daniel:User1234

# Plan

4 modules every week

<https://skillcertpro.com/product/developing-solutions-for-microsoft-azure-az-204-practice-exam-test/> (not recommended)

Learn downloaded git repo <https://github.com/arvigeus/AZ-204>

Take test (the answers in git repo as well) <https://az-204.vercel.app/?id=585a8c1fb45e87c4d5f9c9134633e8239f53275f34a6df54c3c7db350c4d1100>

Good practice test (if not enough) <https://www.whizlabs.com/microsoft-azure-certification-az-204/>

Test questions are downloaded in pdf (2025), more tests if not enough: <https://www.dumps-files.com/files/AZ-204/>

# Exam

Free practical assessment (test exam) <https://learn.microsoft.com/en-ca/credentials/certifications/exams/az-204/practice/assessment?assessment-type=practice&assessmentId=35>

Channel with practice questions <https://www.youtube.com/watch?v=si37TjkmWBA>   
and good video <https://www.youtube.com/watch?v=k4Gm7HnIpcg>

# Azure SQL Server

Server name: dy-azure-sql-server

Db name: schoolRegister

Local db can be saved as .bacpac file (MS SQL Server Management Studio -> right click on db -> Tasks -> Export Data Tier application), but to restore from .bacpac file you must:

* Load .bacpac to Azure Storage – Blob Container,
* Create new Azure SQL database ,
* During creation configuration select import from and select file in Azure Storage.

To create db from code-first app:

* Create empty db,
* Paste connection string to your app,
* Run app locally on computer
* Execute EF CLI command like  
  dotnet ef database update --project SchoolRegister.DAL --startup-project SchoolRegister.Web

It will build program and create all tables and relations in db.

# Azure Storage

Account name: dystorage

# Azure App Service (AAS)

## Explore Azure App Service

Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends.

scale up/down – scale number of cores/RAM of the underlying machine.

scale out/in – scale the number of machines

 Azure App Service works with orchestration platforms: Windows containers and Docker Compose (.yml file). For Kubernetes use **Azure Kubernetes Service** **(AKS)** instead of AAS.

Containers (Windows or Linux) are pulled from Azure Container Registry or Docker Hub.

Summary:

* Windows apps are run without container
* Linux based apps are run within single container – Azure creates it under the hood
* To configure custom container(s) you can use orchestration and any registry (Docker Hub etc.)

For Linux containers **custom container** reads app files faster than **built-in image** because in built-in image the app files are in separate from container place – in Azure Storage.

**App Service Environment** **(ASE)**

* A premium, isolated version of App Service.
* Runs inside your own virtual network (VNet).
* You get your own VM
* Isolated and IsolatedV2 plans

Ideal for high-security, high-scale, or compliance-restricted scenarios.

**App Service plan** - scale unit of the App Service apps. Each App Service plan defines:

* Operating System (Windows, Linux)
* Region (West US, East US, etc.)
* Number of VM instances
* Size of VM instances (Small, Medium, Large)
* Pricing tier (Free, Shared, Basic, Standard, Premium, PremiumV2, PremiumV3, Isolated, IsolatedV2)

Free and Shared tier – can’t scale out, your apps are on the same VM (and other customers apps)

Basic, Standard, Premium, PremiumV2, and PremiumV3 ­– you can scale out, but still VM shared with other customers. But your code is isolated from others code – Compute isolation.

Isolated and IsolatedV2 tiers run dedicated Azure VMs on dedicated Azure Virtual Networks. Network isolation on top of compute isolation. Maximum scale-out capabilities.

Scaling out the plan:

* An app runs on all the VM instances configured in the App Service plan.
* If multiple apps are in the same App Service plan, they all share the same VM instances.
* If you have multiple deployment slots for an app, all deployment slots also run on the same VM instances.

If the plan is configured to run five VM instances, then all apps in the plan run on all five instances – every app will be run on every of 5 VM, but requests from users to app will be divided between VM equally (1 request handles 1 VM). But if you have a singleton service – the singleton state will be encapsulated within VM and other VM won’t track state changes. Multiple apps can be deployed within one plan.

**Deployment**

Automated deployment: Azure DevOps, GitHub, BitBucket  
Manual deployment:

* GIT – generate Git URL in Azure and paste it to local git repo as **remote** and git push azure master – Azure automatically builds and deploys the app
* CLI (az webapp up --name myapp --resource-group myrg) – the fastest way to deploy in Azure.
* ZIP – pack to ZIP and sent via POST to Azure, example:  
  curl -X POST -u <username>:<password> \

--data-binary @app.zip \

https://<your-app-name>.scm.azurewebsites.net/api/zipdeploy

* FTP/S: Azure gives you FTP credentials and an FTP hostname. You use an FTP client (like FileZilla or VS Code extensions) to connect. You drag and drop your files into the web root folder.

Slots allow firstly to deploy a new production build to a staging environment, warm up the necessary worker instance to match production scale and then swap staging and production slot.

Sidecar container – separate container related to main app container. Available up to nine sidecar containers for each sidecar-enabled custom container app. Add in **Deployment Center** in the app's management page.

**AAP Authentication and Authorization**

AAP provides own solution for authentication and authorization. User accounts managed through **Microsoft Entra ID**. It easily works with third party authorization (through MS/Google/Meta/X accounts etc.). ASP roles (attributes) can be connected to AAP authorization – the APP can send configurated roles in token. But the passwords or passwords hashes are hidden and there is no access to them. Access another user account can be through code changes (turn off authorization etc.).

Two ways of authorization:

* **Server-directed flow** – without provider SDK, typically for web projects. The authorization takes place on provider’s login page.
* **Client-directed flow** – with provider SDK, typically for browser-less apps. The application signs users in to the provider manually (using SDK code) and then submits the authentication token to App Service for validation. This applies to REST APIs, Azure Functions and native mobile apps.

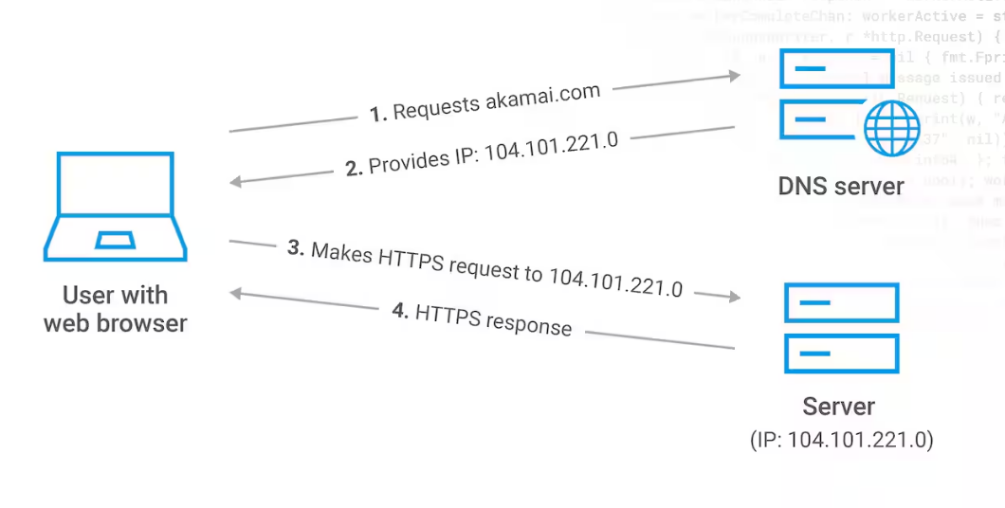
Authentication can be configured to allow or not allow the unauthenticated requests. For allowed unauthenticated requests the provider will pass all requests to application, but only authenticated requests will have authentication info in the HTTP headers. For disallowed unauthenticated requests – disallowed unauthenticated requests will be redirected to another chosen auth provider /.auth/login/<provider> or simply rejected by HTTP 401 Unauthorized response. **This restriction applies to all calls to application, to any page.**

AAP writes all logs about authentication and authorization with details, it available to browse.

**App Service networking features**

Most plans deploy apps in shared VMs, so different apps may have same network. That is why connecting and configuring the network are available only in App Service Environment (Iso IsoV2). But in other plans the network can be configured through **networking features**. The features dedicated to configure calls TO app can’t be used to configure calls FROM app, same for opposite.

**Worker** – a VM managed by Azure.

**DNS –** domain name system. It converts entered e. g. in browser domain like <https://yourapp.azurewebsites.net> to real ip address. The requests go to real ip address.  
**Inbound IP** – the IP that users can use to reach the app. AAP by default don't have a guaranteed dedicated inbound IP — it's shared. You get a dedicated IP with App Service Environment. When client enter the azure app URL, the Azure DNS resolves it to IP address.

1. User accesses your app via a domain like:  
   <https://yourapp.azurewebsites.net>
2. The DNS lookup resolves this to a shared inbound IP, like:  
   20.42.72.18
3. The request hits Azure’s **App Service Front End**, which is a **load balancer** that:  
   - Accepts traffic on that IP  
   - Inspects the **Host header** in the HTTP request:  
   Host: yourapp.azurewebsites.net
4. Azure knows which app matches yourapp.azurewebsites.net and routes the request to your specific app instance behind the scenes.

**Outbound IP** – the IP from which app send requests to another resources (API, servers, apps). If the app makes a request like:

HttpClient.Get("https://api.something.com/data");

The request will appear to come from one of the outbound IP addresses assigned to your App Service.

Outbound IPs are shared by all apps running on the same worker (or workers running within same unit scale – plan).

#the outbound IP addresses currently used by app (equal to all used outbound IP used by plan)

az webapp show \

    --resource-group <group\_name> \

    --name <app\_name> \

    --query outboundIpAddresses \

    --output tsv #tsv fomrat - tab separated values

A **resource group** is like a **folder** or **container** in Azure that holds all the related resources for an application or project (App Service app, App Service plan, accounts, databases, Key Vaults, Application Insights…).

For most plans the Azure might change the current used outbound IPs e. g. when switching the Plan tier or during scale in/out. But the new outbound IPs are taken from limited list of all possible outbound IPs for current scale unit.

#Check all possible outbound Ips of plan (enter any app\_name from plan)

az webapp show \

    --resource-group <group\_name> \

    --name <app\_name> \

    --query possibleOutboundIpAddresses \

    --output tsv

Changing the region of plan requires creating a new plan, so the list of possible outbound IPs will change as well.

Azure App Service **does not guarantee fixed outbound IPs. IPs might be changed without any interruption from developer.**

**If you're using this IP to whitelist firewalls or APIs, you should always whitelist the full possibleOutboundIpAddresses set, not just the current active ones.**

**There are several ways to guarantee static outbound IP, e. g. using App Service Environment.**

**The command** > az webapp up without parameters performs the following actions:

* Create a default resource group if one isn't specified.
* Create a default app service plan.
* Create an app with the specified name.
* Zip deploy files from the current working directory to the web app.

## Application settings configuration

**App settings** – variables passed as environment variables to the application code, they're injected into app environment at app startup. When you add, remove, or edit app settings, App Service triggers an app restart. They are always encrypted when stored. App settings example:

            [

                {

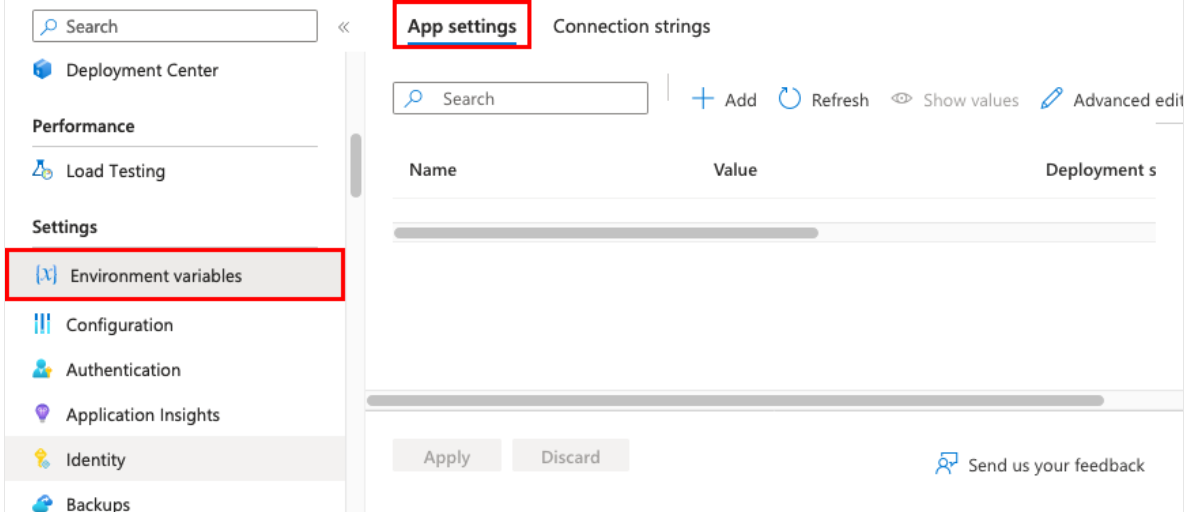
                    "name": "ConnectionStrings:MyDb",

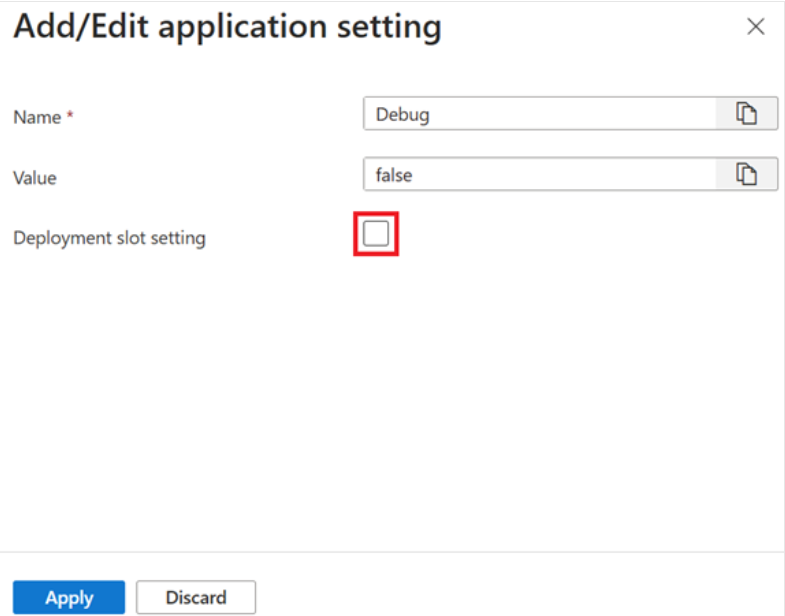
                    "value": "Server=myserver.database.windows.net;Database=mydb;User Id=myuser;Password=mypassword;",

                    "slotSetting": false

                }

            ]





The following setting:

{

  "Logging": {

    "LogLevel": {

      "Default": "Information",

      "Microsoft.AspNetCore": "Warning"

    }

  }

}

In Azure environmental variables must be written as:

{

  "name": "Logging:LogLevel:Default",

  "value": "Debug",

  "slotSetting": false

}

And in ASP.NET app you can access them by:

var level = Configuration["Logging:LogLevel:Default"];

But in Linux Systems environment variable names can’t contain colons “:”. The AAS in Linux-based plan (or in Linux custom containers) will inject name as

    "name": " Logging\_\_LogLevel\_\_Default ",

Exchanging : to double \_

Connection strings can be written in special way – add “type” field and use keywords for certain db engine:

{

  "name": "ICOM-CS",

  "value": "conn-string-1",

  "type": "SQLServer",

  "slotSetting": false

}

App service creates variable with prefix and you can access this CS:

        var conn = Environment.GetEnvironmentVariable("SQLCONNSTR\_ICOM-CS ");

* SQLServer: SQLCONNSTR\_
* MySQL: MYSQLCONNSTR\_
* SQLAzure: SQLAZURECONNSTR\_
* Custom: CUSTOMCONNSTR\_
* PostgreSQL: POSTGRESQLCONNSTR\_
* Notification Hub: NOTIFICATIONHUBCONNSTR\_
* Service Bus: SERVICEBUSCONNSTR\_
* Event Hub: EVENTHUBCONNSTR\_
* Document DB: DOCDBCONNSTR\_
* Redis Cache: REDISCACHECONNSTR\_

Add environmental variable in Bash:

az webapp config appsettings set \

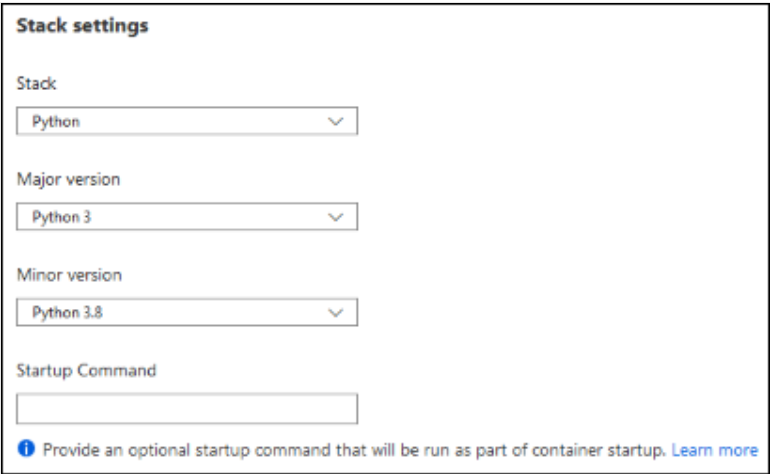
  --resource-group <group-name> \

  --name <app-name> \

  --settings key1=value1 key2=value2

**General settings**

Stack settings:



**Startup Command** - (Linux/Custom Container)

Platform settings:

* **Platform bitness**: 32-bit or 64-bit. For Windows apps only.
* **FTP state**: Allow only FTPS or disable FTP altogether.
* **HTTP version**: Set to **2.0** to enable support for HTTPS/2 protocol.
* **Web sockets**: For ASP.NET SignalR or socket.io, for example.
* **Always On**:Keeps the app loaded even when there's no traffic. When **Always On** isn't turned on (default), the app is unloaded after 20 minutes without any incoming requests. The unloaded app can cause high latency for new requests because of its warm-up time. When **Always On** is turned on, the front-end load balancer sends a GET request to the application root every five minutes.
* **ARR affinity**: In a multi-instance deployment, ensure that the client is routed to the same instance for the life of the session. You can set this option to **Off** for stateless applications.
* **HTTPS Only**
* **Minimum TLS version**

**Debugging**: Enable remote debugging for ASP.NET, ASP.NET Core, or Node.js apps. This option turns off automatically after 48 hours.

**Incoming client certificates: turns on** TLS mutual authentication. **By default the server has a TLS certificate and client validates it. With** TLS mutual authentication server requires a TLS from client and communicates only if trust the client certificate. Trusted certificates are loaded in AAS.

## Configuration > Path mappings

App repo are deployed to D:\home\site\wwwroot. To access files in root in code use

        var path = Path.Combine(\_env.WebRootPath, "myfile.txt");

where \_env is IWebHostEnvironment that must be added through DI (not available in .NET Framework)

If your app root is in a different folder, or if your repository has more than one application, you can edit or add virtual applications and directories.

Public files like .js .html. .png can be accessible via URL:

<https://rekrutacja.invent.ag/Content/img/cz.png> (returns picture)

In **Configuration** **->** **Path mappings**  you can define handler mappings for uncontainerized Windows apps that will define how to handle url access to files with selected extension. URL example:

https://yourapp.azurewebsites.net/myapp/index.php

this is a php script that won’t be handled properly without handler mappings because Azure App Service is **primarily for .NET, Node.js, Python, etc.**

But you can add php script index.php and define handler mapping for php with properties:

* Extension: \*.php
* Script Processor: D:\home\site\wwwroot\php\php-cgi.exe
* Arguments: (optional) like -f or --some-flag

Azure will run php-cgi.exe with your file as input.

For standard languages (like ASP.NET, Python), Azure already sets up the handler for you.

**Files storage**

Windows apps are deployed to D:\home\site\wwwroot and every redeploy, scaling out etc. will wipe out this path and load files again. But you have an access to D:\home and you can place files in other paths – the will persist after redeploying.

It doesn’t work in containerized apps. Containerized apps include all Linux apps and also the Windows and Linux custom containers running on App Service. In containerized apps you need to add **Azure Storage Mount ­– it’s the network file share (like local file system but in network).**

**Mount can have and choose type Azure Blobs or Azure Files**. Windows container apps only support Azure Files. Azure Blobs Mount is not the same as Azure Blobs.  Azure Blobs Mount in AAS is read-only and good for static files.

Why would I need Storage Mount?

* 1. **With a storage mount**. your data (e.g., logs, user uploads) lives **outside the container**, so it **survives restarts, redeployments, or scaling**.
  2. Sharing data between instances. If your app scales out (e.g., 3+ instances), each instance runs in a separate container. You might need to share files between them (e.g., a shared image folder)
  3. Separate code and data. Sometimes it’s helpful to **keep your application code and runtime separate from your data,** like:  
     - Configuration files  
     - Large media files   
     You don’t have to **rebuild your container** just to update data.

Example:

* You have a photo-sharing app that lets users upload pictures
* You want those photos to be available to all app instances and persist even after redeployment
* So, you mount an Azure File Share to /mnt/photos in your container

You can use Azure Storage Mount also on windows app if the default file system not enough.

**Azure Storage Mount Cofigurations**

* **Name**: The display name.
* **Configuration options**: **Basic** or **Advanced**. Select **Basic** if the storage account isn't using service endpoints, private endpoints, or Azure Key Vault. Otherwise, select **Advanced**.
* **Storage accounts**: The storage account with the container you want.
* **Storage type**: **Azure Blobs** or **Azure Files**. Windows container apps only support Azure Files. Azure Blobs only supports read-only access.
* **Storage container**: For basic configuration, the container you want.
* **Share name**: For advanced configuration, the file share name.
* **Access key**: For advanced configuration, the access key.
* **Mount path**: The absolute path in your container to mount the custom storage.
* **Deployment slot setting**: When checked, the storage mount settings also apply to deployment slots.