

Azure App Services - Web Apps

- Introduction
- App Service - Application Types
- Deploy Web Apps
 - Deploying Web App directly from Visual Studio
 - Automate deployment from Dropbox and One Drive
 - Create, configure and deploy packages
 - Implement pre- and post-deployment actions
- App Service plans
 - Create App Service Plan
 - Migrate Web Apps between App Service plans;
 - Create a Web App within an App Service plan
- Configuring Web Apps
 - Application Settings Configuration,
 - Database Connection Strings,
 - Configuring Handlers and Virtual Directories,
- Manage App Services
 - Configure a custom domain name
 - Enable SSL for your custom domain
 - Understanding Deployment Slots and Roll back deployments;
 - App Service Protection
 - Manage Roles for an App service
- Configure Web Apps for scale and resilience
 - Horizontal and Vertical Scaling of a Web App
 - Configure auto-scale
 - Change the size of an instance
- Monitoring, Debugging and Diagnosis
 - Retrieve Diagnostics data
 - View Streaming Logs
 - Configure endpoint monitoring
 - Configuring Alerts
 - Configure diagnostics
 - Use remote debugging

- Monitor Web App resources

Azure App Services Introduction

- Azure App Service is the cloud PaaS service that integrates everything you need to **quickly and easily** build web and mobile apps for **any platform** and **any device**.
- Built for developers, App Service is a **fully managed platform** with powerful capabilities such as built-in DevOps, continuous integration with Visual Studio Team Services and GitHub, staging and production support, and automatic patching.
- App Service is a platform as a service (PaaS), which means that the OS and application stack are managed for you by Azure; you only manage your application and its data.
- Azure manages OS patching on two levels, the physical servers and the guest virtual machines (VMs) that run the App Service resources. Both are updated monthly. These updates are applied automatically, in a way that guarantees the high availability SLA of Azure services.
- New stable versions of supported language runtimes (major, minor, or patch) are periodically added to App Service instances. Some updates overwrite the existing installation, while others are installed side by side with existing versions. An overwrite installation means that your app automatically runs on the updated runtime. A side-by-side installation means you must manually migrate your app to take advantage of a new runtime version.

App Types

App Service allows you to create the following app types from a single development experience:

1. **Web Apps** - Quickly create and deploy mission critical Web apps that scale with your business.
2. **API Apps** - Easily build and consume Cloud APIs.
3. **Logic Apps** - Automate the access and use of data across clouds without writing code.
4. **Function Apps** – Function written by developer and executed without any dedicated hardware.

As a single integrated service, App Service makes it easy to compose the above app types into a single solution, allowing you to easily build apps that target both **web and mobile clients** using the same back-end and integrate with on premise systems as well as popular services such as Office 365 and salesforce.com.

App Service Web Apps is a fully managed **compute platform** that is optimized for hosting websites and web applications. This **platform-as-a-service** (PaaS) offering of Microsoft Azure lets you focus on your business logic while Azure takes care of the infrastructure to run and scale your apps.

The compute resources may be on shared or dedicated virtual machines (VMs), depending on the pricing tier that you choose.

Your code can be in any language or framework that is supported by [Azure App Service](#), such as **ASP.NET**, **ASP.NET Core**, **.NET6**, **Node.js**, **Ruby**, **Java**, **PHP**, or **Python**.

Why Web Apps:

1. Multiple languages and frameworks
2. Application templates in Azure Marketplace. Templates in the Azure Marketplace, such as WordPress, Joomla, and Drupal.
3. DevOps optimization. Continuous integration and Continuous deployment.
4. Test in production using Deployment slots
5. Global scale with high availability. Scale up or out manually or automatically.
6. Connections to SaaS platforms and on-premises data.
7. Visual Studio Integration.
8. App Service is ISO, SOC, and PCI compliant.

Create and Deploy App Service Web Apps

Deploying your app to App Service is a matter of deploying your code, binaries, content files, and their respective directory structure, to the [/site/wwwroot](#) directory in Azure.

1. Create a Web App in Azure Portal

- a. Login to Azure Portal, <https://portal.azure.com/>
- b. Azure Portal → More Services → Web App → + Add
- c. Select Web Apps → Create
- d. Name = "DssDemoWebApp", Subscription = "Free Trail" Resource Group="DemoRG", App Service plan/Location=Create New Plan (Name=Standard_Plan, Location=Central US, Pricing tier=S1 Standard.
- e. Application Insights=Off
- f. Create

2. Visual Studio → Create a new ASP.NET MVC Web Application

- a. File → New → Project
- b. Visual C# → ASP.NET.Core Web Application, Project Name="DemoWebApp" → OK
- c. Select Template = MVC, Change Authentication = No Authentication → OK

Note: In a few seconds, Visual Studio creates the web project in the folder that you specified

3. Deploy / Publish the project from VS.NET.

- a. In **Solution Explorer**, right-click the project, and choose **Publish**.

The wizard opens to a *publish profile* that has settings for deploying the web project to the new web app. If you wanted to deploy to a different web app, you could click the **Profile** tab to create a different profile.

- b. Choose default options and finally **click on Publish**.

The **Output** and **Azure App Service Activity** windows show what deployment actions were taken and report successful completion of the deployment.

Deploying using Eclipse?

<https://docs.microsoft.com/en-us/azure/developer/java/toolkit-for-eclipse/create-hello-world-web-app>

4. Getting Publish Profile from Azure Portal and publishing from Visual Studio

Ideally used when developer doesn't have direct access to Azure Subscription.

- a. Click **App Services**, and then click the name of your web app.
- b. In the tool bar click on **Get Publish Profile**
- c. Save the Profile locally on your disk.
- d. Go to VS.NET, Right Click on Project → Publish
- e. Select Profile Tab → **Click Import** → Provide the downloaded profile name → OK
- f. Click Publish.

Note: Profile will be saved for further use in <Project>/Properties/PublishProfiles/*.pubxml

To gets an Azure Web App publishing profile using PowerShell:

```
Get-AzWebAppPublishingProfile -ResourceGroupName "DemoRG" -Name "DemoWebApp" -Format "WebDeploy" -  
OutputFile "D:\outputfile.publishsettings"
```

5. Publishing using FTP tools like FileZilla/Windows Explorer

- a. Azure Portal → Click **App Services**, and then click the name of your web app.
- b. Go to Settings → select **Deployment Center**
- c. Provide FTP/deployment user name and password
- d. Save
- e. Look at Essentials Section of Selected App Service and copy FTPS hostname and user name
- f. Open **Windows Explore** and use the above hostname and credentials to connect and upload files.

4

Note: The User Credentials: FTP Username/Password is same for all applications in a given subscription.

Note: Although it's easy to copy your web app's files to Azure using FTP utilities, they don't automatically take care of or coordinate related deployment tasks such as deploying a database or changing connection strings. Also, many FTP tools don't compare source and destination files in order to skip copying files that haven't changed. For large Apps, always copying all files can result in long deployment times even for minor updates since all files are always copied.

Automate deployment from Dropbox and One Drive

Dropbox is not a source control system, but if you store your source code in Dropbox you can automate deployment from your Dropbox account.

1. Create a drop box account @ <http://www.dropbox.com>
2. Go to <http://portal.azure.com/>
3. Select the App Service → Settings → Deployment → **Deployment options** → Configure required Settings → Dropbox
4. Authorize Azure to access your drop box
5. Go to <https://www.dropbox.com/> and go to folder Apps → Azure → Create a folder by name: <WebApp Name>
6. Upload the files to the above folder
7. Go to Azure Portal
8. Select the App Service → Settings → Publishing → Deployment Source → Sync the App Service
9. View the page in browser.

The cons of syncing with a cloud folder are:

- No version control for rollback when failures occur.
- No automated deployment, manual sync is required.

Note: Similar steps are required even for One Drive.

ZIP Deploy:

Zip the project and drag and drop to file explorer in the below url

<https://dssdemo-appsrv.scm.azurewebsites.net/ZipDeployUI>

Deploying Java Web App to App Service using Maven Plug-in

Deccansoft Software Services H.No: 153, A/4, Balamrai, Rasoolpura, Secunderabad-500003 TELANGANA, NDIA.

<http://www.deccansoft.com> | <http://www.bestdotnettraining.com>

Phone: +91 40 2784 1517 OR +91 8008327000 (INDIA)

1. Download and extract Maven 3.8.1 from <https://archive.apache.org/dist/maven/maven-3/> to C:\Java
2. Search Environmental Variables and add to Path: C:\Java\apache-maven-3.8.1\bin
3. Run the following command to create a New Tomcat App

```
mvn archetype:generate "-DgroupId=example.demo" "-DartifactId=helloworld" "-DarchetypeArtifactId=maven-archetype-webapp" "-Dversion=1.0-SNAPSHOT"
cd helloworld
```

OR

Clone the Spring Boot getting started sample project

```
git clone https://github.com/spring-guides/gs-spring-boot
cd gs-spring-boot/complete
```

4. Run the Maven command below to configure the deployment. This command will help you to set up the App Service operating system, Java version, and Tomcat version.

```
mvn com.microsoft.azure:azure-webapp-maven-plugin:2.1.0:config
```

Note: You can modify the configurations for App Service directly in your pom.xml if needed.

5. Deploy the Java application to Azure with below command

```
mvn package azure-webapp:deploy
```

App Service Plan

- App Service plans represent the collection of physical resources used to host your apps.
- App Service plans define:
 - Region (West US, East US, etc.)
 - SKU (Free, Shared, Basic, Standard, Premium, Isolated) – Features.
 - Instance size (Small, Medium, Large) – Memory and Processor
 - Scale count (one, two, three instances, etc.)
- Web Apps, Mobile Apps, API Apps, or Functions, in Azure App Service all run in an App Service plan. Apps in the same subscription, region, and resource group can share an App Service plan.
- All applications assigned to an **App Service plan** share the resources defined by it allowing you to save cost when hosting multiple apps in a single App Service plan.

Note:

The requested app service plan cannot be created in the current resource group because it is hosting Linux apps. Please choose a different resource group or create a new one.

	FREE Try for free	SHARED Environment for dev/test	BASIC Dedicated environment for dev/test	STANDARD Run production workloads	PREMIUM Enhanced performance and scale	ISOLATED High- Performance, Security and Isolation
Web, mobile, or API apps	10	100	Unlimited	Unlimited	Unlimited	Unlimited
Disk space	1 GB	1 GB	10 GB	50 GB	250 GB	1 TB
Maximum instances	–	–	Up to 3	Up to 10	Up to 30*	Up to 100
Custom domain	–	Supported	Supported	Supported	Supported	Supported
Auto Scale	–	–	–	Supported	Supported	Supported
Hybrid Connectivity	–	–	Supported	Supported	Supported	Supported
Virtual Network Connectivity	–	–	–	Supported	Supported	Supported
Private Endpoints	–	–	–	–	Supported	Supported
Compute Type	Shared	Shared	Dedicated	Dedicated	Dedicated	Isolated
Price	Free	\$0.013/hour	\$0.075/hour	\$0.10/hour	\$0.20/hour	\$0.40/hour

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

Because a single resource group can have multiple App Service plans, you can allocate different apps to different physical resources that spans geographical regions. For example, a highly available app running in two regions includes at least two plans, one for each region, and one app associated with each plan. In such a situation, all the copies of the app are then contained in a single resource group. Having a resource group with multiple plans and multiple apps makes it easy to manage, control, and view the health of the application.

- It is recommended to isolate an app into a new App Service plan when:
 - App is resource-intensive.
 - App has different scaling factors from the other apps hosted in an existing plan.
 - App needs resource in a different geographical region.
- You can move an app to a different App Service plan in the Azure portal. **You can move apps between plans as long as the plans are in the same resource group and geographical region.**

AS0, AS1, AS6 - AP1 (RG1/East US)

AS2 - AP2 (RG1/West US)

AS3 - AP3 (RG1/East India)

- AP4 (RG1/East US)
- AP5 (RG2/East US)

- You can create an **empty App Service plan** and then select the same while creating an App Service or you can create an App Service Plan while creating an App Service.
- If you want to move the app to a different region, one alternative is app cloning. Cloning makes a copy of your app in a new or existing App Service plan in any region. You can find **Clone App** in the **Development Tools** section of the menu. The web app must be running in the **Standard** mode in order for you to create a clone for the web app.

Scaling a ~~Web App~~ App Service Plan

- Whether your application needs to handle a few hundred requests per day or a few million requests per day, the Azure Web Apps scalability features provide ways for you to deliver the right level of scale in a robust, cost-effective manner.
- When you consider the scalability requirements of an application, you should look at its resource requirements **vertically** (scaling up) **horizontally** (scaling out).
- You typically choose to **scale up** when any single request demands more memory and processing power to complete, and the **bottleneck / latency in the system is the intensive number of software objects created in the computer's memory or the intensive algorithms and business logic that is performed.** When you scale up a web app, you increase the resource capacity, **such as RAM and CPU cores**, of the virtual machine on which your web app is running.
- You typically **scale out** when any single request requires **less** memory and processing power to complete, but the real **bottleneck / latency is in network communication, disk access, etc.** In this case, the key to completing each request more efficiently is to **run it in parallel** to other requests as each wait on external components to complete. To scale out a web app, you **increase the number of virtual machine/app service instances** on which your web app is running. For the properly architected app, this means your web app can handle more load and therefore service more user requests.

If you scale an app in the Basic tier to two instances, you have 350 concurrent connections for each of the two instances. For Standard tier and above, there are no theoretical limits to web sockets, but other factors can limit the number of web sockets. For example, maximum concurrent requests allowed (defined by maxConcurrentRequestsPerCpu) are: 7,500 per small VM, 15,000 per medium VM (7,500 x 2 cores), and 75,000 per large VM (18,750 x 4 cores).

Scale Up (Vertical Scaling) the Azure Web App:

- The ability to scale up a web app exists only for web apps configured for Basic, Standard, or Premium pricing tiers.
- The scale settings take only seconds to apply and affect all web apps in your App Service plan. They do not require your code to be changed or your applications to be redeployed.

To Scale Up

1. App Services → Select App Service → Settings → **Change App Service Plan** (In App Service Plan) → Select / Create New Plan
2. Select the Pricing tier based on following options:
 - a. **Number of Cores**
 - b. **RAM**
 - c. Storage
 - d. Slots (Number of CPU Instances)
 - e. Backup frequency
 - f. Traffic Manager facility

To Scale Out: (Horizontal Scaling)

The number of Virtual Machine Instances you can scale out is limited by the pricing tier configured for your web app.

1. App Services → Select App Service → Settings → App Service Plan
2. Select Scale Out (App Service Plan) to configure settings
 - a. Scale by: **Manual** - Manual setup means that the number of instances you choose won't change, even if there are changes in load.
 - i. Basic = 3 Instances
 - ii. Standard = 10 Instances
 - iii. Premium P1/P2 = 20 Instances
 - iv. Premium P3 = 30 Instances
 - b. Scale by: **CPU percentage**: Automatically scale based on CPU Percentage used. You can choose an average value you want to target.
 - c. Scale by: **Schedule and Performance Rules** - Create your own set of rules. Create a schedule that adjusts your instance counts based on time and performance metrics.

Autoscale Metrics

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

Auto scale based on CPU percentage:

- The Target range setting defines the **minimum** and **maximum** CPU percentage to target.
- As long as the CPU percentage is within this range, Autoscale will not increase or decrease the number of instances.
- When the CPU percentage exceeds the maximum CPU percentage you specify, Autoscale will add an instance. If CPU percentage continues to exceed the maximum CPU specified, then Autoscale will add another instance. At no point will you have more than the maximum number of instances specified in the Instances setting.
- Similarly, when CPU percentage falls below the minimum CPU percentage you specify, Autoscale will remove an instance. If CPU percentage continues to fall below the minimum CPU percentage specified, then Autoscale will remove another instance. At no point will you have fewer than the minimum number of instances specified in the Instances setting

Note: The CPU percentage is measured as an **average across all instances**. For example, if you have two instances, one of which is running at 50 percent CPU and the other of which is running at 100 percent CPU, then the CPU percentage would be 75 percent for all the instances at that point in time

Scale out			
When	dssdemo-appsrvplan	(Average) CpuPercentage > 70	Increase count by 1
Or	dssdemo-appsrvplan	(Average) MemoryPercentage ...	Increase count by 1
Scale in			
When	dssdemo-appsrvplan	(Average) CpuPercentage > 25	Decrease count by 1
And	dssdemo-appsrvplan	(Average) MemoryPercentage ...	Decrease count by 1

App Services Configuration

Azure Portal:

App Services → Select App Service → Settings → **Configuration**

1. .NET version = .NET Core 6.0
2. Enable Debugging
3. Add Key – Value pair to the App Settings, Note that this overwrites the same key added to web.config / appSettings.json in VS.NET project and published.
4. Add Connection String if required
5. Set Default page for the web site.
6. HttpHandler mappings can be Set
7. Virtual applications and directories can be added

ASP.NET Core:

appSettings.json

```
"Name": {
  "FirstName": "Sandeep",
  "LastName": "Soni"
}
```

HomeController.cs

```
string name;
public HomeController(IConfiguration configuration)
{
    name = configuration["Name:FirstName"];
}
```

```
public IActionResult Index()
{
    ViewBag.Name = name;
    return View();
}
```

Index.cshtml

```
<div>
    @ViewBag.Name
</div>
```

web.config OR appSettings.json = Least Precedence

web.release.config / web.debug.config OR appSettings.json / appSettings.Development.json

Azure → App Service → Configuration → Application Settings = Highest Precedence

Default documents:

- Only available for **Windows** apps
- List of documents to show when navigating to a directory on the web server:
 - First matching file is used
- Alternative to building a custom module

CORS:

- Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the **same-origin policy**. The same-origin policy prevents a malicious site from accessing sensitive data on another site.
- In a standard CORS implementation, the JavaScript client will send a **pre-flight request** by using an **OPTIONS** verb to assess the server's willingness to accept a cross-site request. If allowed, the JavaScript client will then issue a **cross-site HTTP request**.
- Servers can specify:
 - Allowed HTTP verbs
 - Allowed origins
 - Allowed headers



Deployment Slots

- A deployment slot technically is an **independent** web app with its own content, configuration, and even a unique host name. So, it functions just like any other web app.
- Each Slot is reachable from its unique URL. For example for Staging deployment slot:
`http://dssdemoapp.azurewebsites.net/`
`http://dssdemoapp-staging.azurewebsites.net/`
- This option is available only in **Standard** and above pricing tier.

Benefits of Deployment Slots:

1. You can deploy changes for your application to a **staging deployment slot** and test the changes without impacting users who are accessing the **production deployment slot**. When you are ready to move the new features into production, you can just **swap the staging and production slots with no downtime (Blue-Green Deployment)**
2. You can **swap back** to the previous deployment if you realize that the new version of your application is not working as you expected.
3. You can "**warm up**" your application in a staging slot before swapping it into the production slot, avoiding the long delays a cold start of your application may incur because of some lengthy initialization code.
4. You can do **A/B testing** with a small set of users to try out new features of your application without impacting the majority of users who are using the production slot.

Note: A/B testing (also known as split testing or bucket testing) is a method of comparing two versions of a webpage or app against each other to determine which one performs better. AB testing uses data & statistics to validate new design changes and improve your conversion rates.

5. Can be used for **Canary Deployments**.

Slot Supports basic on Plan

Basic = No slots support

Standard = 5 Slots

Premium = 20 Slots

Configuration for deployment slots

When you clone configuration from another deployment slot, the cloned configuration is editable. Furthermore, some configuration elements will follow the content across a swap (not slot specific) while other configuration elements will stay in the same slot after a swap (slot specific).

Settings that are swapped:	Settings that are not swapped:
<ul style="list-style-type: none"> General settings - such as framework version, 32/64-bit, Web sockets... App settings (can be configured to stick to a slot) Connection strings (can be configured to stick to a slot) Handler mappings Monitoring and diagnostic settings. WebJobs content 	<ul style="list-style-type: none"> Publishing endpoints (URL) Custom Domain Names SSL certificates and bindings Scale settings WebJobs schedulers

Adding a Deployment slot:

1. App Services → Select App Service → Settings → Deployment Slots (Publishing) → Add Slot (blade)
2. Set Name and Configuration Source

Format of **Domain Name** for Deployment Slot = <WebApp>-<DeploymentSlotName>.azurewebsite.net

To Swap with Production:

1. Go to Deployment Slot Blade → Swap
2. Set Swap type, Source and Destination.

Note: Make sure that the swap source and swap target are set properly. Usually, the swap target is the production slot.

Auto Swap?

Auto swap streamlines Azure DevOps scenarios where you want to deploy your app continuously with zero cold starts and zero downtime for app customers.

When auto swap is enabled from a slot into production, every time you push your code changes to that slot App Service automatically swaps the app into **production after it's warmed up in the source slot**.

Select App Service → Deployment Slots → Select Slot (staging) → Configuration → Deployment Slot section, **Auto swap enabled** = On

Route traffic between slots (Canary Deployment)

- All traffic is normally routed to production:
 - Production slot has 100% weighting
- You can manually configure the weight of traffic between multiple slots

Swap with Preview?

Swap with preview breaks down a normal swap into two phases. In phase one, any slot-specific application settings and connections strings on the destination will be temporarily copied to the source slot. This allows you to test the slot with its final configuration values. From here, you may choose to either cancel phase one to revert to your normal configuration, or proceed to phase two, which would remove the temporary config changes and complete swapping the source to destination slot.

Note: It is possible only if there is atleast one slot specific application settings.

Configure a Custom Domain Name in Azure App Service

Step 1: Reserve the domain name. There are many domain registrars to choose from eg: GoDaddy.com

Step 2: Create **DNS records** that map the domain to your Azure web app.

The Domain Name System (DNS) uses data records to map domain names into IP addresses. There are several types of DNS records. For web apps, you'll create either an **A record** or a **CNAME record**.

1. An **A (Address)** record maps a domain name to an IP address.
2. A **CNAME (Canonical Name)** record maps a domain name to another domain name. DNS uses the second name to look up the address. Users still see the first domain name in their browser. For example, you could map contoso.com to <yourwebapp>.azurewebsites.net.

www.dssdemoapp.com

3. Create CName record (www.dssdemoapp.com => dssdemoapp.azurewebsites.net)
4. Create Txt Record (asuid.www => 6D7F5E02604A7CC6A9FD19B8022AC8DD505ED4D51ADE0AB3D37B31ED639214C4)

dssdemoapp.com

5. Create a ARecord (@ => IP of Service) (Only if we want to map Main domain eg: dssdemoapp.com)
6. Create Txt Record (asuid => 6D7F5E02604A7CC6A9FD19B8022AC8DD505ED4D51ADE0AB3D37B31ED639214C4)

Note: If the IP address changes, a CNAME entry is still valid, whereas an A record must be updated. However, some domain registrars do not allow CNAME records for the root domain or for wildcard domains. In that case, you must use an A record.

Step 3: Add the domain name inside the Azure Portal.

1. App Services → Select App Service → Settings → **Custom domains** → Add Custom Domain
2. Use the **DOMAIN NAMES** text boxes to enter the domain names to associate with this web app.

When does Inbound IP Address change:

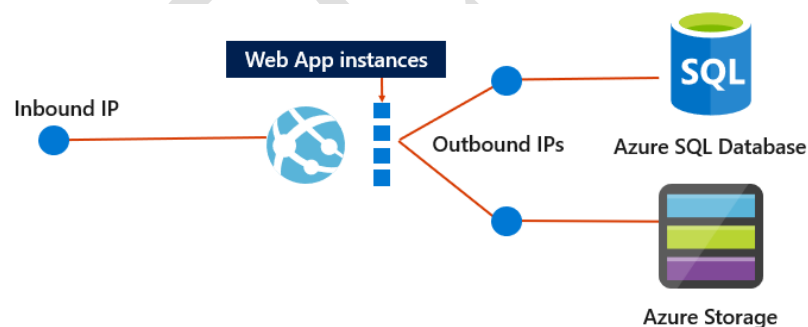
Regardless of the number of scaled-out instances, each app has a **single inbound IP address**.

The inbound IP **address might change** when you perform one of the following actions:

- Delete an app and recreate it in a different resource group.
- Delete the last app in a resource group and region combination and recreate it.
- Delete an existing SSL binding, such as during certificate renewal.

Outbound IP addresses:

- Regardless of the number of scaled-out instances, each app has a **set number of outbound IP addresses** at any given time. Any outbound connection from the App Service app, such as a connection to a back-end database, uses one of the outbound IP addresses as the origin IP address.
- You can't know beforehand which IP address a given app instance will use to make the outbound connection. Therefore, your back-end service must open its firewall to all the outbound IP addresses of your app.



You can find the same information by observing your app in the Azure portal. Select **Properties** in your app's left-hand navigation.

OR

Find Outbound IP address


```
az webapp show --resource-group <group_name> --name <app_name> --query outboundIpAddresses --output tsv
```

Find all possible IP addresses (regardless of tier)

```
az webapp show --resource-group <group_name> --name <app_name> --query possibleOutboundIpAddresses --output tsv
```

The set of **outbound IP addresses** for your app **changes** when you scale your app between the lower tiers (Basic, Standard, and Premium) and the Premium V2 tier.

Enable SSL for your custom domain

Step 1: Get the Certificate for the custom domain from the Certificate Authority (Verisign)

Step 2: Configure Standard pricing tier:

Enabling HTTPS for a custom domain is only available for the **basic** tier and above in Azure App Service. Use the following steps to switch your App Service plan to S2 **Standard** tier.

1. App Services → Select App Service → Settings → App Service Plan → Pricing Tier → **S1 Standard**

Step 3: Configure SSL in your App

3. App Services → Select App Service → Settings → SSL Settings
4. In the certificates section, click Upload
5. Upload Certificate file and provide the password
6. In the **SSL bindings** section of the SSL Settings tab, use the dropdowns to select the domain name to secure with SSL, and the certificate to use. You may also select whether to use [Server Name Indication](#) (SNI) or IP based SSL.

How SSL Works:

https://www.deccansoft.com =>

1. Browser goes to server and finds who is the certificate authority (CA).
2. It will go to CA store and get client certificate of that domain.
3. All data before sending to server is encrypted using the public key in the certificate.
4. Server receives the request and decrypt the data using the private key in the certificate.

Important Note: Sometimes you might want a dedicated, **static IP address** for your app. To get a static inbound IP address, you need to configure an **IP-based SSL binding**.

If you don't actually need SSL functionality to secure your app, you can even upload a self-signed certificate for this binding. In an IP-based SSL binding, the certificate is bound to the IP address itself, so App Service provisions a static IP address to make it happen.

App Service Authentication and Authorization

Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your web app, RESTful API, and mobile back end, and also Azure Functions. App Service uses federated identity, in which a third-party identity provider manages the user identities and authentication flow for you. Five identity providers are available by default:

Provider	Sign-in endpoint
Microsoft Account	/.auth/login/microsoftaccount
Facebook	/.auth/login/facebook
Google	/.auth/login/google
Twitter	/.auth/login/twitter

Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK:

- **Without provider SDK:** The application delegates federated sign-in to App Service. This is typically the case with **browser apps**, which can present the provider's login page to the user. The server code manages the sign-in process, so it is also called **server-directed flow** or **server flow**. This case applies to browser apps. It also applies to native apps that sign users in using the Mobile Apps client SDK because the SDK opens a web view to sign users in with App Service authentication.
- **With provider SDK:** The application signs users in to the provider manually and then submits the authentication token to App Service for validation. This is typically the case with **browser-less apps**, which can't present the provider's sign-in page to the user. The application code manages the sign-in process, so it is also called **client-directed flow** or **client flow**. This case applies to REST APIs, [Azure Functions](#), and JavaScript browser clients, as well as browser apps that need more flexibility in the sign-in process. It also applies to native mobile apps that sign users in using the provider's SDK.

Step	Without provider SDK	With provider SDK
------	----------------------	-------------------

1. Sign user in	Redirects client to <code>/.auth/login/<provider></code> .	Client code signs user in directly with provider's SDK and receives an authentication token.
2. Post-authentication	Provider redirects client to <code>/.auth/login/<provider>/callback</code> .	Client code posts token from provider to <code>/.auth/login/<provider></code> for validation.
3. Establish authenticated session	App Service adds authenticated cookie to response.	App Service returns its own authentication token to client code.
4. Serve authenticated content	Client includes authentication cookie in subsequent requests (automatically handled by browser).	Client code presents authentication token in X-ZUMO-AUTH header (automatically handled by Mobile Apps client SDKs).

Authorization behavior

In the Azure portal, you can configure App Service authorization with a number of behaviors.

a) Allow all requests (default)

Authentication and authorization are not managed by App Service (turned off).

Choose this option if you don't need authentication and authorization, or if you want to write your own authentication and authorization code.

b) Allow only authenticated requests

The option is **Log in with <provider>**. App Service redirects all anonymous requests to `/.auth/login/<provider>` for the provider you choose. If the anonymous request comes from a native mobile app, the returned response is an HTTP 401 Unauthorized.

With this option, you don't need to write any authentication code in your app. Finer authorization, such as role-specific authorization, can be handled by inspecting the user's claims.

c) Allow all requests, but validate authenticated requests

The option is **Allow Anonymous requests**. This option turns on authentication and authorization in App Service, but defers authorization decisions to your application code. For authenticated requests, App Service also passes along authentication information in the HTTP headers.

Register your application with Facebook

1. Navigate to <https://developers.facebook.com/>
2. If you have not already registered, click **Apps > Register as a Developer**, then accept the policy and follow the registration steps.

3. My Apps → Add a New App
4. Display Name = "Demo Web App", contact email = <your login> → Create App ID
5. +Add Product → Facebook Login → Set Up → Web
6. Facebook Login → Settings → Client OAuth Settings Section, **Valid OAuth redirect URIs** =
<https://dssdemoapp.azurewebsites.net/.auth/login/facebook/callback>
7. Settings → Basics → Copy and Store Application ID and App Secret,
8. Settings → Basics
 1. Set Privacy Policy URL = <https://www.bestdotnettraining.com/pdf/PrivacyAgreement.pdf>
Terms of Service URL = <https://www.bestdotnettraining.com/pdf/TermsAndConditions.pdf>
 2. Save Changes
9. Make App public: Switch Status from Development to Live

Add Facebook information to your application

10. App Service → Settings → **Authentication**
11. App Service Authentication = On
12. **Action to take when request is not authenticated** = Facebook.
13. Click **Facebook**, paste in the App ID and App Secret values which you obtained previously.
14. Save.
15. Navigate to web app and note that you are redirected to Facebook to Authenticate.

Steps For Google Authentication: <https://docs.microsoft.com/en-us/azure/app-service/configure-authentication-provider-google>

Note: If required you can use **ClaimsPrincipal.Current** to get all claims from the Authentication Provider.

Backup and Restore your app in Azure

Snapshots automatically create periodic restore points of your app when hosted in a **Standard or Premium App** Service plan.

The Backup and Restore feature in Azure App Service lets you easily create app backups **manually** or on a **schedule**. You can restore the app to a snapshot of a previous state by overwriting the existing app or restoring to another app.

App Service can back up the following information to an Azure storage account and container that you have configured your app to use.

- App configuration
- File content
- Database connected to your app







Backups can be up to 10 GB of app and database content (max 4GB). If the backup size exceeds this limit, **you get an error.**

1. Azure Portal → App Services → Select the App Service → Settings → Backups
2. Configure the Backup
 - Backup Storage
 - Backup Schedule
 - Backup Database
3. Backup page → Click Backup Button
4. You see a progress message during the backup process.

Configure Partial Backups

1. Click **Advanced Tools** -> **Go** setting for your web app to access Kudu
2. Identify the folders that you want to exclude from your backups. For example, you want to filter out the highlighted folder and files.

... / Images + | 6 items

	Name
	2013
	2014
	2015
	Products
	bkg.png
	brand.png

3. Create a text file called **D:\home\site\wwwroot_backup.filter** and put the preceding list in the file. List one directory or file per line.

So the content of the file should be:

```
\site\wwwroot\Images\brand.png
\site\wwwroot\Images\2019
\site\wwwroot\Images\2020
```

If you wish, you can create the file directly using Kudu [advanced tools](#) and insert the content there.

Integrate an App with an Azure Virtual Network

VNet Integration enables apps to access resources in or through a VNet.

Azure App Service has two variations on vNet Integration:

- Multitenant systems that support the full range of pricing plans except Isolated.
- **App Service Environment (ASE)**, which deploys into your VNet and supports **Isolated pricing plan** apps and **doesn't** require use of the VNet Integration feature.

VNet Integration gives your app access to resources in a VNet, but it doesn't grant inbound private access to your app from the VNet.

VNet Integration is used only to make outbound calls from your app into your VNet. The VNet Integration feature behaves differently when it's used with VNet in the same region and with VNet in other regions.

1. **Regional VNet Integration:** When you connect to Azure Resource Manager virtual networks **in the same region**, you must have a dedicated subnet in the VNet you're integrating with.
2. **Gateway-required VNet Integration:** When you connect to VNet in **other regions** or to a **classic virtual network** in the same region, you need an **Azure Virtual Network Gateway with Point to Site** provisioned in the target VNet.

Azure App Service → Networking → VNet Integration → Click here to configure → + Add VNet

Using regional VNet Integration enables your app to access:

- Resources in a VNet in the same region as your app. Private IP address of resources in VNet of can be used which programming code in App Service.
- Resources in VNets peered to the VNet your app is integrated with.
- Service endpoint secured services.
- Resources across Azure ExpressRoute connections.
- Resources in the VNet you're integrated with.
- Private endpoints.

When you use VNet Integration with VNets in the same region, you can use the following Azure networking features:

- **Network security groups (NSGs):** You can block outbound traffic with an NSG that's placed on your integration subnet. The inbound rules don't apply because you can't use VNet Integration to provide inbound access to your app.
- **Route tables (UDRs):** You can place a route table on the integration subnet to send outbound traffic where you want.

By default, your app routes only RFC1918 traffic (Private IP of VNet Resources) into your VNet. If you want to route all of your outbound traffic into your VNet, apply the app setting **WEBSITE_VNET_ROUTE_ALL** to your app.

To configure the app setting:

1. Go to the Configuration UI in your app portal. Select New application setting.
2. Enter **WEBSITE_VNET_ROUTE_ALL** in the Name box, and enter 1 in the Value box.

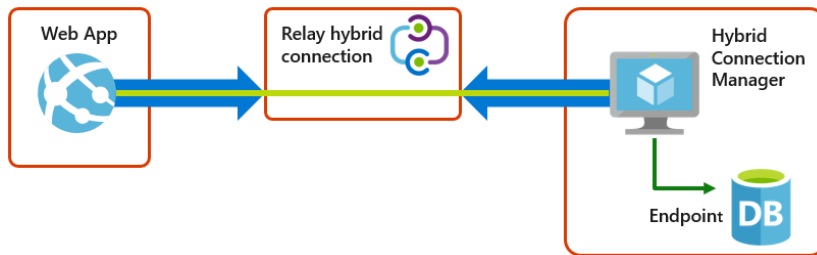
Azure App Service Hybrid Connections

- Hybrid Connections is both a service in Azure and a feature in Azure App Service. As a service, it has uses and capabilities beyond those that are used in App Service.
- Within App Service, Hybrid Connections can be used to access application resources in other networks. It provides access from your app to an application endpoint. It does not enable an alternate capability to access your application.
- As used in App Service, each Hybrid Connection correlates to a single TCP host and port combination. This means that the Hybrid Connection endpoint can be on any operating system and any application, provided you are accessing a TCP listening port.
- The Hybrid Connections feature does not know or care what the application protocol is, or what you are accessing. It is simply providing network access.

How it works

The Hybrid Connections feature consists of two outbound calls to Azure Service Bus Relay. There is a connection from a library on the host where your app is running in App Service. There is also a connection from the Hybrid Connection Manager (HCM) to Service Bus Relay. The HCM is a relay service that you deploy within the network hosting the resource you are trying to access.

Through the two joined connections, your app has a TCP tunnel to a fixed **host:port** combination on the other side of the HCM. The connection uses TLS 1.2 for security and shared access signature (SAS) keys for authentication and authorization.



When your app makes a DNS request that matches a configured Hybrid Connection endpoint, the outbound TCP traffic will be redirected through the Hybrid Connection.

Note: This means that you should try to always use a DNS name for your Hybrid Connection. Some client software does not do a DNS lookup if the endpoint uses an IP address instead.

App Service Hybrid Connection benefits

There are a number of benefits to the Hybrid Connections capability, including:

- Apps can access on-premises systems and services securely.
- The feature does not require an internet-accessible endpoint.
- It is quick and easy to set up.
- Each Hybrid Connection matches to a single host:port combination, helpful for security.
- It normally does not require firewall holes. The connections are all outbound over standard web ports.
- Because the feature is network level, it is agnostic to the language used by your app and the technology used by the endpoint.
- It can be used to provide access in multiple networks from a single app.

Things you cannot do with Hybrid Connections

Things you cannot do with Hybrid Connections include:

- Mount a drive.
- Use UDP.
- Access TCP-based services that use dynamic ports, such as FTP Passive Mode or Extended Passive Mode.
- Support LDAP, because it can require UDP.
- Support Active Directory, because you cannot domain join an App Service worker.


```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=Password_*123" --name "sql-demo" -e "MSSQL_PID=Developer" -d -p 1412:1433 -v db-data-sqlsrv:/var/opt/mssql mcr.microsoft.com/mssql/server:2019-latest
```

Edit the below file as Administrator

C:\Windows\System32\drivers\etc\hosts

127.0.0.1 sandeepmachine

Azure Traffic Manager

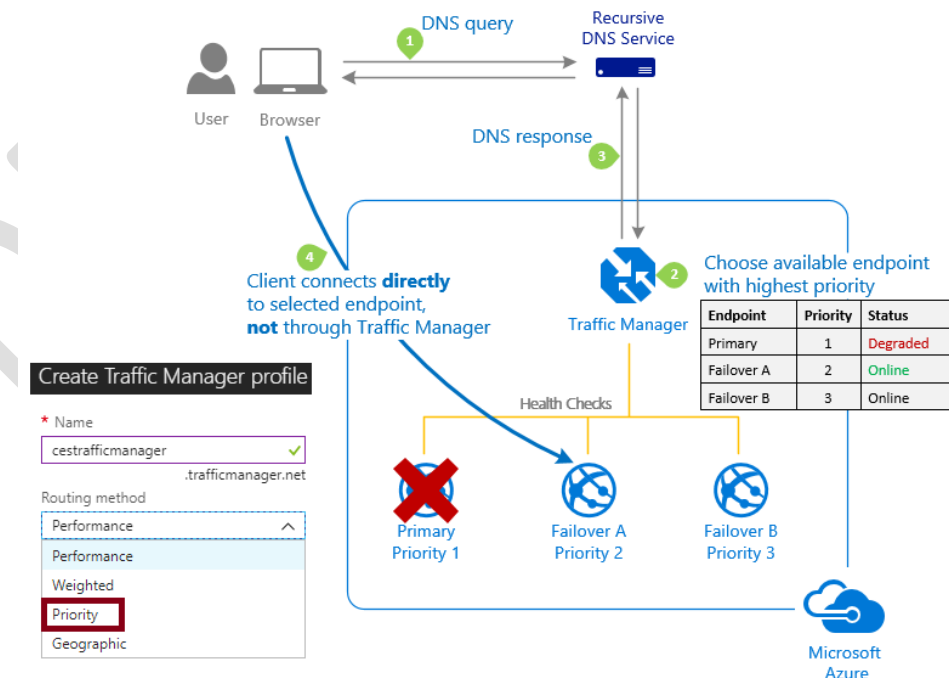
Microsoft Azure Traffic Manager allows you to control the distribution of user traffic for service endpoints in **different datacenters around the world**.

Service endpoints supported by Traffic Manager include Azure VMs, Web Apps, and cloud services. You can also use Traffic Manager with external, non-Azure endpoints.

Traffic Manager uses the **Domain Name System (DNS)** to direct client requests to the most appropriate endpoint based on a **traffic-routing method** and the health of the endpoints.

There are four traffic routing methods available in Traffic Manager:

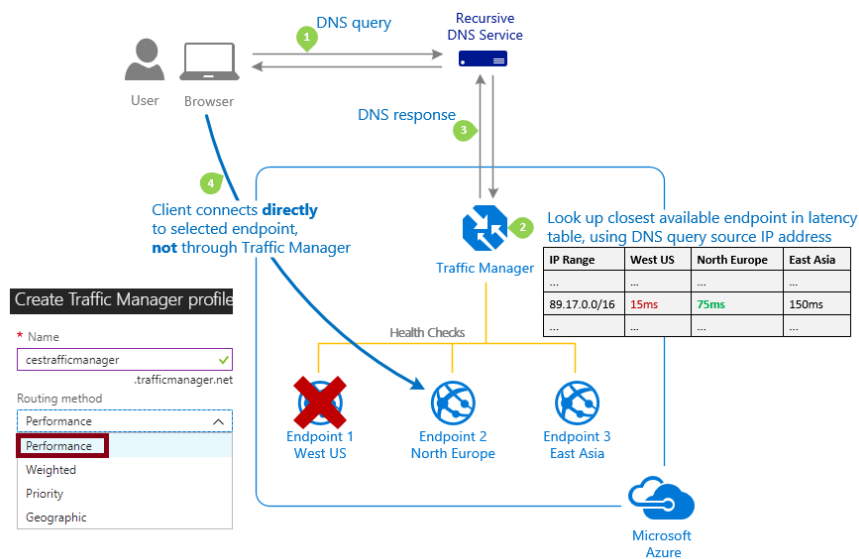
1. **Priority:** Select 'Priority' when you want to use a primary service endpoint for all traffic, and provide backups in case the primary or the backup endpoints are unavailable.



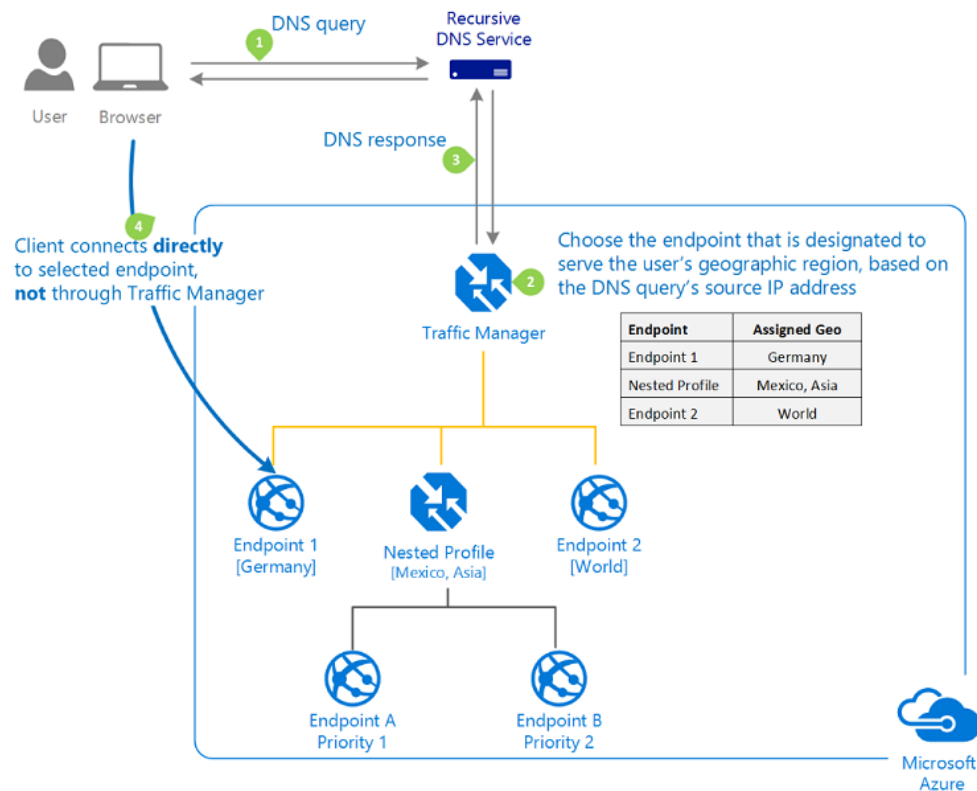
2. **Performance:** Select 'Performance' when you have endpoints in different geographic locations and you want end users to use the "**closest**" endpoint in terms of the **lowest network latency**.

The closest endpoint is not necessarily measured by geographic distance. Instead Traffic Manager determines closeness by **measuring network latency**. Traffic Manager maintains an Internet Latency Table to track the round-trip time between IP address ranges and each Azure datacenter.

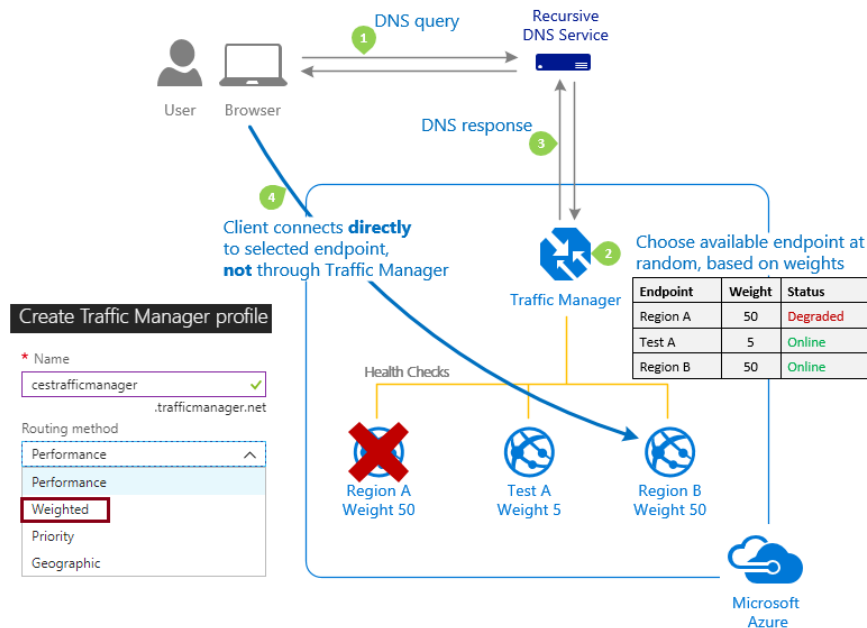
With this method Traffic Manager looks up the source IP address of the incoming DNS request in the Internet Latency Table. Traffic Manager chooses an available endpoint in the Azure datacenter that has the lowest latency for that IP address range, then returns that endpoint in the DNS response.



3. **Geographic:** Select 'Geographic' so that users are directed to specific endpoints (Azure, External or Nested) based on **which geographic location their DNS query originates** from. This empowers Traffic Manager customers to enable scenarios where knowing a user's geographic region and routing them based on that is important. Examples include complying with data sovereignty mandates, localization of content & user experience and measuring traffic from different regions.



4. **Weighted:** Select 'Weighted' when you want to distribute traffic across a set of endpoints, either evenly or according to weights, which you define. The weight is an integer from 1 to 1000. The higher weight, the higher the priority.



Benefits of Traffic Manager

1. Improve availability of critical applications.
2. Improve responsiveness for high performance applications.
3. Upgrade and perform service maintenance without downtime.
4. Combine on-premises and Cloud-based applications.
5. Distribute traffic for large, complex deployments.

To implement Traffic Manager

1. Deploy the Web Apps in different Geographical locations
2. Browse → Traffic Manager profiles → Add
3. Set Name=Demo, Routing Method = Weighted → Create
4. Go to Traffic Manager → Settings → End Points → Add
5. Type = Azure EndPoint, Name=WebApp1EP, Target Resource Type = App Service, Choose an App Service, Weight = 1 → OK
6. Repeat step 5 for every Web App deployment.

Monitoring, Debugging and Diagnostics

- Visual Studio, and the Azure platform collectively provide a rich set of services and tools that you can use to monitor and troubleshoot your applications.

- You can monitor your application **in real time** and interact with **near-real-time data** using the Azure portal. Or you can have the platform alert you if performance degrades or your application becomes unavailable.
- If you need to debug your app live or post-mortem, you will find the Azure Web Apps platform rich with data and analysis to get you to the root cause and resolution as fast as possible.

Remote debugging:

It enables you to debug your web app interactively while it is running in Azure.

Step1: Enable Remote Debugging

In Portal

Settings → Application Settings → Debugging →

Select Remote Debugging = On,

Select Remote Visual Studio version = 2017/2019

Step 2: Publish the **Debug Build** Configuration

Step 3: In Server Explorer, right-click the web app and **select Attach Debugger**.

Diagnostic logs for a web app generally fall into one of two categories:

1. **Application Diagnostic Logs** are generated as a result of **logging code** you add to your application
2. **Site diagnostic logs** are generated automatically by a monitoring service configured on the web server on which your web app is running.

The types of logs that can be enabled for a web app are as follows:

1. Application Logging

- These are logs that are written specifically from your application code using diagnostic classes such as the **System.Diagnostics.Trace** or the **System.Diagnostics.Debug**.
- When you enable application logging, you also must specify a **log level**, which can be **Error**, **Warning**, **Information** or **Verbose**.

```
public ActionResult Index()
{
    Trace.TraceError("Something is definitely wrong here.");
    Trace.TraceWarning("Something could be wrong here.");
    Trace.TraceInformation("Entered {0}.", this.GetType().Name);
    Debug.WriteLine("This is a debug only trace message.");
    return View();
}
```

```
}
```

2. Web Server Logging

- These are HTTP logs (that is, IIS logs) that are written by the web server on which your web app is running.
- Data in these logs contains fields defined in the W3C extended log file format defined at <https://msdn.microsoft.com/library/windows/desktop/aa814385.aspx> and includes things such as the **time it took** the server to process a request, **cookies** that were sent to the client or received by the client, the **client's IP address**, and much more.

3. Detailed error messages

- These are HTML files written by the web server for any requests to the server that result in an HTTP status code 400 or above response. For example, if you request a resource that doesn't exist on the server, you will get an HTTP 404 (Not Found) response.
- With detailed error messages enabled, an HTML file also will be generated containing suggested causes, possible solutions, and additional details about the request.

4. Failed request tracing

- These are XML files written by the web server containing a deeper level of trace information for failed requests.
- These logs contain visibility into the HTTP modules that were invoked when processing the request, time taken in each module, module tracing messages, and more.
- A new XML file is generated for each failed request and is named **fr<x>.xml** where <x> is an incrementing number. Failed request logs are intended to be viewed using a browser, and Azure Web Apps facilitates this by generating a **style sheet file** named in the directory where these files are stored.

Enable application and site diagnostic logs:

App Service → Settings → **App Service Logs** → Enable features as required

Log files for an Azure web app are stored on the web server's file system hosting your web app.

1. Application logs: D:\home\LogFiles\application
2. Web server logs: D:\home\LogFiles\http\RawLogs
3. Detailed error messages: D:\home\LogFiles\DetailedErrors
4. Failed request tracing: D:\home\LogFiles\W3SVC<x>, where <x> is a random number

Note: To Access log files stored in the web app file system either FTP can be used or using Server Explorer in VS.NET

OR

Azure Portal → Select Web App → Settings → Advanced Settings (**Kudu**) → **Debug Console** → **Cmd**

OR

Azure Portal → Select Web App → Settings → Console (Development tools) → Change to directory as required.

Log Stream:

The log-streaming service in Azure Web Apps enables you to view application logs, web server logs, and detailed error messages in nearly real time.

Azure Portal → Select Web App → Settings → Log Stream

Log streaming using Visual Studio:

Server Explorer → **Double Click App Service** → **Right Click on Web App** → **View Streaming Logs**

In the Output window, you will see a message stating **You Are Now Connected to Log-Streaming Service**.

Note: You can change the logs the log-streaming service monitors by clicking the gear icon in the toolbar.

Using Site Control Manager (Kudu) to retrieve log files

Site Control Manager, often referred to as "Kudu", is a website extension that you can use to retrieve log files, browse the file system, edit files, delete files, view environment variables, and even capture diagnostic dump files.

To access the Site Control Manager, open your browser and navigate to

<https://<your site name>.scm.azurewebsites.net>

App Service Environment

The Azure App Service Environment is an Azure App Service feature that provides a **fully isolated and dedicated environment** for securely running App Service apps at high scale.

App Service environments (ASEs) are appropriate for application workloads that require:

- Very high scale.
- Isolation and secure network access.
- High memory utilization.
- Customers can create multiple ASEs within a single Azure region or across multiple Azure regions. This flexibility makes ASEs ideal for horizontally scaling stateless application tiers in support of high Remote PowerShell workloads.

- ASEs are isolated to running only a single customer's applications and are always deployed into a virtual network. Customers have fine-grained control over inbound and outbound application network traffic. Applications can establish high-speed secure connections over virtual private networks (VPNs) to on-premises corporate resources.
- Apps also frequently need to access corporate resources such as internal databases and web services. If you deploy the ASE in a virtual network that has a VPN connection to the on-premises network, the apps in the ASE can access the on-premises resources.

Scaling ASE

- An ASE is dedicated exclusively to a single subscription and can host 100 App Service Plan instances. The range can span 100 instances in a single App Service plan to 100 single-instance App Service plans, and everything in between.
- ASE is composed of front ends and workers. Front ends are responsible for HTTP/HTTPS termination and automatic load balancing of app requests within an ASE. Front ends are automatically added as the App Service plans in the ASE are scaled out.
- Workers are roles that host customer apps. Workers are available in three fixed sizes:
 - One vCPU/3.5 GB RAM
 - Two vCPU/7 GB RAM
 - Four vCPU/14 GB RAM
- Customers do not need to manage front ends and workers. All infrastructure is automatically added as customers scale out their App Service plans. As App Service plans are created or scaled in an ASE, the required infrastructure is added or removed as appropriate.

Pricing ASE

There is a flat monthly rate for an ASE that pays for the infrastructure and doesn't change with the size of the ASE. In addition, there is a cost per App Service plan vCPU. All apps hosted in an ASE are in the Isolated pricing SKU.

There are two ways to deploy an App Service environment (ASE):

- With a VIP on an external IP address, often called an External ASE.
- With the VIP on an internal IP address, often called an ILB ASE because the internal endpoint is an internal load balancer (ILB).

NOTE1: After you create your ASE, you can't change the following:

- Location
- Subscription
- Resource group
- VNet used
- Subnet used
- Subnet size

NOTE2: When you choose a VNet and specify a subnet, make sure that it's large enough to accommodate future growth and scaling needs. Microsoft recommend a size of `/24` with 256 addresses.

Create an ASE and an App Service plan together

1. **Create a resource > Web + Mobile > Web App.**
2. App Service Plan → Create New → Select **Pricing tier**, and choose one of the **Isolated** pricing SKUs. If you choose an **Isolated** SKU and a location that's not an ASE, a **new ASE** is created in that location.
3. Specify your Azure virtual networking details. Select either **Create New** or **Select Existing**.
 - If you select **Create New**, enter a name for the VNet. A new Resource Manager VNet with that name is created. The new VNet has the address range 192.168.250.0/23 and a subnet named default. The subnet is defined as 192.168.250.0/24.
 - If you select **Select Existing**, you need to:
 - a) Select the VNet address block, if you have more than one.
 - b) Enter a new subnet name.
 - c) Select the size of the subnet.

Remember to select a size large enough to accommodate future growth of your ASE. We recommend /25, which has 128 addresses and can handle a maximum-sized ASE. We don't recommend /28, for example, because only 16 addresses are available. Infrastructure uses at least seven addresses and Azure Networking uses another 5. In a /28 subnet, you're left with a maximum scaling of 4 App Service plan instances for an External ASE and only 3 App Service plan instances for an ILB ASE.

- d) Select the subnet IP range.

4. Create (THIS WOULD TAKE AROUND 60 to 70 MINUTES)

Create and use an internal load balancer with an App Service Environment

You can deploy an ASE with an IP address in your VNet. To set the IP address to a VNet address, the ASE must be deployed with an ILB. When you deploy your ASE with an ILB, you must provide:

- Your own domain that you use when you create your apps.
- The certificate used for HTTPS.
- DNS management for your domain.

In return, you can do things such as:

- Host intranet applications securely in the cloud, which you access through a site-to-site or Azure ExpressRoute VPN.
- Host apps in the cloud that aren't listed in public DNS servers.
- Create internet-isolated back-end apps, which your front-end apps can securely integrate with.

Steps

1. **Create a resource > Web Mobile > App Service Environment.**
2. Enter the name of your ASE
3. Select your subscription. This subscription is also the one that all apps in the ASE use. You can't put your ASE in a VNet that's in another subscription.
4. Select or specify a new resource group. The resource group used for your ASE must be the same one that's used for your VNet. If you select an existing VNet, the resource group selection for your ASE is updated to reflect that of your VNet. *You can create an ASE with a resource group that is different from the VNet resource group if you use a Resource Manager template.*
5. Select your VNet and location. You can create a new VNet or select an existing VNet
6. The **VIP Type** selection determines if your ASE can be directly accessed from the internet (External) or if it uses an ILB.
 - If you select **External** for the **VIP Type**, you can select how many external IP addresses the system is created with for IP-based SSL purposes.
 - If you select **Internal** for the **VIP Type**, you must specify the domain that your ASE uses. You can deploy an ASE into a VNet that uses public or private address ranges. To use a VNet with a public address range, you need to create the VNet ahead of time.
7. Enter a domain name = contoso-internal.com

The custom domain name used for apps and the domain name used by your ASE can't overlap.

For example, you can use something like **contoso-internal.com** for the domain of your ASE so that won't conflict with custom domain names for app like **www.contoso.com**.

8. Create a Self Signed Certificate: **Open PowerShell in Administrator mode**

```
$certificate = New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname "*.internal-  
contoso.com", "*.scm.internal-contoso.com"  
$certThumbprint = "cert:\localMachine\my\" + $certificate.Thumbprint  
$password = ConvertTo-SecureString -String "CHANGETHISPASSWORD" -Force -AsPlainText  
$fileName = "exportedcert.pfx"  
Export-PfxCertificate -cert $certThumbprint -FilePath $fileName -Password $password
```

9. ILP ASE → Settings → ILB Certificate → Set ILB Certificate → Select the certificate .pfx file and enter the password
10. To ASE add Web App with App Service plan = <ILB ASE as created above>
11. Create a VM if you don't have one in that VNet. (Don't try to create this VM in the same subnet as the ASE because it will fail or cause problems.)
12. Get the ILB address for your ASE. Select **ASE → Properties → Virtual IP Address**.
13. RDP to VM → Edit Hosts file in folder C:\Windows\System32\drivers\etc\
14. Set *mytestapp.internal-contoso.com* to IP Address found above
15. Use a browser on the VM and go to <https://mytestapp.internal-contoso.com>.

Basic CRUD Operations using Entity Framework Core

1. Create a New ASP.NET Application

File → New Project → Visual C# → .NET Core → ASP.NET Core Web Application → OK

Ensure the options **.NET Core** and **ASP.NET Core** are selected in the drop down lists

2. Create Model classes

- Define a context and entity classes that make up your model.
- Right-click on the Models folder and select Add -> Class...

```
public class Department  
{  
    [Key]  
    public int Id { get; set; }  
    public string DeptName { get; set; }  
    public bool IsActive { get; set; }  
    public ICollection<Employee> Employees { get; set; }  
}  
public class Employee  
{
```

```

[Key]
public int Id { get; set; }
[ForeignKey("Department")]
public int FKDeptId { get; set; }
public string EmpName { get; set; }
public decimal Salary { get; set; }
public bool IsActive { get; set; }
public Department Department { get; set; }
}

```

Note: By default, EF Core interprets a property that's named **ID** or **classnameID** as the primary key.

[DatabaseGenerated(DatabaseGeneratedOption.None)]

The **DatabaseGenerated** attribute allows the app to specify the primary key rather than having the DB generate it.

3. Build the Project and ensure there are **no compilation errors**.
4. In **Solution Explorer**, right click on the *Controllers Folder* → **Add** → **New Scaffolded Item**. → **MVC Controller with views, using Entity Framework** → **Add**
5. Select the Model class = "Employee", Data context class: Click +, New data context type: **OrganizationContext** → **Add**
6. **Examine the code generated.**

Examine the context generated context class: **Data/DemoDbContext.cs**

```

public class DemoDbContext : DbContext
{
    public DemoDbContext(DbContextOptions<DemoDbContext> options)
        : base(options)
    {
    }

    public DbSet<DataAccessUsingEF.Models.Employee> Employee { get; set; }
    public DbSet<DataAccessUsingEF.Models.Department> Department { get; set; }
}

```

Examine the context registered with dependency injection:

Open **Startup.cs** and look at the **ConfigureService**

```
services.AddDbContext<DemoDbContext>(options =>
```

```
options.UseSqlServer(Configuration.GetConnectionString("DemoDbContext"));
```

Examine the appsettings.json file

```
"ConnectionStrings": {  
  "DemoDbContext":  
    "Server=(localdb)\\mssqllocaldb;Database=DemoDbContext;Trusted_Connection=True;MultipleActiveResultSets=true"  
}
```

7. Edit the Code in Main as below

```
public static void Main(string[] args)  
{  
  var host = CreateWebHostBuilder(args).Build();  
  using (var scope = host.Services.CreateScope())  
  {  
    var services = scope.ServiceProvider;  
    try  
    {  
      var context = services.GetRequiredService<DemoDbContext>();  
      context.Database.EnsureCreated();  
    }  
    catch (Exception ex)  
    {  
      var logger = services.GetRequiredService<ILogger<Program>>();  
      logger.LogError(ex, "An error occurred creating the DB.");  
    }  
  }  
  host.Run();  
}
```

- **EnsureCreated** ensures that the database for the context exists. If it exists, no action is taken. If it does not exist, then the database and all its schema are created.

- EnsureCreated does not use migrations to create the database. A database that is created with EnsureCreated cannot be later updated using migrations.
 - EnsureCreated is called on app start, which allows us with the following workflow:
 - Delete the DB.
 - Change the DB schema (for example, add an EmailAddress field).
 - Run the app.
 - EnsureCreated creates a DB with the EmailAddress column.
 - EnsureCreated is convenient early in development when the schema is rapidly evolving. Later in the tutorial the DB is deleted and migrations are used.
8. Build the application

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the DB are executed asynchronously. That includes, `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Sam")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the DB.

Add code to initialize the DB with test data

9. Add New class DbInitializer to the Project.

```
public static class DbInitializer
{
    public static void Initialize(DemoContext context)
    {
        // context.Database.EnsureCreated();
        if (!context.Department.Any())
        {
            var departments = new Department[]
            {
                new Department(){DeptName = "Development"},
            }
        }
    }
}
```

```
new Department(){DeptName = "Training"}
};
foreach (Department dept in departments)
    context.Department.Add(dept);
context.SaveChanges();
}
// Look for any students.
if (!context.Employee.Any())
{
    var employees = new Employee[]
    {
        new Employee(){EmpName="Sandeep", FKDeptId=1, Salary=100000, IsActive=true},
        new Employee(){EmpName="Rahul",FKDeptId=2, Salary=100000, IsActive=true},
        new Employee(){EmpName="Naveen",FKDeptId=1, Salary=50000, IsActive=true}
    };
    foreach (Employee emp in employees)
        context.Employee.Add(emp);
    context.SaveChanges();
}
}
```

10. Edit Main.cs and add the following line after EnsureCreated call

```
context.Database.EnsureCreated();
DbInitializer.Initialize(context);
```

Migration Tools

For early development, **EnsureCreated** was used. In this tutorial, migrations are used. It has the following limitations:

- Bypasses migrations and creates the DB and schema.
- Doesn't create a migrations table.
- Cannot be used with migrations.

- Is designed for testing or rapid prototyping where the DB is dropped and re-created frequently.

11. Remove the following line from Main

```
//context.Database.EnsureCreated();
```

The following steps use **migrations** to create a database.

12. Package Manager Console → Drop Database

13. Install NuGet Package **Microsoft.EntityFrameworkCore.Tools**

14. Create the Initial Migration and Update the database.

Open Tools → NuGet Package Manager → **Package Manager Console**

- Run **Add-Migration InitialCreate** to scaffold a migration to create the initial set of tables for your model.

If you receive an error

- Run **Update-Database** to apply the new migration to the database. This command creates the database before applying migrations.

Examine the Up and Down methods:

Note that Migrations calls the **Up** method to implement the data model changes for a migration. When you enter a command to roll back the update, migrations calls the **Down** method.

15. Run the app and verify the DB is seeded.

context.Database.Migrate()

Applies any pending migrations for the context to the database. Will create the database if it does not already exist.

Applying migrations in production

- We recommend production apps should **not** call **Database.Migrate** at application startup.
- **Migrate** shouldn't be called from an app in server farm. For example, if the app has been cloud deployed with scale-out (multiple instances of the app are running).
- Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:
 - Using migrations to create SQL scripts and using the SQL scripts in deployment.

- Running `dotnet ef database update` from a controlled environment.
- EF Core uses the `__MigrationsHistory` table to see if any migrations need to run. If the DB is up-to-date, no migration is run.

DECCANSOFT