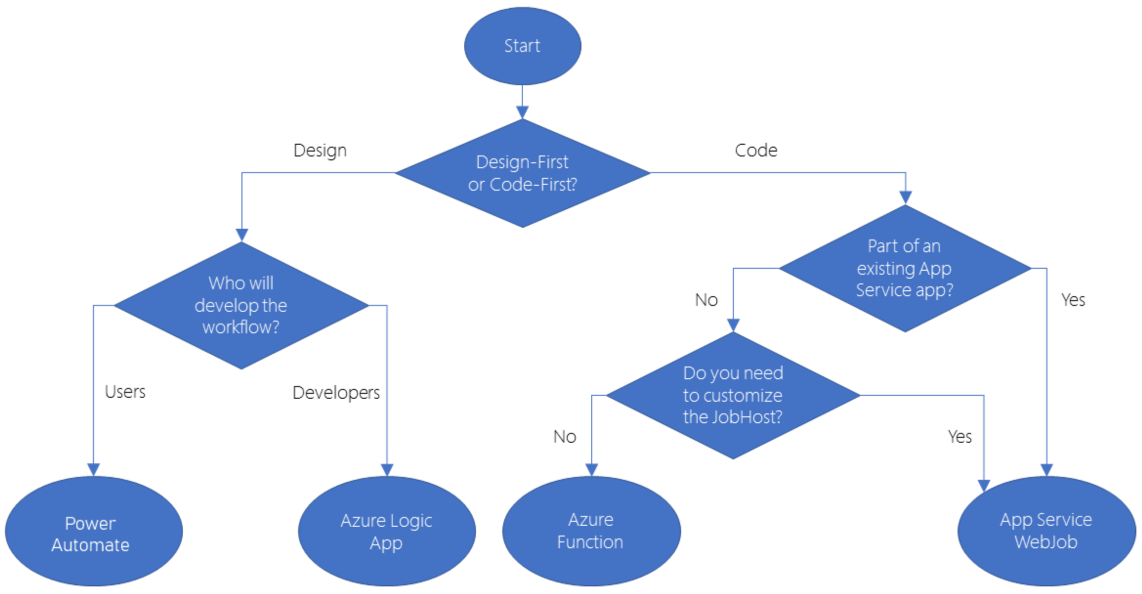
Code-First technologies compared

In most cases, the simple administration and more flexible coding model provided by Azure Functions may lead you to choose them in preference to WebJobs. However, you may choose WebJobs for the following reasons:

- You want the code to be a part of an existing App Service application and to be managed as part of that application, for example in the same Azure DevOps environment.
- You need close control over the object that listens for events that trigger the code. This object in question is the JobHost class, and you have more flexibility to modify its behavior in WebJobs.

	Azure WebJobs	**Azure Functions**
Supported languages	C# if you are using the WebJobs SDK	C#, Java, JavaScript, PowerShell, etc.
Automatic scaling	No	Yes
Development and testing in a browser	No	Yes
Pay-per-use pricing	No	Yes
Integration with Logic Apps	No	Yes
Package managers	NuGet if you are using the WebJobs SDK	Nuget and NPM
Can be part of an App Service application	Yes	No
Provides close control of JobHost	Yes	No

I you choose a code-first approach, you can develop a complex business logic and wrap the solution in a custom connector which can be integrated with Logic Apps or Power Automate. As a developer, you get more flexibility by this approach.


Start
Design-First or Code-First?
Design
Code
Who will develop the workflow?
Users
Developers
Power Automate
Azure Logic App
Part of an existing App Service app?
No
Yes
Do you need to customize the JobHost?
No
Yes
Azure Function
App Service WebJob

2. Create serverless logic with Azure Functions

What is serverless compute? Serverless compute can be thought of as a function as a service (FaaS), or a microservice that is hosted on a cloud platform. Your business logic runs as functions and you don't have to manually provision or scale infrastructure. The cloud provider manages infrastructure. Your app is automatically scaled out or down depending on load. Azure has several ways to build this sort of architecture. The two most common approaches are Azure Logic Apps and Azure Functions, which we focus on in this module.

Benefits of serverless compute solution

Avoids over-allocation of infrastructure - Suppose you've provisioned VM servers and configured them with enough resources to handle your peak load times. When the load is light, you are potentially paying for infrastructure you're not using. Serverless computing helps solve the allocation problem by scaling up or down automatically, and you're only billed when your function is processing work.

Stateless logic - Stateless functions are great candidates for serverless compute; function instances are created and destroyed on demand. If state is required, it can be stored in an associated storage service.

Event driven - Functions are event driven. This means they run only in response to an event (called a "trigger"), such as receiving an HTTP request, or a message being added to a queue. You configure a trigger as part of the function definition. This approach simplifies your code by allowing you to declare where the data comes from (trigger/input binding) and where it goes (output binding).

Funtions can be used in traditional compute environments - Functions are a key component of serverless computing, but they are also a general compute platform for executing any type of code. Should the needs of your app change, you can take your project and deploy it in a non-serverless environment, which gives you the flexibility to manage scaling, run on virtual networks, and even completely isolate your functions.

Drawbacks of serverless compute solution

Execution time - By default, functions have a timeout of 5 minutes. This timeout is configurable to a maximum of 10 minutes. If your function requires more than 10 minutes to execute, you can host it on a VM. Additionally, if your service is initiated through an HTTP request and you expect that value as an HTTP response, the timeout is further restricted to 2.5 minutes. Finally, there's also an option called Durable Functions that allows you to orchestrate the executions of multiple functions without any timeout.

Execution frequency - The second characteristic is execution frequency. If you expect your function to be executed continuously by multiple clients, it would be prudent to estimate the usage and calculate the cost of using functions accordingly. It might be cheaper to host your service on a VM.

While scaling, only one function app instance can be created every 10 seconds, for up to 200 total instances. Keep in mind, each instance can service multiple concurrent executions, so there is no set limit on how much traffic a single instance can handle. Different types of triggers have different scaling requirements, so research your choice of trigger and investigate its limits.

Exercise - Create a function app in the Azure portal

Choosing a service plan

Consumption service plan - This is the plan that you choose when using the Azure serverless application platform. The Consumption service plan provides automatic scaling and bills you when your functions are running. The Consumption plan comes with a configurable timeout period for the execution of a function. By default, it is 5 minutes, but may be configured to have a timeout as long as 10 minutes.

Azure App Service plan - This plan allows you to avoid timeout periods by having your function run continuously on a VM that you define. When using an App Service plan, you are responsible for managing the app resources the function runs on, so this is technically not a serverless plan. However, it may be a better choice if your functions are used continuously or if your functions require more processing power or execution time than the Consumption plan can provide.

Storage account requirements

When you create a function app, it must be linked to a storage account. You can select an existing account or create a new one. The function app uses this storage account for internal operations such as logging function executions and managing execution triggers. On the Consumption service plan, this is also where the function code and configuration file are stored.

Triggers
The type of event that starts the function is called a trigger. You must configure a function with exactly one trigger. Azure supports triggers for the following services:

Service	Trigger Description
Blob storage	Start a function when a new or updated blob is detected.
Azure Cosmos DB	Start a function when inserts and updates are detected.
Event Grid	Start a function when an event is received from Event Grid.
HTTP	Start a function with an HTTP request.
Microsoft Graph Events	Start a function in response to an incoming webhook from the Microsoft Graph. Each instance of this trigger can react to one Microsoft Graph resource type.
Queue storage	Start a function when a new item is received on a queue. The queue message is provided as input to the function.
Service Bus	Start a function in response to messages from a Service Bus queue.
Timer	Start a function on a schedule.

Bindings
Bindings are a declarative way to connect data and services to your function.

Bindings know how to talk to different services, which means you don't have to write code in your function to connect to data sources and manage connections. The platform takes care of that complexity for you as part of the binding code. Each binding has a direction - your code reads data from input bindings and writes data to output bindings. Each function can have zero or more bindings to manage the input and output data processed by the function.

Monitoring dashboard
The Azure portal provides a monitoring dashboard available if you turn on the Application Insights integration. This monitor dashboard provides a quick way to view the log of function executions populated by Application Insights.

Exercise - Add logic to the function app

Securing HTTP triggers
When you create a HTTP triggered function, you select the authorization level. By default, it's set to "Function", which requires a function-specific API key, but it can also be set to "Admin" to use a global "master" key, or "Anonymous" to indicate that no key is required. You can also change the authorization level through the function properties after creation. You should pass the function-specific API key in the header of an HTTP request as 'x-funcion-key' being the key.

3. Execute an Azure Function with triggers

What is a CRON expression? A CRON expression is a string that consists of six fields that represent a set of times.
The order of the six fields in Azure is: {second} {minute} {hour} {day} {month} {day of the week}.
For example, a CRON expression to create a trigger that executes every five minutes looks like: **0 */5 * * * ***

Special characters in CRON expression

Special Character	Meaning	Example
**	Selects every value in a field	An asterisk "*" in the day of the week field means every day.

Special Character	Meaning	Example
,	Separates items in a list	A comma "1,3" in the day of the week field means just Mondays (day 1) and Wednesdays (day 3).
-	Specifies a range	A hyphen "10-12" in the hour field means a range that includes the hours 10, 11, and 12.
/	Specifies an increment	A slash "*/10" in the minutes field means an increment of every 10 minutes.

Exercise - Create a timer trigger

What is an HTTP trigger Authorization level? - An HTTP trigger Authorization level is a flag that indicates if an incoming HTTP request needs an API key for authentication reasons.

There are three Authorization levels:

- Function
- Anonymous
- Admin

The **Function** and **Admin** levels are "key" based. There are two types of keys: function and host. Function keys are specific to a function. Host keys apply to all functions inside the function app. If your Authorization level is set to **Function**, you can use either a function or a host key. If your Authorization level is set to **Admin**, you must supply a host key.
The **Anonymous** level means that there's no authentication required.

Exercise - Create an HTTP trigger

What is Azure Blob storage? - Azure Blob storage is an object storage solution that's designed to store large amounts of unstructured data.
For example, Azure Blob storage is great at doing things like:

- Storing files
- Serving files
- Streaming video and audio
- Logging data

There are three types of blobs: block blobs, append blobs, and page blobs. Block blobs are the most common type. They allow you to store text or binary data efficiently. Append blobs are like block blobs, but they're designed more for append operations like creating a log file that's being constantly updated. Finally, page blobs are made up of pages and are designed for frequent random read and write operations.

4. Chain Azure Functions together using input and output bindings

Types of bindings

- **Input binding** - An input binding is a connection to a data source. Our function can read data from these inputs.
- **Output binding** - An output binding is a connection to a data destination. Our function can write data to these destinations.

Triggers are special types of input bindings that cause a function to execute.

Types of supported bindings

A binding type can be used as an input, an output or both. For example, a function can write to Azure Blob Storage output binding, but a blob storage update could trigger another function.
Some common binding types are listed below:

- Blob Storage
- Azure Service Bus Queues
- Azure Cosmos DB
- Azure Event Hubs
- External Files
- External Tables
- HTTP endpoints

These types are just a sample. There are more, plus functions have an extensibility model to add more bindings.

Binding properties

- **Name** - Defines the function parameter through which you access the data. For example, in a queue input binding, this is the name of the function parameter that receives the queue message content.

- **Type** - Identifies the type of binding, i.e., the type of data or service we want to interact with.
- **Direction** - Indicates the direction data is flowing, i.e., is it an input or output binding?
- **Connection** - Provides the name of an app setting key that contains the connection string. Bindings use connection strings stored in app settings to keep secrets out of the function code. It is only used for bindings which require connection string.

Create a binding
Bindings are defined in JSON. A binding is configured in your function's configuration file, which is named function.json and lives in the same folder as your function code. Following is a sample of binding definition.

```
...
{
  "name": "headshotBlob",
  "type": "blob",
  "path": "thumbnail-images/{filename}",
  "connection": "HeadshotStorageConnection",
  "direction": "in"
},
...
```

Exercise - Explore input and output binding types

What is a binding expression? - A binding expression is specialized text in function.json, function parameters, or code that is evaluated when the function is invoked to yield a value. For example, if you have a Service Bus Queue binding, you could use a binding expression to get the name of the queue from App Settings.

Types of binding expressions

- App settings
- Trigger file name
- Trigger metadata
- JSON payloads
- New GUID
- Current date and time

Most expressions are identified by wrapping them in curly braces. However, app setting binding expressions are wrapped in percent signs rather than curly braces. For example if the blob output binding path is %Environment%/newblob.txt and the Environment app setting value is Development, a blob will be created in the Development container.

Exercise - Read data with input bindings

Exercise - Write data with output bindings

Combining input and output bindings
It's possible to apply multiple bindings to a single function. This allows you to define both input and output bindings, and the input and output can even be the same binding type.

5. Create a long-running serverless workflow with Durable Functions

What is Durable functions? - Durable functions are an extension of Azure Functions. Whereas Azure Functions operate in a stateless environment, Durable Functions can retain state between function calls. This approach enables you to simplify complex stateful executions in a serverless-environment.

Durable Functions scales as needed, and provides a cost effective means of implementing complex workflows in the cloud. Some benefits of using Durable Functions include:

- They enable you to write event driven code. A durable function can wait asynchronously for one or more external events, and then perform a series of tasks in response to these events.
- You can chain functions together. You can implement common patterns such as fan-out/fan-in, which uses one function to invoke others in parallel, and then accumulate the results.
- You can orchestrate and coordinate functions, and specify the order in which functions should execute.
- The state is managed for you. You don't have to write your own code to save state information for a long-running function.

Durable functions allows you to define stateful workflows using an Orchestration function. An orchestration function provides these extra benefits:

- You can define the workflows in code. You don't need to write a JSON description or use a workflow design tool.
- Functions can be called both synchronously and asynchronously. Output from the called functions is saved locally in variables and used in subsequent function calls.
- Azure checkpoints the progress of a function automatically when the function awaits. Azure may choose to dehydrate the function and save its state while

the function waits, to preserve resources and reduce costs. When the function starts running again, Azure will rehydrate it and restore its state.

Function types

You can use three durable function types: client, orchestrator, and activity.

- **Client** functions are the entry point for creating an instance of a Durable Functions orchestration. They can run in response to an event from many sources, such as a new HTTP request arriving, a message being posted to a message queue, an event arriving in an event stream. You can write them in any of the supported languages.
- **Orchestrator** functions describe how actions are executed, and the order in which they are run. You write the orchestration logic in code (C# or JavaScript).
- **Activity** functions are the basic units of work in a durable function orchestration. An activity function contains the actual work performed by the tasks being orchestrated.

Application patterns

You can use Durable Functions to implement many common workflow patterns. These patterns include:

- **Function chaining** - In this pattern, the workflow executes a sequence of functions in a specified order. The output of one function is applied to the input of the next function in the sequence. The output of the final function is used to generate a result.
- **Fan out/fan in** - This pattern runs multiple functions in parallel and then waits for all the functions to finish. The results of the parallel executions can be aggregated or used to compute a final result.
- **Async HTTP APIs** - This pattern addresses the problem of coordinating state of long-running operations with external clients. An HTTP call can trigger the long-running action. Then, it can redirect the client to a status endpoint. The client can learn when the operation is finished by polling this endpoint.
- **Monitor** - This pattern implements a recurring process in a workflow, possibly looking for a change in state. For example, you could use this pattern to poll until specific conditions are met.
- **Human interaction** - This pattern combines automated processes that also involve some human interaction. Human interaction can be incorporated using timeouts and compensation logic that runs if the human fails to interact correctly within a specified response time. An approval process is an example of a process that involves human interaction.

Exercise - Create a workflow using Durable Functions

Timers in Durable Functions

Durable Functions provides timers for use in the orchestrator functions. They can implement delays or set up timeouts for asynchronous actions. Use durable timers in orchestrator functions instead of the **setTimeout()** and **setInterval()** functions.
You create a durable timer by calling the **createTimer** method of the DurableOrchestrationContext. This method returns a task that resumes on a specified date and time.

Using timers for delay

The following example illustrates how to use durable timers for delay. The example sends a reminder every day for 10 days.

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

module.exports = df.orchestrator(function*(context) {
   for (let i = 0; i < 10; i++) {
       const dayOfMonth = context.df.currentUtcDateTime.getDate();
       const deadline = moment.utc(context.df.currentUtcDateTime).add(1, 'd');
       yield context.df.createTimer(deadline.toDate());
       yield context.df.callActivity("SendReminder");
   }
});
```

Always use **currentUtcDateTime** to obtain the current date and time, instead of **Date.now** or **Date.UTC**.

Using timers for timeout

The following example illustrates how to use durable timers for timeout. Executing a different path if a timeout occurs. In this example, the function waits until either the GetQuote activity function completes or the deadline timer expires. If the activity function completes the code follows the success case, otherwise it follows the timeout case.

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

module.exports = df.orchestrator(function*(context) {
   const deadline = moment.utc(context.df.currentUtcDateTime).add(30, "s");

   const activityTask = context.df.callActivity("GetQuote");
   const timeoutTask = context.df.createTimer(deadline.toDate());

   const winner = yield context.df.Task.any([activityTask, timeoutTask]);
   if (winner === activityTask) {
       // success case
       timeoutTask.cancel();
       return true;
   }
   else
```

```
    {
        // timeout case
        return false;
    }
});
```

Exercise - Add a durable timer to manage a long-running task

6. Develop, test, and publish Azure Functions by using Azure Functions Core Tools

Exercise - Create a function locally by using the Core Tools

Exercise - Publish a function to Azure by using the Core Tools

7. Develop, test, and deploy an Azure Function with Visual Studio

Structure of an Azure Function

An Azure Function is implemented as a static class. The class provides a static, asynchronous method named Run, which acts as the entry point for the function.

The parameters passed to the Run method provide the context for the trigger. In the case of an HTTP trigger, the function receives an HttpRequest object. This object contains the header and body of the request. You can access the data in the request using the same techniques available in any HTTP application. The attributes applied to this attribute specify the authorization requirements (Anonymous in this case), and the HTTP operations to which the Azure function responds (GET and POST).

```
public static class Function1
{
    [FunctionName("Function1")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
HttpRequest req,
        ILogger log)
    {
        log.LogInformation("C# HTTP trigger function processed a request.");

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

        string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
        dynamic data = JsonConvert.DeserializeObject(requestBody);
        name = name ?? data?.name;

        return name != null
            ? (ActionResult)new OkObjectResult($"Hello, {name}")
            : new BadRequestObjectResult("Please pass a name on the query string
or in the request body");
```

```
    }
}
```

The function returns a value containing any output data and results, wrapped in an IActionResult object. The value is returned in the body of the HTTP response for the request.

Different types of trigger receive different input parameters and return types. The next example shows the code generated for a Blob trigger. In this example, the contents of the blob is made accessible through a Stream object, and the name of the blob is also provided. No data is returned by the trigger; its purpose is to read and process the data in the named blob:

```
public static class Function2
{
    [FunctionName("Function2")]
    public static void Run([BlobTrigger("samples-workitems/{name}", Connection =
"xxxxxxxxxxxxxxxxxxxxxxx")]Stream myBlob, string name, ILogger log)
    {
        log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name}
\n Size: {myBlob.Length} Bytes");
    }
}
```

In all cases, an Azure Function is passed an ILogger parameter. The function can use this parameter to write log messages, which the function app will write to storage for later analysis.

An Azure Function also contains metadata that specify the type of the trigger and any other specific information and security requirements. You can modify this metadata using the HttpTrigger, BlobTrigger, or other trigger attributes, as shown in the examples. The FunctionName attribute that precedes the function is an identifier for the function used by the function app. This name doesn't have to be the same as the name of the function, but it's good practice to keep them synchronized to avoid confusion.

Exercise - Create and test a simple Azure Function locally with Visual Studio

Publish a simple Azure Function

An Azure Function runs in the cloud in the context of an Azure Function App. A function app is a container that specifies the operating system for running an Azure Function, together with the resources available, such as the memory, computing power, and disk space. The Azure Function App also provides the public URL for running your functions. Behind the scenes, an Azure Function App is a collection of one or more virtual machines, running a web server. When you publish an Azure Function, you deploy it to these virtual machines.
There are several of options available for publishing an Azure Function. In this unit, you'll learn about some of these options.

- **Deploy from Visual Studio** - Azure Functions tools for Visual Studio enable you to deploy an Azure Function directly from Visual Studio. The Azure Functions template provides a Publish wizard. Using this wizard, you connect to your Azure account, and either specify an existing Azure Function App, or create a new one. The functions in your project are rebuilt and then deployed to the Azure Function App.
- **Continuous deployment** - Azure Functions makes it easy to deploy your function app using App Service continuous integration. Azure Functions integrates with BitBucket, Dropbox, GitHub, and Azure DevOps. This enables a workflow where function code updates made by using one of these integrated services triggers deployment to Azure.
- **Zip deployment** - Azure Function can be deployed from a zip file using the push deployment technique. You can do this with the Azure CLI, or by using the REST interface.

Exercise - Publish a simple Azure Function

Exercise - Unit test an Azure Function

8. Monitor GitHub events by using a webhook with Azure Functions

Webhooks are user-defined HTTP callbacks. They offer a lightweight mechanism for apps to be notified by another service when something of interest happens via an HTTP endpoint. You can use a webhook to trigger an Azure function, and then analyze the message, to determine what happened and how to respond.

Exercise - Create an Azure function triggered by a webhook

Exercise - Set up a webhook for a GitHub repository

Exercise - Trigger an Azure Function with a GitHub event

Exercise - Secure webhook payloads with a secret

9. Enable automatic updates in a web application using Azure Functions and SignalR Service

SignalR and persistent connections

In contrast to polling, a more favorable design features persistent connections between the client and server. Establishing a persistent connection allows the server to push data to the client at will. The on-demand nature of the connection reduces network traffic and load on the server. SignalR allows you to easily add this type of architecture to your application.

SignalR is an abstraction for a series of technologies that allows your app to enjoy two-way communication between the client and server. SignalR handles connection management automatically, and lets you broadcast messages to all connected clients simultaneously, like a chat room. You can also send messages to specific clients. The connection between the client and server is persistent, unlike a classic HTTP connection, which is re-established for each communication.

A key benefit of the abstraction provided by SignalR is the way it supports "transport" fallbacks. A transport is method of communicating between the client and server. SignalR connections begin with a standard HTTP request. As the server evaluates the connection, the most appropriate communication method (transport) is selected. Transports are chosen depending on the APIs available on the client.

For clients that support HTML 5, the WebSockets API transport is used by default. If the client doesn't support WebSockets, then SignalR falls back to Server Sent Events (also known as EventSource). For older clients, Ajax long polling or Forever Frame (IE only) is used to mimic a two-way connection.

The abstraction layer offered by SignalR provides two benefits to your application. The first advantage is future-proofing your app. As the web evolves and APIs superior to WebSockets become available, your application doesn't need to change. You could update to a version of SignalR that supports any new APIs and your application code won't need an overhaul.

The second benefit is that SignalR allows your application to gracefully degrade depending on supported technologies of the client. If it doesn't support WebSockets, then Server Sent Events are used. If the client can't handle Server Sent Events, then it uses Ajax long polling, and so on.

Exercise – Enable automatic updates in a web application using SignalR Service

Use a storage account to host a static website

Azure Storage includes a feature where you can place files in a specific storage container, which makes them available for HTTP requests. This feature, known as static website support makes hosting publicly available web pages a simple process.

When you copy files to a storage container named $web, those files are available to web browsers via a secure server using the https://<ACCOUNT_NAME>.<ZONE_NAME>.web.core.windows.net/<FILE_NAME> URI scheme.

Exercise - Use a storage account to host a static website

2. Connect your services together

1. Choose a messaging model in Azure to loosely connect your services

What is a Message?
In the terminology of distributed applications, messages have the following characteristics:

- A message contains raw data, produced by one component, that will be consumed by another component.
- A message contains the data itself, not just a reference to that data.
- The sending component expects the message content to be processed in a certain way by the destination component. The integrity of the overall system may depend on both sender and receiver doing a specific job.

What is an Event?
Events are lighter weight than messages, and are most often used for broadcast communications. The components sending the event are known as publishers, and receivers are known as subscribers.
With events, receiving components will generally decide in which communications they are interested, and will "subscribe" to those events. The subscription is managed by an intermediary, like Azure Event Grid or Azure Event Hubs. When publishers send an event, the intermediary will route that event to interested subscribers. This pattern is known as a "publish-subscribe architecture." It's not the only way to deal with events, but it is the most common.

Events have the following characteristics:

- An event is a lightweight notification that indicates that something happened.
- The event may be sent to multi**ple receivers, or to none at all.
- Events are often intended to "fan out," or have a large number of subscribers for each publisher.
- The publisher of the event has no expectation about the action a receiving component takes.
- Some events are discrete units and unrelated to other events.
- Some events are part of a related and ordered series.

How to choose messages or events?
Events are more likely to be used for broadcasts and are often ephemeral, meaning a communication might not be handled by any receiver if none is currently subscribing. **Messages** are more likely to be used where the distributed application requires a guarantee that the communication will be processed.

For each communication, consider the following question: **Does the sending component expect the communication to be processed in a particular way by the destination component?**
If the answer is yes, choose to use a message. If the answer is no, you may be able to use events.

Azure offers two solution to message-based approach, **Azure Queue Storage** & **Azure Service Bus**

Azure Queue Storage - Queue storage is a service that uses Azure Storage to store large numbers of messages that can be securely accessed from anywhere in the world using a simple REST-based interface. Queues can contain millions of messages, limited only by the capacity of the storage account that owns it.

Azure Service Bus Queues - Service Bus is a message broker system intended for enterprise applications. These apps often utilize multiple communication protocols, have different data contracts, higher security requirements, and can include both cloud and on-premises services. Service Bus is built on top of a dedicated messaging infrastructure designed for exactly these scenarios.

Azure Service Bus Topics - Azure Service Bus topics are like queues, but can have multiple subscribers. When a message is sent to a topic instead of a queue multiple components can be triggered to do their work. Internally, topics use queues. When you post to a topic, the message is copied and dropped into the queue for each subscription. The queue means that the message copy will stay around to be processed by **each subscription branch** even if the component processing that subscription is too busy to keep up.

Benefits of Queue
Queue infrastructures can support many advanced features that make them very useful in the following ways:

- **Increased reliability**
  Queues are used by distributed applications as a temporary storage location for messages pending delivery to a destination component. The source component can add a message to the queue and destination components can

retrieve the message at the front of the queue for processing. Queues increase the reliability of the message exchange because, at times of high demand, messages can simply wait until a destination component is ready to process them.

- **Message delivery guarantees**
  Queuing systems usually guarantee delivery of each message in the queue to a destination component. However, these guarantees can take different approaches:

  - At-Least-Once Delivery - In this approach, each message is guaranteed to be delivered to at least one of the components that retrieve messages from the queue.
    Note - In certain circumstances, it is possible that the same message may be delivered more than once. For example, if there are two instances of a web app retrieving messages from a queue, ordinarily each message goes to only one of those instances. However, if one instance takes a long time to process the message, and a time-out expires, the message may be sent to the other instance as well. Your web app code should be designed with this possibility in mind.

  - At-Most-Once Delivery - In this approach, each message is not guaranteed to be delivered, and there is a very small chance that it may not arrive. However, unlike At-Least-Once delivery, there is no chance that the message will be delivered twice.

  - First-In-First-Out (FIFO) - In most messaging systems, messages usually leave the queue in the same order in which they were added, but you should consider whether this order is guaranteed. If your distributed application requires that messages are processed in precisely the correct order, you must choose a queue system that includes a FIFO guarantee.

- **Transaction support**
  Some closely related groups of messages may cause problems when delivery fails for one message in the group.
  For example, consider an e-commerce application. When the user clicks the Buy button, a series of messages might be generated and sent off to various processing destinations:

  - A message with the order details is sent to a fulfillment center
  - A message with the total and payment details is sent to a credit card processor.

  - A message with the receipt information is sent to a database to generate an invoice for the customer

In this case, we want to make sure all messages get processed, or none of them are processed. We won't be in business long if the credit card message is not delivered, and all our orders are fulfilled without payment! You can avoid these kinds of problems by grouping the two messages into a transaction. Message transactions succeed or fail as a single unit - just like in the database world.

Choose Service Bus Topics if

- you need multiple receivers to handle each message

Choose Service Bus queues if

- You need an At-Most-Once delivery guarantee.
- You need a FIFO guarantee.
- You need to group messages into transactions.
- You want to receive messages without polling the queue.
- You need to provide a role-based access model to the queues.
- You need to handle messages larger than 64 KB but less than 256 KB.
- Your queue size will not grow larger than 80 GB.
- You would like to be able to publish and consume batches of messages.

Choose Queue storage if

- You need an audit trail of all messages that pass through the queue.
- You expect the queue to exceed 80 GB in size.
- You need to handle messages less than 64 KB.
- You want to track progress for processing a message inside of the queue.

What is Azure Event Grid?
Azure Event Grid is a fully-managed event routing service running on top of Azure Service Fabric. Event Grid distributes events from different sources, such as Azure Blob storage accounts or Azure Media Services, to different handlers, such as Azure

Functions or Webhooks. Event Grid was created to make it easier to build event-based and serverless applications on Azure.
Event Grid supports most Azure services as a publisher or subscriber and can be used with third-party services. It provides a dynamically scalable, low-cost, messaging system that allows publishers to notify subscribers about a status change. The following illustration shows Azure Event Grid receiving messages from multiple sources and distributing them to event handlers based on subscription.

There are several concepts in Azure Event Grid that connect a source to a subscriber:

- **Events**: What happened.
- **Event sources**: Where the event took place.
- **Topics**: The endpoint where publishers send events.
- **Event subscriptions**: The endpoint or built-in mechanism to route events, sometimes to multiple handlers. Subscriptions are also used by handlers to filter incoming events intelligently.
- **Event handlers**: The app or service reacting to the event.

The following illustration shows an Azure Event Grid positioned between multiple event sources and multiple event handlers. The event sources send events to the Event Grid and the Event Grid forwards relevant events to the subscribers. Event Grid use topics to decide which events to send to which handlers. Events sources tag each event with one or more topics, and event handlers subscribe to the topics they are interested in.

What is an event?
Events are the data messages passing through Event Grid that describe what has taken place. Each event is self-contained, can be up to 64 KB, and contains several pieces of information based on a schema defined by Event Grid:

```
[
  {
    "topic": string,
    "subject": string,
    "id": string,
    "eventType": string,
    "eventTime": string,
    "data":{
      object-unique-to-each-publisher
    },
    "dataVersion": string,
    "metadataVersion": string
  }
]
```

Field	Description
topic	The full resource path to the event source. Event Grid provides this value.
subject	Publisher-defined path to the event subject.
id	The unique identifier for event.
eventType	One of the registered event types for this event source. This is a value you can create filters against, e.g. CustomerCreated, BlobDeleted, HttpRequestReceived, etc.
eventTime	The time the event was generated based on the provider's UTC time.
data	Specific information that is relevant to the type of event. For example, an event about a new file being created in Azure Storage has details about the file, such as the lastTimeModified value. Or, an Event Hubs event has the URL of the Capture file. This field is optional.
dataVersion	The schema version of the data object. The publisher defines the schema version.
metadataVersion	The schema version of the event metadata. Event Grid defines the schema of the top-level properties. Event Grid provides this value.

What is an event source?
Event sources are responsible for sending events to Event Grid. Each event source is related to one or more event types. For example, Azure Storage is the event source for blob created events. IoT Hub is the event source for device created events. Your application is the event source for custom events that you define.

Types of event sources Events can be generated by the following Azure resource types:

- **Azure Subscriptions and Resource Groups**: Subscriptions and resource groups generate events related to management operations in Azure. For example, when a user creates a virtual machine, this source generates an event.

- **Container registry**: The Azure Container Registry service generates events when images in the registry are added, removed, or changed.
- **Event Hub**: Event Hub can be used to process and store events from a variety of data sources - typically logging or telemetry related. Event Hub can generate events to Event Grid when files are captured.
- **Service Bus**: Service bus can generate events to Event Grid when there are active messages with no active listeners.
- **Storage accounts**: Storage accounts can generate events when users add blobs, files, table entries, or queue messages. You can use both blob accounts and General-purpose V2 accounts as event sources.
- **Media Services**: Media Services hosts video and audio media and provides advanced management features for media files. Media Services can generate events when an encoding job is started or completed on a video file.
- **Azure IoT Hub**: IoT Hub communicates with and gathers telemetry from IoT devices. It can generate events whenever such communications arrive.
- **Custom events**: Custom events can be generated using the REST API, or with the Azure SDK on Java, GO, .NET, Node, Python, and Ruby. For example, you could create a custom event in the Web Apps feature of Azure App Service. This can happen in the worker role when it picks up a message from a storage queue.

What is an event topic?
Event topics categorize events into groups. Topics are represented by a public endpoint and are where the event source sends events to. When designing your application, you can decide how many topics to create. Larger solutions will create a custom topic for each category of related events, while smaller solutions might send all events to a single topic.

Topics are divided into system topics, and custom topics.

- **System topics** are built-in topics provided by Azure services. You don't see system topics in your Azure subscription because the publisher owns the topics, but you can subscribe to them. To subscribe, you provide information about the resource you want to receive events from. As long as you have access to the resource, you can subscribe to its events.
- Custom topics are application and third-party topics. When you create or are assigned access to a custom topic, you see that custom topic in your subscription.

What is an event subscription?
Event Subscriptions define which events on a topic an event handler wants to receive. A subscription can also filter events by their type or subject, so you can ensure an event handler only receives relevant events.

What is an event handler?
An event handler (sometimes referred to as an event "subscriber") is any component (application or resource) that can receive events from Event Grid. For example, Azure Functions can execute code in response to the new song being added to the Blob storage account. Subscribers can decide which events they want to handle and Event Grid will efficiently notify each interested subscriber when a new event is available - no polling required.

Types of event handlers The following object types in Azure can receive and handle events from Event Grid:

- **Azure Functions**: Custom code that runs in Azure, without the need for explicit configuration of a host virtual server or container. Use an Azure function as an event handler when you want to code a custom response to the event.
- **Webhooks**: A webhook is a web API that implements a push architecture.
- **Azure Logic Apps**: An Azure logic app hosts a business process as a workflow.
- **Microsoft Flow**: Flow also hosts workflows, but it is easier for non-technical staff to use.

Should you use Event Grid? Use Event Grid when you need these features:

- **Simplicity**: It is straightforward to connect sources to subscribers in Event Grid.
- **Advanced filtering**: Subscriptions have close control over the events they receive from a topic.
- **Fan-out**: You can subscribe to an unlimited number of endpoints to the same events and topics.
- **Reliability**: Event Grid retries event delivery for up to 24 hours for each subscription.
- **Pay-per-event**: Pay only for the number of events that you transmit.

Event Grid is a simple but versatile event distribution system. Use it to deliver discrete events to subscribers, which will receive those events reliably and quickly. We have one more messaging model to examine - what if we want to deliver a large stream of events? In this scenario, Event Grid isn't a great solution because it's designed for one-event-at-a-time delivery. Instead, we need to turn to another Azure service: Event Hubs.

2. Implement message-based communication workflows with Azure Service Bus

Creating a Queue in Service Bus

- **Message time to live** - Message time to live determines how long a message will stay in the queue before it expires and is removed or dead lettered. When sending messages it is possible to specify a different time to live for only that message. This default will be used for all messages in the queue which do not specify a time to live for themselves.
- **Locked duration** - Sets the amount of time that a message is locked for other receivers. After its lock expires, a message pulled by one receiver becomes available to be pulled by other receivers. Defaults to 30 seconds, with a maximum of 5 minutes.
- **Enable duplicate detection** - Enabling duplicate detection configures your queue to keep a history of all messages sent to the queue for a configurable amount of time. During that interval, your queue will not accept any duplicate messages. Enabling this property guarantees exactly-once delivery over a user-defined span of time.
- **Enable dead lettering on message expiration** - Dead lettering messages involves holding messages that cannot be successfully delivered to any receiver to a separate queue after they have expired. Messages do not expire in the dead letter queue, and it supports peek-lock delivery and all transactional operations.
- **Enable sessions** - Service bus sessions allow ordered handling of unbounded sequences of related messages. With sessions enabled a queue can guarantee first-in-first-out delivery of messages. **Enable partitioning** - Partitions a queue across multiple message brokers and message stores. Disconnects the overall throughput of a partitioned entity from any single message broker or messaging store. This property is not modifiable after a queue has been created.

Exercise - Implement a Service Bus topic and queue

Code with topics vs. code with queues
If you want every message sent to be delivered to all subscribing components, you'll use topics. Writing code that uses topics is a way to replace queues. You will use the same Microsoft.Azure.ServiceBus NuGet package, configure connection strings, and use asynchronous programming patterns.
However, you'll use the TopicClient class instead of the QueueClient class to send messages and the SubscriptionClient class to receive messages.

Exercise - Write code that uses Service Bus queues

Setting filters on subscriptions
If you want to control that specific messages sent to the topic are delivered to particular subscriptions, you can place filters on each subscription in the topic. Filters can be one of three types:

- **Boolean Filters** - The *TrueFilter* ensures that all messages sent to the topic are delivered to the current subscription. The *FalseFilter* ensures that none of the messages are delivered to the current subscription. (This effectively blocks or switches off the subscription.)
- **SQL Filters** A SQL filter specifies a condition by using the same syntax as a *WHERE* clause in a SQL query. Only messages that return *True* when evaluated against this subscription will be delivered to the subscribers.
- **Correlation Filters** A correlation filter holds a set of conditions that are matched against the properties of each message. If the property in the filter and the property on the message have the same value, it is considered a match.

Exercise - Write code that uses Service Bus topics

3. Communicate between applications with Azure Queue storage

Exercise - Create a storage account

Access authorization
Every request to a queue must be authorized and there are several options to choose from. |Authorization Type|Description| |Azure Active Directory|You can use role-based authentication and identify specific clients based on AAD credentials.| |Shared Key|Sometimes referred to as an account key, this is an encrypted key signature associated with the storage account. Every storage account has two of these keys that can be passed with each request to authenticate access. Using this approach is like using a root password - it provides full access to the storage account.| |Shared access signature|A shared access signature (SAS) is a generated URI that grants limited access to objects in your storage account to clients. You can restrict access to specific resources, permissions, and scope to a data range to automatically turn off access after a period of time.|

Exercise - Identify a queue

Programatically accessing a queue

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

The Azure Storage Client Library for .NET
The Azure Storage Client Library for .NET provides types to represent each of the objects you need to interact with:

- **CloudStorageAccount** represents your Azure storage account.
- **CloudQueueClient** represents Azure Queue storage.
- **CloudQueue** represents one of your queue instances.
- **CloudQueueMessage** represents a message.

Exercise - Add a message to the queue

Exercise - Retrieve a message from the queue

4. Enable reliable messaging for Big Data applications using Azure Event Hubs

Consumer groups
An Event Hub consumer group represents a specific view of an Event Hub data stream. By using separate consumer groups, multiple subscriber applications can process an event stream independently, and without affecting other applications. However, the use of many consumer groups isn't a requirement, and for many applications, the single default consumer group is sufficient.

Pricing
There are three pricing tiers for Azure Event Hubs: Basic, Standard, and Dedicated. The tiers differ in terms of supported connections, the number of available Consumer groups, and throughput. When using Azure CLI to create an Event Hubs namespace, if you don't specify a pricing tier, the default of Standard (20 Consumer groups, 1000 Brokered connections) is assigned.

Defining an Event Hubs namespace
An Event Hubs namespace is a containing entity for managing one or more Event Hubs. Creating an Event Hubs namespace typically involves the following configuration:

- Define namespace-level settings. Certain settings such as namespace capacity (configured using throughput units), pricing tier, and performance metrics are defined at the namespace level. These settings are applicable for all the Event Hubs within that namespace. If you don't define these settings, a default value is used: 1 for capacity and Standard for pricing tier.
  Keep the following aspects in mind:
  - You can't change the throughput unit once you set it.
  - You must balance your configuration against your Azure budget expectations.
  - You might consider configuring different Event Hubs for different throughput requirements. For example, if you have a sales data application and you're planning for two Event Hubs, it would make sense to use a separate namespace for each hub.
    You'll configure one namespace for high throughput collection of real-time sales data telemetry and one namespace for infrequent event log collection. This way, you only need to configure (and pay for) high throughput capacity on the telemetry hub.
- Select a unique name for the namespace. The namespace is accessible through this URL: namespace.servicebus.windows.net
- Defining the following optional properties:
  - Enable Kafka. This option enables Kafka applications to publish events to the Event Hub.
  - Make this namespace zone redundant. Zone-redundancy replicates data across separate data centers with their independent power, networking, and cooling infrastructures.
  - Enable Auto-Inflate and Auto-Inflate Maximum Throughput Units. Auto-Inflate provides an automatic scale-up option by increasing the number of throughput units up to a maximum value. This option is useful to avoid throttling in situations when incoming or outgoing data rates exceed the currently set number of throughput units.

Configuring a new Event Hub
After you create the Event Hubs namespace, you can create an Event Hub. When creating a new Event Hub, there are several mandatory parameters.
The following parameters are required to create an Event Hub:

- **Event Hub name** - Event Hub name that is unique within your subscription and:
  - Is between 1 and 50 characters long
  - Contains only letters, numbers, periods, hyphens, and underscores
  - Starts and ends with a letter or number

- **Partition Count** - The number of partitions required in an Event Hub (between 2 and 32). The partition count should be directly related to the expected number of concurrent consumers and can't be changed after the hub has been created. The partition separates the message stream so that consumer or receiver applications only need to read a specific subset of the data stream. If not defined, this value defaults to 4.
- **Message Retention** - The number of days (between 1 and 7) that messages will remain available if the data stream needs to be replayed for any reason. If not defined, this value defaults to 7.
  You can also optionally configure an Event Hub to stream data to an Azure Blob storage or Azure Data Lake Store account.

Exercise - Use the Azure CLI to Create an Event Hub

Exercise - Configure applications to send or receive messages through an Event Hub

Exercise - Evaluate the performance of the deployed Event Hub using the Azure portal

Applications that publish messages to Azure Event Hub very frequently will get the best performance using Advanced Message Queuing Protocol (AMQP) because it establishes a persistent socket.

3. Work with relational data in Azure

1. Provision an Azure SQL database to store application data

Why choose Azure SQL Database?

- **Convenience**
  Setting up SQL Server on a VM or on physical hardware requires you to know about hardware and software requirements. You'll need to understand the latest security best practices and manage operating system and SQL Server patches on a routine basis. You also need to manage backup and data retention issues yourself.
  With Azure SQL Database, we manage the hardware, software updates, and OS patches for you. All you specify is the name of your database and a few options. You'll have a running SQL database in minutes.

- **Cost**
  Because we manage things for you, there are no systems for you to buy,

provide power for, or otherwise maintain.
Azure SQL Database has several pricing options. These pricing options enable you to balance performance versus cost. You can start for just a few dollars a month.

- **Scale**
  With Azure SQL Database, you can adjust the performance and size of your database on the fly when your needs change.

- **Security** Azure SQL Database comes with a firewall that's automatically configured to restrict connections from the Internet.
  You can allow access to specific IP addresses that you trust. Doing so allows you to use Visual Studio, SQL Server Management Studio, or other tools to manage your Azure SQL database.

Azure SQL logical server
When you create your first Azure SQL database, you also create an Azure SQL logical server. Think of a logical server as an administrative container for your databases. You can control logins, firewall rules, and security policies through the logical server. You can also override these policies on each database within the logical server.

Azure SQL Database has two purchasing models:

- **DTU**
  DTU stands for Database Transaction Unit, and is a combined measure of compute, storage, and IO resources. Think of the DTU model as a simple, preconfigured purchase option.
  Because your logical server can hold more than one database, there's also the idea of eDTUs, or elastic Database Transaction Units. This option enables you to choose one price, but allow each database in the pool to consume fewer or greater resources depending on current load.

- **vCore**
  vCores are Virtual cores, which give you greater control over the compute and storage resources that you create and pay for.
  While the DTU model provides fixed combinations of compute, storage, and IO resources, the vCore model enables you to configure resources independently. For example, with the vCore model you can increase storage capacity but keep the existing amount of compute and IO throughput.

What are SQL elastic pools?
When you create your Azure SQL database, you can create a SQL elastic pool.
SQL elastic pools relate to eDTUs. They enable you to buy a set of compute and storage resources that are shared among all the databases in the pool. Each database

can use the resources they need, within the limits you set, depending on current load.

What is collation?
Collation refers to the rules that sort and compare data. Collation helps you define sorting rules when case sensitivity, accent marks, and other language characteristics are important.
Let's take a moment to consider what the default collation, SQL_Latin1_General_CP1_CI_AS, means.

- **Latin1_General** refers to the family of Western European languages.
- **CP1** refers to code page 1252, a popular character encoding of the Latin alphabet.
- **CI** means that comparisons are case insensitive. For example, "HELLO" compares equally to "hello".
- **AS** means that comparisons are accent sensitive. For example, "résumé" doesn't compare equally to "resume".

2. Create an Azure Database for PostgreSQL server

What is an Azure Database for PostgreSQL server?
The PostgreSQL server is a central administration point for one or more databases. The PostgreSQL service in Azure is a managed resource that provides performance guarantees, and provides access and features at the server level.
An Azure Database for PostgreSQL server is the parent resource for a database. A resource is a manageable item that's available through Azure. Creating this resource allows you to configure your server instance.

Pricing tiers
Azure Database for PostgreSQL provides you with the option to choose pricing tiers based on parameters like compute power and storage.

Exercise - Create an Azure Database for PostgreSQL server via the Azure CLI

Exercise - Connect to an Azure Database for PostgreSQL server

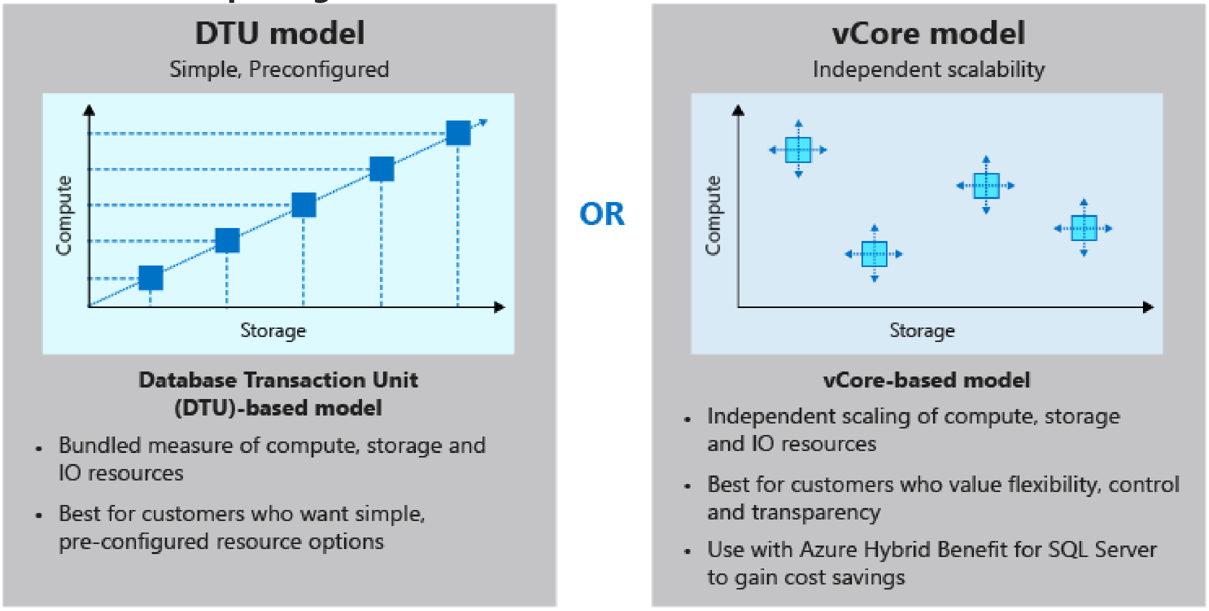
3. Scale multiple Azure SQL Databases with SQL elastic pools

How many databases to add to a pool?
The general guidance is, if the combined resources you would need for individual

databases to meet capacity spikes is more than 1.5 times the capacity required for the elastic pool, then the pool will be cost effective.
At a minimum, it is recommended to add at least two S3 databases or fifteen S0 databases to a single pool for it to have potential cost savings.
Depending on the performance tier, you can add up to 100 or 500 databases to a single pool.

Exercise - Create a SQL elastic pool

DTUs vs vCore pricing model



Database Transaction Unit (DTU)-based model

- Bundled measure of compute, storage and IO resources
- Best for customers who want simple, pre-configured resource options

vCore-based model

- Independent scaling of compute, storage and IO resources
- Best for customers who value flexibility, control and transparency
- Use with Azure Hybrid Benefit for SQL Server to gain cost savings

Exercise - Manage SQL elastic pools

4. Secure your Azure SQL Database

Firewall rules

Azure SQL Database has a built-in firewall that is used to allow and deny network access to both the database server itself, as well as individual databases. Initially, all public access to your Azure SQL Database is blocked by the SQL Database firewall. Firewall rules are configured at the server and/or database level, and will specifically state which network resources are allowed to establish a connection to the database. Depending on the level, the rules you can apply will be as follows:

- Server-level firewall rules
  - Allow access to Azure services - allows services within Azure to connect to your Azure SQL Database. When enabled, this setting allows communications from all Azure public IP addresses. This includes all Azure Platform as a Service (PaaS) services, such as Azure App Service

and Azure Container Service, as well as Azure VMs that have outbound Internet access. This rule can be configured through the ON/OFF option in the firewall pane in the portal, or by an IP rule that has 0.0.0.0 as the start and end IP addresses.
This rule is used when you have applications running on PaaS services in Azure, such as Azure Logic Apps or Azure Functions, that need to access your Azure SQL Database. Many of these services don't have a static IP address, so this rule is needed to ensure they are able to connect to the database.

- IP address rules - are rules that are based on specific public IP address ranges. IP addresses connecting from an allowed public IP range will be permitted to connect to the database.

- Virtual network rules - allow you to explicitly allow connection from specified subnets inside one or more Azure virtual networks (VNets). Virtual network rules can provide greater access control to your databases and can be a preferred option depending on your scenario. Since Azure VNet address spaces are private, you can effectively eliminate exposure to public IP addresses and secure connectivity to those addresses you control.

- Database-level firewall rules
    - IP address rules - These rules allow access to an individual database on a logical server and are stored in the database itself. For database-level rules, only IP address rules can be configured. They function the same as when applied at the server-level, but are scoped to the database only.
      The benefits of database-level rules are their portability. When replicating a database to another server, the database-level rules will be replicated, since they are stored in the database itself.
      The downside to database-level rules is that you can only use IP address rules. This may limit the flexibility you have and can increase administrative overhead.

Exercise - Restrict network access

TLS network encryption

Azure SQL Database enforces Transport Layer Security (TLS) encryption at all times for all connections, which ensures all data is encrypted "in transit" between the database and the client.

Transparent data encryption
Azure SQL Database protects your data at rest using transparent data encryption (TDE). TDE performs real-time encryption and decryption of the database, associated backups, and transaction log files at rest without requiring changes to the application. Using a database encryption key, transparent data encryption performs real-time I/O encryption and decryption of the data at the page level. Each page is decrypted when it's read into memory and then encrypted before being written to disk.
By default, TDE is enabled for all newly deployed Azure SQL databases. It's important to check that data encryption hasn’t been turned off, and older Azure SQL Server databases may not have TDE enabled.

Dynamic data masking
Dynamic data masking is a policy-based security feature that hides the sensitive data in the result set of a query over designated database fields, while the data in the database is not changed.
Data masking rules consist of the column to apply the mask to, and how the data should be masked. You can create your own masking format, or use one of the standard masks such as:
Default value, which displays the default value for that data type instead.

- Credit card value, which only shows the last four digits of the number, converting all other numbers to lower case x’s.
- Email, which hides the domain name and all but the first character of the email account name.
- Number, which specifies a random number between a range of values. For example, on the credit card expiry month and year, you could select random months from 1 to 12 and set the year range from 2018 to 3000.
- Custom string, which allows you to set the number of characters exposed from the start of the data, the number of characters exposed from the end of the data, and the characters to repeat for the remainder of the data.

When querying the columns, database administrators will still see the original values, but non-administrators will see the masked values. You can allow other users to see the non-masked versions by adding them to the SQL users excluded from masking list.

Exercise - Secure your data in transit, at rest, and on display

Azure SQL Database auditing
By enabling auditing, operations that occur on the database are stored for later inspection or to have automated tools analyze them. Auditing is also used for compliance management or understanding how your database is used. Auditing is

also required if you wish to use Azure threat detection on your Azure SQL database. You can use SQL database auditing to:

- Retain an audit trail of selected events. You can define categories of database actions to be audited.
- Report on database activity. You can use pre-configured reports and a dashboard to get started quickly with activity and event reporting.
- Analyze reports. You can find suspicious events, unusual activity, and trends.
- Audit logs are written to Append Blobs in an Azure Blob storage account that you designate. Audit policies can be applied at the server-level or database-level. Once enabled, you can use the Azure portal to view the logs, or send them to Log Analytics or Event Hub for further processing and analysis.

Auditing in practice
As a best practice, avoid enabling both server blob auditing and database blob auditing together, unless:

- You want to use a different storage account or retention period for a specific database.
- You want to audit event types or categories for a specific database that differs from the rest of the databases on the server. For example, you might have table inserts that need to be audited but only for a specific database.

Otherwise, it's recommended you enable only server-level blob auditing and leave the database-level auditing disabled for all databases.

Advanced Data Security for Azure SQL Database
Advanced Data Security (ADS) provides a set of advanced SQL security capabilities, including data discovery & classification, vulnerability assessment, and Advanced Threat Protection.

- **Data discovery & classification** (currently in preview) provides capabilities built into Azure SQL Database for discovering, classifying, labeling & protecting the sensitive data in your databases. It can be used to provide visibility into your database classification state, and to track the access to sensitive data within the database and beyond its borders.

- **Vulnerability assessment** is an easy to configure service that can discover, track, and help you remediate potential database vulnerabilities. It provides visibility into your security state, and includes actionable steps to resolve security issues, and enhance your database fortifications.

- **Advanced Threat Protection** detects anomalous activities indicating unusual and potentially harmful attempts to access or exploit your database. It continuously monitors your database for suspicious activities, and provides immediate security alerts on potential vulnerabilities, SQL injection attacks, and anomalous database access patterns. Advanced Threat Protection alerts provide details of the suspicious activity and recommend action on how to investigate and mitigate the threat.

Exercise - Monitor your database

5. Develop and configure an ASP.NET application that queries an Azure SQL database

Exercise - Create tables, bulk import, and query data

Exercise - Connect an ASP.NET application to Azure SQL Database

4. Store data in Azure

1. Choose a data storage approach in Azure

Structured data
Structured data, sometimes referred to as relational data, is data that adheres to a strict schema, so all of the data has the same fields or properties. The shared schema allows this type of data to be easily searched with query languages such as SQL (Structured Query Language). This capability makes this data style perfect for applications such as CRM systems, reservations, and inventory management. Structured data is straightforward in that it's easy to enter, query, and analyze. All of the data follows the same format. However, forcing a consistent structure also means evolution of the data is more difficult as each record has to be updated to conform to the new structure.

Semi-structured data
Semi-structured data is less organized than structured data, and is not stored in a relational format, as the fields do not neatly fit into tables, rows, and columns. Semi-structured data contains tags that make the organization and hierarchy of the data apparent - for example, key/value pairs. Semi-structured data is also referred to as non-relational or NoSQL data. The expression and structure of the data in this style is defined by a serialization language.

- **XML**, or extensible markup language, was one of the first data languages to receive widespread support. It's text-based, which makes it easily human and machine-readable. In addition, parsers for it can be found for almost all popular development platforms. XML allows you to express relationships and has standards for schema, transformation, and even displaying on the web.
- **JSON** – or JavaScript Object Notation, has a lightweight specification and relies on curly braces to indicate data structure. Compared to XML, it is less verbose and easier to read by humans. JSON is frequently used by web services to return data.
- **YAML** – or YAML Ain't Markup Language, is a relatively new data language that's growing quickly in popularity in part due to its human-friendliness. The data structure is defined by line separation and indentation, and reduces the dependency on structural characters like parentheses, commas and brackets.

Unstructured data

The organization of unstructured data is ambiguous. Unstructured data is often delivered in files, such as photos or videos. The video file itself may have an overall structure and come with semi-structured metadata, but the data that comprises the video itself is unstructured. Therefore, photos, videos, and other similar files are classified as unstructured data.

What is a transaction?

A transaction is a logical group of database operations that execute together. Transactions are often defined by a set of four requirements, referred to as ACID guarantees. ACID stands for Atomicity, Consistency, Isolation, and Durability:

- **Atomicity** means a transaction must execute exactly once and must be atomic; either all of the work is done, or none of it is. Operations within a transaction usually share a common intent and are interdependent.
- **Consistency** ensures that the data is consistent both before and after the transaction.
- **Isolation** ensures that one transaction is not impacted by another transaction.
- **Durability** means that the changes made due to the transaction are permanently saved in the system. Committed data is saved by the system so that even in the event of a failure and system restart, the data is available in its correct state.

OLTP vs OLAP

Transactional databases are often called OLTP (Online Transaction Processing) systems. OLTP systems commonly support lots of users, have quick response times, and handle large volumes of data. They are also highly available (meaning they have very minimal downtime), and typically handle small or relatively simple transactions. On the contrary, OLAP (Online Analytical Processing) systems commonly support fewer users, have longer response times, can be less available, and typically handle large and complex transactions.

Good to read - Choose a storage solution on Azure

2. Create an Azure Storage account

What is a storage account?
A storage account is a container that groups a set of Azure Storage services together. Only data services from Azure Storage can be included in a storage account (Azure Blobs, Azure Files, Azure Queues, and Azure Tables).

Storage account settings

- **Name**: Each storage account has a name. The name must be globally unique within Azure, use only lowercase letters and digits and be between 3 and 24 characters.
- **Subscription**: The Azure subscription that will be billed for the services in the account.
- **Location**: The datacenter that will store the services in the account.
- **Performance**: Determines the data services you can have in your storage account and the type of hardware disks used to store the data. Standard allows you to have any data service (Blob, File, Queue, Table) and uses magnetic disk drives. Premium allows you to create premium page blob in all regions, and block blob accounts in supported regions. These storage accounts use solid-state drives (SSD) for storage.
- **Replication**: Determines the strategy used to make copies of your data to protect against hardware failure or natural disaster. At a minimum, Azure will automatically maintain a copy of your data within the data center associated with the storage account. This is called locally-redundant storage (LRS), and guards against hardware failure but does not protect you from an event that incapacitates the entire datacenter. You can upgrade to one of the other options such as geo-redundant storage (GRS) to get replication at different datacenters across the world.
- **Access tier**: Controls how quickly you will be able to access the blobs in this storage account. Hot gives quicker access than Cool, but at increased cost. This applies only to blobs, and serves as the default value for new blobs.
- **Secure transfer required**: A security feature that determines the supported protocols for access. Enabled requires HTTPs, while disabled allows HTTP.
- **Virtual networks**: A security feature that allows inbound access requests only from the virtual network(s) you specify.

The number of storage accounts you need is typically determined by your data diversity, cost sensitivity, and tolerance for management overhead.

- **Data diversity** - Organizations often generate data that differs in where it is consumed, how sensitive it is, which group pays the bills, etc. Diversity along any of these vectors can lead to multiple storage accounts. Let's consider two examples:
    - Do you have data that is specific to a country or region? If so, you might want to locate it in a data center in that country for performance or compliance reasons. You will need one storage account for each location.
    - Do you have some data that is proprietary and some for public consumption? If so, you could enable virtual networks for the proprietary data and not for the public data. This will also require separate storage accounts. In general, increased diversity means an increased number of storage accounts.
- **Cost sensitivity** - A storage account by itself has no financial cost; however, the settings you choose for the account do influence the cost of services in the account. Geo-redundant storage costs more than locally-redundant storage. Premium performance and the Hot access tier increase the cost of blobs.
  You can use multiple storage accounts to reduce costs. For example, you could partition your data into critical and non-critical categories. You could place your critical data into a storage account with geo-redundant storage and put your non-critical data in a different storage account with locally-redundant storage.
- **Tolerance for management overhead** - Each storage account requires some time and attention from an administrator to create and maintain. It also increases complexity for anyone who adds data to your cloud storage; everyone in this role needs to understand the purpose of each storage account so they add new data to the correct account.
  Storage accounts are a powerful tool to help you get the performance and security you need while minimizing costs. A typical strategy is to start with an analysis of your data and create partitions that share characteristics like location, billing, and replication strategy, and then create one storage account for each partition.

Advanced settings

- **Secure transfer required** setting controls whether HTTP can be used for the REST APIs used to access data in the Storage account. Setting this option to

Enabled will force all clients to use SSL (HTTPS). Most of the time you'll want to set this to Enabled as using HTTPS over the network is considered a best practice.
If Secured transfer(HTTPS) is enabled, it will enforce some additional restrictions. Azure files service connections without encryption will fail, including scenarios using SMB 2.1 or 3.0 on Linux. Because Azure storage doesn't support SSL for custom domain names, this option cannot be used with a custom domain name.

- **Large file shares** provides support up to a 100TiB, however this type of storage account can't convert to a Geo-redundant storage offering and upgrades are permanent.
- **Blob Soft delete** lets you recover your blob data in many cases where blobs or blob snapshots are deleted accidentally or overwritten.
- **Data Lake Storage Gen2** option is for big-data applications.

Exercise - Create a storage account using the Azure portal

3. Connect an app to Azure Storage

A single Azure subscription can host up to 200 storage accounts, each of which can hold 500 TB of data. If you have a business case, you can talk to the Azure Storage team and get approval for up to 250 storage accounts in a subscription, which pushes your max storage up to 125 Petabytes!

Blob storage

Azure Blob storage is an object storage solution optimized for storing massive amounts of unstructured data, such as text or binary data. Blob storage is ideal for:

- Serving images or documents directly to a browser, including full static websites.
- Storing files for distributed access.
- Streaming video and audio.
- Storing data for backup and restoration, disaster recovery, and archiving.
- Storing data for analysis by an on-premises or Azure-hosted service.

Azure Storage supports three kinds of blobs:

Blob type	Description
Block blobs	Block blobs are used to hold text or binary files up to ~5 TB (50,000 blocks of 100 MB) in size. The primary use case for block blobs is the storage of files that are read from beginning to end, such as media files or image files for websites. They are named block blobs because files larger than 100 MB must be uploaded as small blocks, which are then consolidated (or committed) into the final blob.
Page blobs	Page blobs are used to hold random-access files up to 8 TB in size. Page blobs are used primarily as the backing storage for the VHDs used to provide durable disks for Azure Virtual Machines (Azure VMs). They are named page blobs because they provide random read/write access to 512-byte pages.
Append blobs	Append blobs are made up of blocks like block blobs, but they are optimized for append operations. These are frequently used for logging information from one or more sources into the same blob. For example, you might write all of your trace logging to the same append blob for an application running on multiple VMs. A single append blob can be up to 195 GB.

Files Azure Files enables you to set up highly available network file shares that can be accessed by using the standard Server Message Block (SMB) protocol. This means that multiple VMs can share the same files with both read and write access. You can also read the files using the REST interface or the storage client libraries. You can also associate a unique URL to any file to allow fine-grained access to a private file for a set period of time. File shares can be used for many common scenarios:

- Storing shared configuration files for VMs, tools, or utilities so that everyone is using the same version.
- Log files such as diagnostics, metrics, and crash dumps.
- Shared data between on-premises applications and Azure VMs to allow migration of apps to the cloud over a period of time.

Exercise - Create a new app to work with Azure storage

Exercise - Create an Azure storage account

Exercise - Add the storage client library to your app

To work with data in a storage account, your app will need two pieces of data:

- An access key - Each storage account has two unique access keys that are used to secure the storage account.
- The REST API endpoint - The REST endpoint is a combination of your storage account name, the data type, and a known domain. For example:
    - Blobs - https://[name].blob.core.windows.net/
    - Queues - https://[name].queue.core.windows.net/
    - Tables - https://[name].table.core.windows.net/
    - Files - https://[name].file.core.windows.net/

Connection strings - The simplest way to handle access keys and endpoint URLs within applications is to use storage account connection strings. A connection string provides all needed connectivity information in a single text string.
DefaultEndpointsProtocol=https;AccountName={your-storage}; AccountKey={your-access-key}; EndpointSuffix=core.windows.net

It's highly recommended that you periodically rotate your access keys to ensure they remain private, just like changing your passwords. If you are using the key in a server application, you can use an Azure Key Vault to store the access key for you. Key Vaults include support to synchronize directly to the Storage Account and automatically rotate the keys periodically. Using a Key Vault provides an additional layer of security, so your app never has to work directly with an access key.

Shared access signatures (SAS)
Access keys are the easiest approach to authenticating access to a storage account. However they provide full access to anything in the storage account, similar to a root password on a computer.
Storage accounts offer a separate authentication mechanism called shared access signatures that support expiration and limited permissions for scenarios where you need to grant limited access. You should use this approach when you are allowing other users to read and write data to your storage account.

Exercise - Add Azure Storage configuration to your app

The simplest way to initialize the object model is to use CloudStorageAccount.Parse or CloudStorageAccount.TryParse to parse the connection string. These methods only guarantees that the connection is well-formatted; they don't verify that the account exists or the access-key is valid. The resulting CloudStorageAccount instance returned from the Parse() or TryParse() method call exposes methods to create a client objects to access the Azure Blob, Files, Queue and Table storage services.

Exercise - Connect with your Azure Storage configuration

4. Secure your Azure Storage account

Encryption at rest

All data written to Azure Storage is automatically encrypted by Storage Service Encryption (SSE) with a 256-bit Advanced Encryption Standard (AES) cipher. This incurs no additional charges and doesn't degrade performance. It can't be disabled. For virtual machines (VMs), Azure lets you encrypt virtual hard disks (VHDs) by using Azure Disk Encryption. This encryption uses BitLocker for Windows images, and it uses dm-crypt for Linux.
Azure Key Vault stores the keys automatically to help you control and manage the disk-encryption keys and secrets. So even if someone gets access to the VHD image and downloads it, they can't access the data on the VHD.

Encryption in transit

Keep your data secure by enabling transport-level security between Azure and the client. Always use HTTPS to secure communication over the public internet. When you call the REST APIs to access objects in storage accounts, you can enforce the use of HTTPS by requiring secure transfer for the storage account. After you enable secure transfer, connections that use HTTP will be refused. This flag will also enforce secure transfer over SMB by requiring SMB 3.0 for all file share mounts.

CORS support

Azure Storage supports cross-domain access through cross-origin resource sharing (CORS). CORS uses HTTP headers so that a web application at one domain can access resources from a server at a different domain. By using CORS, web apps ensure that they load only authorized content from authorized sources. CORS support is an optional flag you can enable on Storage accounts. The flag adds the appropriate headers when you use HTTP GET requests to retrieve resources from the Storage account.

Role-based access control

To access data in a storage account, the client makes a request over HTTP or HTTPS. Every request to a secure resource must be authorized. The service ensures that the client has the permissions required to access the data. You can choose from several access options. Arguably, the most flexible option is role-based access.
Azure Storage supports Azure Active Directory and role-based access control (RBAC) for both resource management and data operations. To security principals, you can assign RBAC roles that are scoped to the storage account. Use Active Directory to authorize resource management operations, such as configuration. Active Directory is supported for data operations on Blob and Queue storage.
To a security principal or a managed identity for Azure resources, you can assign

RBAC roles that are scoped to a subscription, a resource group, a storage account, or an individual container or queue.

Auditing access
Auditing is another part of controlling access. You can audit Azure Storage access by using the built-in Storage Analytics service.
Storage Analytics logs every operation in real time, and you can search the Storage Analytics logs for specific requests. Filter based on the authentication mechanism, the success of the operation, or the resource that was accessed.

Azure Storage accounts can create authorized apps in Active Directory to control access to the data in blobs and queues. This authentication approach is the best solution for apps that use Blob storage or Queue storage. For other storage models, clients can use a shared key, or shared secret. This authentication option is one of the easiest to use, and it supports blobs, files, queues, and tables.

Shared Access Signatures(SAS)
A shared access signature is a string that contains a security token that can be attached to a URI. Use a shared access signature to delegate access to storage objects and specify constraints, such as the permissions and the time range of access.
Types of shared access signatures

- You can use a **service-level** shared access signature to allow access to specific resources in a storage account. You'd use this type of shared access signature, for example, to allow an app to retrieve a list of files in a file system or to download a file.
- Use an **account-level** shared access signature to allow access to anything that a service-level shared access signature can allow, plus additional resources and abilities. For example, you can use an account-level shared access signature to allow the ability to create file systems.

Network access control
By default, storage accounts accept connections from clients on any network. To limit access to selected networks, you must first change the default action. You can restrict access to specific IP addresses, ranges, or virtual networks.

Advanced Threat Protection
Detecting threats to your data is an important part of security. You can check an audit trail for all activity against a storage account. But that will often only show you that an intrusion has already occurred. What you really want is a way to be notified when suspicious activity is happening. That's where the Advanced Threat Protection feature in Azure Storage can help.
Advanced Threat Protection, now in public preview, detects anomalies in account activity. It then notifies you of potentially harmful attempts to access your account.

You don't have to be a security expert or manage security monitoring systems to take advantage of this layer of threat protection.
Currently, Advanced Threat Protection for Azure Storage is available for the Blob service. Security alerts are integrated with Azure Security Center. The alerts are sent by email to subscription admins.

Azure Data Lke Storage security features
Azure Data Lake Storage Gen2 provides a first-class data lake solution that allows enterprises to pull together their data. It's built on Azure Blob storage, so it inherits all of the security features we've reviewed in this module.
Along with role-based access control (RBAC), Azure Data Lake Storage Gen2 provides access control lists (ACLs) that are POSIX-compliant and that restrict access to only authorized users, groups, or service principals. It applies restrictions in a way that's flexible, fine-grained, and manageable. Azure Data Lake Storage Gen2 authenticates through Azure Active Directory OAuth 2.0 bearer tokens. This allows for flexible authentication schemes, including federation with Azure AD Connect and multifactor authentication that provides stronger protection than just passwords.
More significantly, these authentication schemes are integrated into the main analytics services that use the data. These services include Azure Databricks, HDInsight, and SQL Data Warehouse. Management tools such as Azure Storage Explorer are also included. After authentication finishes, permissions are applied at the finest granularity to ensure the right level of authorization for an enterprise's big-data assets.
The Azure Storage end-to-end encryption of data and transport layer protections complete the security shield for an enterprise data lake. The same set of analytics engines and tools can take advantage of these additional layers of protection, resulting in complete protection of your analytics pipelines.

5. Store application data with Azure Blob storage

Azure Blob storage is unstructured, meaning that there are no restrictions on the kinds of data it can hold. Blobs aren't limited to common file formats — a blob could contain gigabytes of binary data streamed from a scientific instrument, an encrypted message for another application, or data in a custom format for an app you're developing.
Blobs are usually not appropriate for structured data that needs to be queried frequently. They have higher latency than memory and local disk and don't have the indexing features that make databases efficient at running queries. However, blobs are frequently used in combination with databases to store non-queryable data. For example, an app with a database of user profiles could store profile pictures in blobs.
Blobs are used for data storage in many ways across all kinds of applications and architectures:

- Apps that need to transmit large amounts of data using messaging system that supports only small messages. These apps can store data in blobs and send the blob URLs in messages.
- Blob storage can be used like a file system for storing and sharing documents and other personal data.
- Static web assets like images can be stored in blobs and made available for public download as if they were files on a web server.
- Many Azure components use blobs behind the scenes. For example, Azure Cloud Shell stores your files and configuration in blobs, and Azure Virtual Machines uses blobs for hard-disk storage.

In Blob storage, every blob lives inside a blob container. You can store an unlimited number of blobs in a container and an unlimited number of containers in a storage account. Containers are "flat" — they can only store blobs, not other containers. Blobs and containers support metadata in the form of name-value string pairs. Your apps can use metadata for anything you like: a human-readable description of a blob's contents to be displayed by the application, a string that your app uses to determine how to process the blob's data, etc.

Apps using blobs as part of a storage scheme that includes a database often don't need to rely heavily on organization, naming, or metadata to indicate anything about their data. Such apps commonly use identifiers like GUIDs as blob names and reference these identifiers in database records. The app will use the database to determine where blobs are stored and the kind of data they contain.
Other apps may use Azure Blob storage more like a personal file system, where container and blob names are used to indicate meaning and structure. Blob names in these kinds of apps will often look like traditional file names and include file name extensions like .jpg to indicate what kind of data they contain. They'll use virtual directories (see below) to organize blobs and will frequently use metadata tags to store information about blobs and containers.

Blob name prefixes (virtual directories)
Technically, containers are "flat" and do not support any kind of nesting or hierarchy. But if you give your blobs hierarchical names that look like file paths (such as finance/budgets/2017/q1.xls), the API's listing operation can filter results to specific prefixes. This allows you to navigate the list as if it was a hierarchical system of files and folders.
This feature is often called virtual directories because some tools and client libraries use it to visualize and navigate Blob storage as if it was a file system. Each folder navigation triggers a separate call to list the blobs in that folder.

Public access and containers as security boundaries
By default, all blobs require authentication to access. However, individual containers

can be configured to allow public downloading of their blobs without authentication. This feature supports many use cases, such as hosting static website assets and sharing files. This is because downloading blob contents works the same way as reading any other kind of data over the web: you just point a browser or anything that can make a GET request at the blob URL.
Enabling public access is important for scalability because data downloaded directly from Blob storage doesn't generate any traffic in your server-side app. Even if you don't immediately take advantage of public access or if you will use a database to control data access via your application, plan on using separate containers for data you want to be publicly available.
In addition to public access, Azure has a shared access signature feature that allows fine-grained permissions control on containers. Precision access control enables scenarios that further improve scalability, so thinking about containers as security boundaries in general is a helpful guideline.

Exercise - Create Azure storage resources

Exercise - Configure and initialize the client library

Exercise - Get blob references

Exercise - Blob uploads and downloads