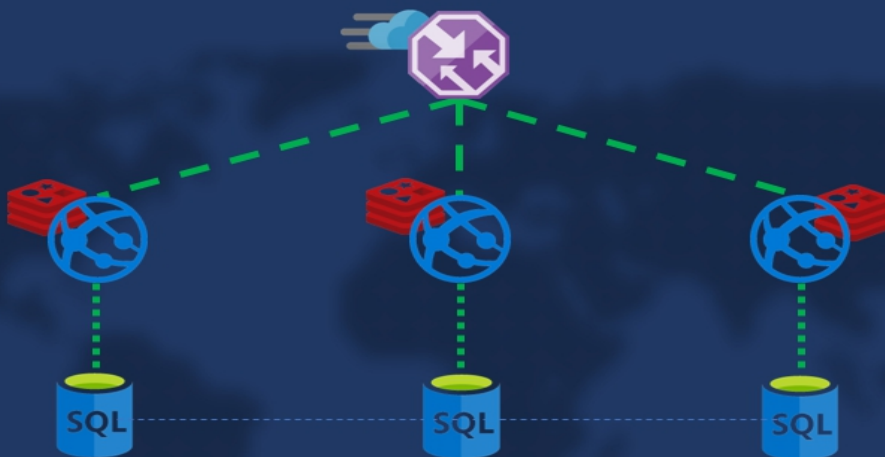


Globally-distributed applications with Microsoft Azure



Build robust, highly available, planet-scale web applications using Microsoft Azure Services. Apply DevOps for business continuity to a data geo-replicated complex system

Globally-Distributed Applications with Microsoft Azure

For developers and architects

Christos Sakellarios

This book is for sale at

<http://leanpub.com/globally-distributed-applications-with-microsoft-azure>

This version was published on 2018-05-15



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 Christos Sakellarios

Contents

WebJobs	1
Primary Resource Group	10

WebJobs

Overview

Azure App Services have a feature named WebJobs which allow you to run background tasks in the same context as a web app, API app, or mobile app. A WebJob can be an executable or a script with one of the following file types:

- .cmd, .bat, .exe (using Windows cmd)
- .ps1 (using PowerShell)
- .sh (using Bash)
- .php (using PHP)
- .py (using Python)
- .js (using Node.js)
- .jar (using Java)

To add a WebJob in an App Service, you simply upload a **zip** file containing your executable or script file to your App Service. There are two different types for WebJobs, *continuous* and *triggered*:

Continuous	Triggered
Starts immediately when the WebJob is created. To keep the job from ending, the program or script typically does its work inside an endless loop. If the job does end, you can restart it.	Starts only when triggered manually or on a schedule.
Runs on all instances that the web app runs on. You can optionally restrict the WebJob to a single instance.	Runs on a single instance that Azure selects for load balancing.
Supports remote debugging.	Doesn't support remote debugging.

You can also monitor your WebJobs status in the relevant blade of your App Service. Any logs

produced by your executable or scripts, appear on the selected WebJob screen. `Online.Store` application uses `WebJobs` to listen for order messages from the `Service Bus` deployed in the same region with the `App Service`.

Local configuration

The `WebJob` library project for submitting orders in database is the `Online.Store.WebJob`. It's a simple .NET Core console application project that uses the [Microsoft.Azure.ServiceBus](#) NuGet package to listen messages on the `Service Bus` *orders* queue. The reason you see an **`appsettings.json.template`** file in this project and not the usual `appsettings.json` is that the latter shouldn't be tracked by the source control. In contrast with the `Online.Store` web project you cannot use a `secrets.json` file or use the **application settings** stored in the `App Service`'s settings (*this is where you save secrets in Azure Portal, more on this on the next section*). Add a new file named **`appsettings.json`** in the `Online.Store.WebJob` project and copy the contents from the *`appsettings.json.template`*.

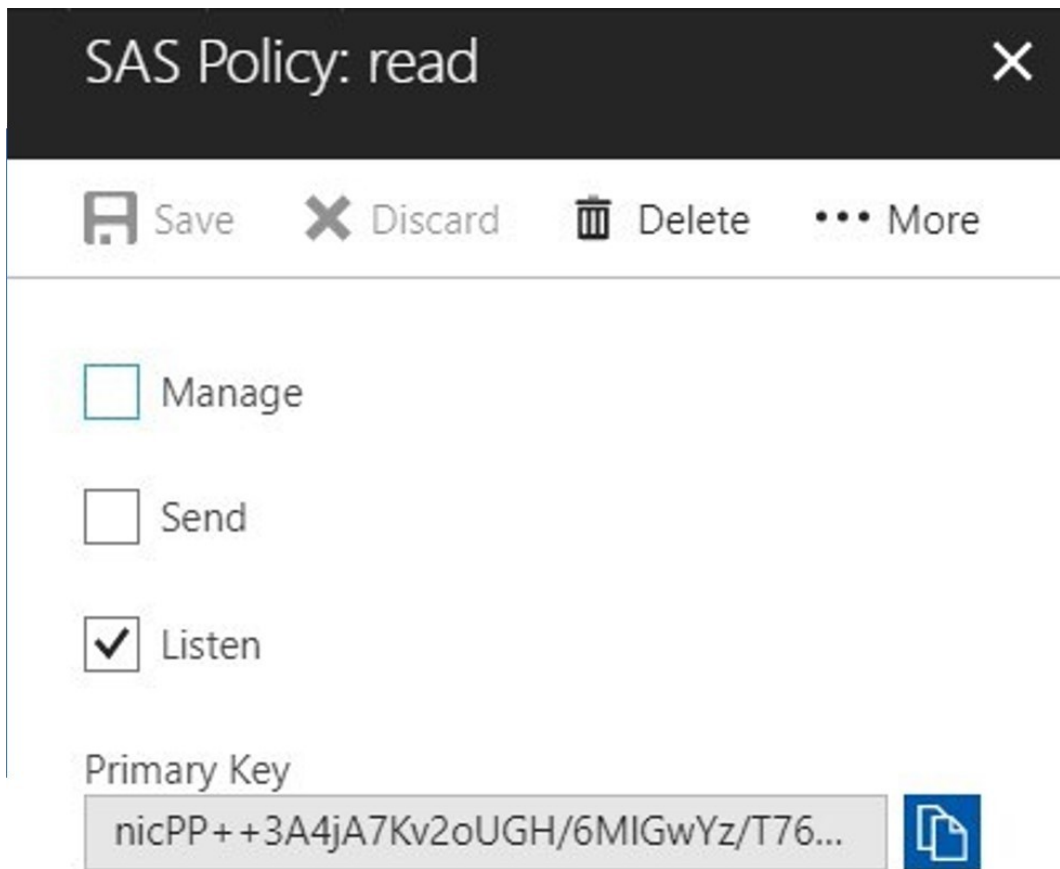
```
{
  "ConnectionStrings": {
    "DefaultConnection": ""
  },
  "ServiceBus:Namespace": "",
  "ServiceBus:Queue": "orders",
  "ServiceBus:ReadAccessKeyName": "Read",
  "ServiceBus:ReadAccessKey": ""
}
```

Set the above configuration properties as follow:



WebJob appsettings.json configuration

1. **ConnectionStrings:DefaultConnection:** Same connection string you used in the **secrets.json** file for `Online.Store` web application. It's the connection string to the database you created
2. **ServiceBus:Namespace:** `<parent-resource-group>-servicebus`. It's the same value you used for the same property in the **appsettings.json** file for `Online.Store` web application.
3. **ServiceBus:ReadAccessKey:** In [Azure Portal](#) open the *orders* queue in the Service Bus namespace you created in **Performance Optimization** part. Select and view the Shared access policies for the queue. Recall that you have created two access policies, one named `Write` for which you used its Primary Key in the `Online.Store` application to send messages to the *orders* queue. Now you need to get the Primary Key for the **Read** access policy in order to read messages from the queue. Click in the `Read` policy and copy the **Primary Key**. Set it as the value for this property



Service Bus Read access policy Primary Key

Your webjob's appsettings.json file should look like the following (*e.g. for planetscale-store*):

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:planetscalestore-westeuropa-app-sql-a.database..\"
  },
  "ServiceBus:Namespace": "planetscalestore-westeuropa-servicebus",
  "ServiceBus:Queue": "orders",
  "ServiceBus:ReadAccessKeyName": "Read",
  "ServiceBus:ReadAccessKey": "N0Lxa5YkHiHF3eXAEqDu2n8hMHMCckoUPi7ntxvND0k="
}
```

The console application has a simple `OrderService` class to save orders in the database. It uses the `ApplicationDbContext` **Entity Framework** context class referenced from the `Online.Store.SqlServer` project.

```
public class OrderService : IOrderService
{
    ApplicationDbContext _context;

    public OrderService(ApplicationDbContext context)
    {
        _context = context;
    }
    public async Task AddOrderAsync(Order order)
    {
        _context.Orders.Add(order);
        await _context.SaveChangesAsync();
    }
}
```

In `Program.cs` an instance of `QueueClient` is used to register a **message handler** for receiving Service Bus queue messages.


```
static void StartReceivingOrders(string queue)
{
    queueClient = new QueueClient(_serviceBusConnString, queue);

    // Configure the MessageHandler Options in terms of exception handling,
    // number of concurrent messages to deliver etc.
    var messageHandlerOptions
        = new MessageHandlerOptions(ExceptionReceivedHandler)
    {
        // Maximum number of Concurrent calls to the callback
        // `ProcessMessagesAsync`, set to 1 for simplicity. Set it according
        // to how many messages the application wants to process in parallel.
        MaxConcurrentCalls = 1,

        // Indicates whether MessagePump should automatically complete
        // the messages after returning from User Callback.
        // False below indicates the Complete will be handled by the
        // User Callback as in `ProcessMessagesAsync` below.
        AutoComplete = false
    };

    // Register the function that will process orders
    queueClient.RegisterMessageHandler(ProcessOrderAsync, messageHandlerOptions);
}
```

The processing method **deserializes** the message from the queue, saves it in the database and finally removes the message from the queue.

```

static async Task ProcessOrderAsync(Message message, CancellationToken token)
{
    // Process the message
    Console.WriteLine($"Received message: SequenceNumber:{message.SystemProperties.

    Order order = JsonConvert.DeserializeObject<Order>(Encoding.UTF8.GetString(me:
    order {Online.Store.Core.DTOs.Order}
    await _orderService.AddOrderAsync(order);
    Console.WriteLine($"Order added successfully");
    // Complete the message so that it is not received again.
    // This can be done only if the queueClient is created in ReceiveMode.PeekLock
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);

    // Note: Use the cancellationToken passed as necessary to determine if the que
    // If queueClient has already been Closed, you may chose to not call Complete
    // to avoid unnecessary exceptions.
}

```

WebJob - Receive messages

```

static async Task ProcessOrderAsync(Message message, CancellationToken token)
{
    // Process the message
    Order order = JsonConvert
        .DeserializeObject<Order>(Encoding.UTF8.GetString(message.Body));

    await _orderService.AddOrderAsync(order);

    // Complete the message so that it is not received again.
    // This can be done only if the queueClient is created in
    // ReceiveMode.PeekLock mode (which is default).
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);
}

```



Run Online.Store.WebJob console app using one of the following methods:



1. Visual Studio
 - Right click Online.Store.WebJob application
 - Select Debug -> Start new instance
2. .NET Core CLI
 - Open a terminal and cd to Online.Store.WebJob application's folder
 - Run `dotnet run .\Online.Store.WebJob.csproj`

It should receive all the order messages exist in the Service Bus queue and process them. When a message is received the app logs related information. This information will be useful when you will deploy the web job up in Azure.

```
Windows PowerShell
Online.Store.WebJob> dotnet run .\Online.Store.WebJob.csproj
===== START RECEIVING ORDERS =====
Received message: SequenceNumber:13792273858822145 Body:{"Id":0,"UserId":"af210
ated":"2018-02-13T16:41:32.7014Z","GrandTotal":789.98,"OrderDetails":[{"Id":0,"
93d5-ac02-4861-bc36-5bafb439610c","ProductTitle":"Sony - Cyber-shot DSC-HX80 1
l":"DSC-HX80","ProductPrice":339.99,"Quantity":1},{"Id":0,"OrderId":0,"Order":
6e7528d55dd4","ProductTitle":"Panasonic - LUMIX DC-ZS70 20.3-Megapixel Digital
,"ProductPrice":449.99,"Quantity":1}}}
ORDER: 1 has been submitted successfully
Received message: SequenceNumber:54324670505156609 Body:{"Id":0,"UserId":"af210
8-02-13T16:56:52.2604796Z","GrandTotal":90.99,"OrderDetails":[{"Id":0,"OrderId
5-6ae5d961ea35","ProductTitle":"Polaroid - Snap 10.0-Megapixel Digital Camera
.99,"Quantity":1}}}
ORDER: 2 has been submitted successfully
```

Web Job - Log messages

If all the configuration have been set properly, you should see the orders in the Online.Store interface as well.

2/13/2018, 6:41 PM		Price	Quantity	Total
	Sony - Cyber-shot DSC-HX80 18.2-Megapixel Digital Camera	\$339.99	1	\$339.99
	Panasonic - LUMIX DC-ZS70 20.3-Megapixel Digital Camera - Black	\$449.99	1	\$449.99
Grand Total				\$789.98
Served by Local				

Web Job - Receive messages

Primary Resource Group

The primary resource group is the base resource group for Online.Store application. It contains the resources that are common for all *sub-resource groups* and once it is created, it shouldn't change often. The 4 types of resources contained in the primary resource group are:

- Storage Account
- CDN Profile
- Azure Cosmos DB account
- Traffic Manager Profile
- Identity SQL Server and Database (*optional*)

The primary resource group has a **unique** name and this name will be the base for constructing all other resource groups and resources type names (*based on region, type, version..*). This will help you locate and monitor resources more easily and of course build more generic PowerShell scripts. The property used for the primary resource group name in the PowerShell scripts that follow, is `PrimaryName`. But why `PrimaryName` should be unique? Most of the resources have **endpoints** which must be globally unique. Take a look how service endpoints look like for the following types in the primary resource group:

- **Storage Account:** `https://<primary-name>storage.blob.core.windows.net/`
- **CDN Profile:** `https://<primary-name>-cdn.azureedge.net`
- **Azure Cosmos DB:** `https://<primary-name>-cosmosdb.documents.azure.com:443/`
- **Traffic Manager Profile:** `http://<primary-name>.trafficmanager.net`

The prefixes you have added will certainly reduce the possibility for conflicts. During the examples of this book, **planetscalestore** has been used as the primary name. Each time you see the word `planetscalestore` in the scripts that follow, make sure to replace it with your own primary name.

Init Primary Resources Script

Ideally, you would like to run a custom script to initialize the primary resource group and all of its resources. This is what the **init-primary-resources.ps1** PowerShell script is for. You will find the script and all others inside the `Online.Store/App_Data/devops` folder. The script accepts 5 parameters of which the last 3 are optional:

1. **PrimaryName:** The primary name to be used
2. **ResourceGroupLocation:** Azure region for the resource group and its resources
3. **CreateIdentityDatabase:** Create or not the SQL Server & database required for ASP.NET Identity (*optional*). If parameter is used, possible values are `$true`, `$false`
4. **SqlServerLogin:** SQL Server login for the primary SQL Server created in case `CreateIdentityDatabase` is `True` (*optional*)
5. **SqlServerPassword:** SQL Server password for the primary SQL Server created in case `CreateIdentityDatabase` is `True` (*optional*)



PowerShell optional parameters

Optional parameters can be omitted when running a PowerShell script

To list all available resource group location values run the following command in PowerShell.

```
Get-AzureRmLocation | select Location
```

Available Locations for Resource Groups

eastasia	southeastasia	centralus	eastus
eastus2	westus	northcentralus	southcentralus
northeurope	westeurope	japanwest	japaneast
brazilsouth	australiaeast	australiasoutheast	southindia
centralindia	westindia	canadacentral	canadaeast
uksouth	ukwest	westcentralus	westus2
koreacentral	koreasouth		



About locations

This book has followed a naming convention where parent and child resource groups belong to the same region. The thing is that some times not all type of resources are available to a specific region. For example, assuming that you want to provision the primary resource group and its resources to West Europe but **at that time** Azure Cosmos DB may not be available **for that region**. If you run the `init-primary-resources` script, it will fail when try to create the Cosmos DB account.

You could catch some of those errors before starting the provisioning process by checking the [Products available by region](#) page. The following regions have been tested many times during the development of `Online.Store` app and are highly recommended

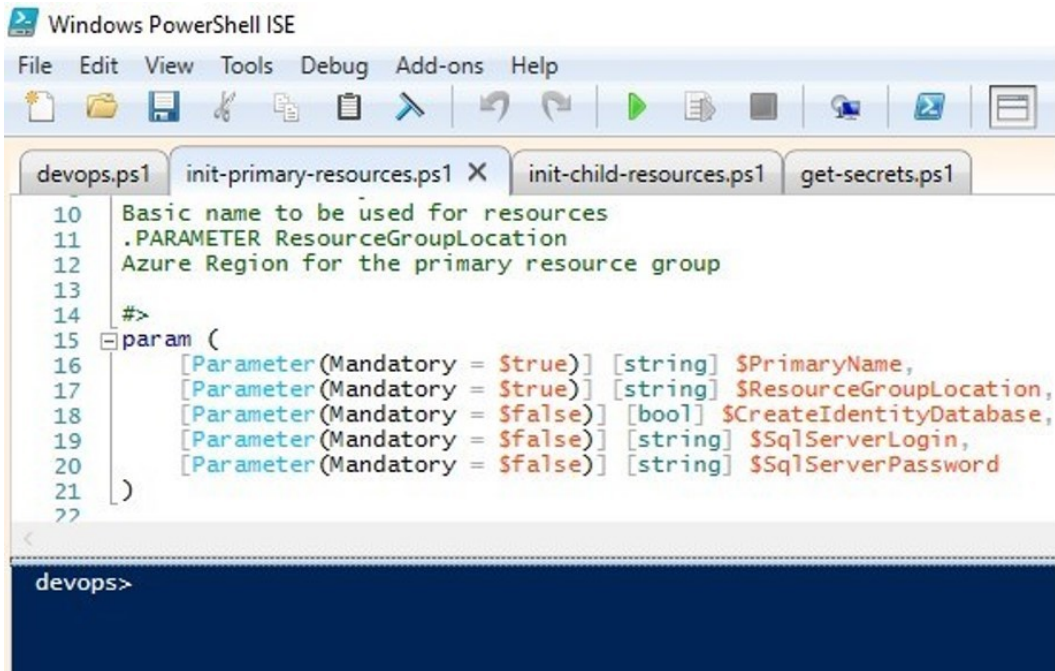
1. westcentralus
2. southcentralus
3. eastus
4. westeurope
5. southeastasia

Open the `init-primary-resources.ps1` script to examine it.



Opening PowerShell scripts

A very usefull tool for viewing and running PowerShell scripts is the [Windows Powershell ISE](#). Most of the times you will find it installed along with PowerShell in Windows OS. On this Integrated Scripting Environment you can have the script opened in one editor and viewing its results in the integrated command line. More over you get a powerfull intellisense experience.



The screenshot shows the Windows PowerShell ISE interface. The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains icons for opening files, saving, undo, redo, running the script, and other standard IDE functions. There are four tabs open: devops.ps1, init-primary-resources.ps1 (which is the active tab), init-child-resources.ps1, and get-secrets.ps1. The active script contains the following PowerShell code:

```

10 Basic name to be used for resources
11 .PARAMETER ResourceGroupLocation
12 Azure Region for the primary resource group
13
14 #>
15 param (
16     [Parameter(Mandatory = $true)] [string] $PrimaryName,
17     [Parameter(Mandatory = $true)] [string] $ResourceGroupLocation,
18     [Parameter(Mandatory = $false)] [bool] $CreateIdentityDatabase,
19     [Parameter(Mandatory = $false)] [string] $SqlServerLogin,
20     [Parameter(Mandatory = $false)] [string] $SqlServerPassword
21 )
22
devops>

```

Windows Powershell ISE

Create Primary Resource Group

```
Get-AzureRmResourceGroup -Name $PrimaryName -ev notPresent -ea 0
```

```

if ($notPresent)
{
    # ResourceGroup doesn't exist
    Write-Host "Trying to create Resource Group: $PrimaryName "
    New-AzureRmResourceGroup -Name $PrimaryName -Location $ResourceGroupLocation
}
else
{
    # ResourceGroup exist
    Write-Host "Resource Group: $PrimaryName already exists.."
}

```

`Get-AzureRmResourceGroup` cmdlet is used to get the resource group named `PrimaryName` in your account. If not exists `New-AzureRmResourceGroup` is used to create one at the location

you provided. Notice that all your scripts should be idempotent meaning that you can run the same script multiple times but it will only make the changes once. In other words, if the resource group already exists it will not try to create it again. The same logic must be followed for every Azure Resource provisioning.

Create Storage Account

The storage account hosts the images for `Online.Store` application. Any containers required (such as *product-images* if you recall) are created at application startup.

```
$storageAccountName = "$PrimaryName" + "$storagePrefix";

Get-AzureRmStorageAccount -ResourceGroupName $PrimaryName `
    -Name $storageAccountName -ev storageNotPresent -ea 0

if ($storageNotPresent)
{
    Write-Host "Creating Storage Account $storageAccountName"
    $skuName = "Standard_GRS"

    # Create the storage account.
    $storageAccount = New-AzureRmStorageAccount -ResourceGroupName $PrimaryName `
        -Name $storageAccountName `
        -Location $ResourceGroupLocation `
        -SkuName $skuName

    Write-Host "Storage Account $storageAccountName successfully created.."
}
else
{
    Write-Host "Storage Account $storageAccountName already exists.."
}
```

`Get-AzureRmStorageAccount` is used to retrieve the storage account named `PrimaryName` in your primary resource group. If not exists `New-AzureRmStorageAccount` cmdlet is used to create one.



SKU name

Specifies the SKU name of the storage account that this cmdlet creates. The acceptable values for this parameter are:

- **Standard_LRS**: Locally-redundant storage
- **Standard_ZRS**: Zone-redundant storage
- **Standard_GRS**: Geo-redundant storage
- **Standard_RAGRS**: Read access geo-redundant storage
- **Premium_LRS**: Premium locally-redundant storage

Create CDN Profile

Following the storage account the script creates the CDN Profile *bound* to that storage account. Not only it provisions the CDN Profile but also configures it to cache the **blobs** on the storage account created before. This is how images in *Online.Store* application will be served by POPs of your CDN Profile.

```
$cdnProfileName = "$PrimaryName-$cdnPrefix";

Get-AzureRmCdnProfile -ProfileName $cdnProfileName `
    -ResourceGroupName $PrimaryName -ev cdnNotPresent -ea 0
if ($cdnNotPresent)
{
    Write-Host "Creating CDN profile $cdnProfileName.."
    # Create a new profile
    New-AzureRmCdnProfile -ProfileName $cdnProfileName `
        -ResourceGroupName $PrimaryName `
        -Sku Standard_Verizon -Location $ResourceGroupLocation

    Write-Host "CDN profile $cdnProfileName succesfully created.."

    # Create a new endpoint
    $cdnEndpointName = "$PrimaryName-$endpointPrefix";
    $endpointHost = "$storageAccountName.blob.core.windows.net"

    $availability =
        Get-AzureRmCdnEndpointNameAvailability -EndpointName $cdnEndpointName
```

```

if($availability.NameAvailable) {
    Write-Host "Creating endpoint..."

    New-AzureRmCdnEndpoint -ProfileName $cdnProfileName `
        -ResourceGroupName $PrimaryName `
        -Location $ResourceGroupLocation -EndpointName $cdnEndpointName `
        -OriginName "$storageAccountName" `
        -OriginHostName $endpointHost `
        -OriginHostHeader $endpointHost
    }
}
else
{
    Write-Host "CDN profile $cdnProfileName already exists.."
}

```

`Get-AzureRmCdnProfile` cmd checks if the profile already exists. In not `New-AzureRmCdnProfile` is used to create one. Next you need to add a CDN endpoint that points to the Blob Service Endpoint of your storage account. The URI of this endpoint is <primary-name>storage.blob.core.windows. Before adding this endpoint using `New-AzureRmCdnEndpoint`, it checks endpoint's availability using the `Get-AzureRmCdnEndpointNameAvailability` cmdlet.



CDN Endpoint & Storage Account

You need to set `OriginHostHeader` equal to `OriginHostName` in the `New-AzureRmCdnEndpoint` cmdlet for storage account endpoints

Create Cosmos DB account

The script continues with creating the Azure Cosmos DB account. The database account name matches the `DocumentDB:DatabaseId` property in the `appsettings.json` file.

```

$documentDbDatabase = "$PrimaryName-$cosmosDbPrefix";

$query = Find-AzureRmResource -ResourceNameContains $documentDbDatabase `
    -ResourceType "Microsoft.DocumentDb/databaseAccounts"

if (!$query)
{
    Write-Host "Creating DocumentDB account $documentDbDatabase.."
    # Create the account

    # Write and read locations and priorities for the database
    $locations = @(@{"locationName"= $ResourceGroupLocation;
        "failoverPriority"=0})

    # Consistency policy
    $consistencyPolicy = @{"defaultConsistencyLevel"="Session";}

    # DB properties
    # https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels
    $DBProperties = @{"databaseAccountOfferType"="Standard";
        "locations"=$locations;
        "consistencyPolicy"=$consistencyPolicy}

    # Create the database
    New-AzureRmResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" `
        -ApiVersion "2015-04-08" `
        -ResourceGroupName $PrimaryName `
        -Location $ResourceGroupLocation `
        -Name $documentDbDatabase `
        -PropertyObject $DBProperties

    Write-Host "DocumentDB account $documentDbDatabase succesfully created.."
}
else
{
    Write-Host "DocumentDB account $documentDbDatabase already exists.."
}

```

The script searches for a resource of type `Microsoft.DocumentDb/databaseAccounts` using the `Find-AzureRmResource` cmdlet. You can use this cmdlet to search for many resource types in Azure. If not found, it creates the Cosmos DB database account using the `New-AzureRmResource` cmdlet. The `$locations` property determines if you wish to replicate data

in other regions and their failover priorities. You will replicate the account's data in a different region through the portal where you can also visually see the available locations at that time.

Create Traffic Manager profile

Last but not least, the script creates the Traffic Manager profile. The Traffic Manager profile will be used to control the distribution of traffic to your Azure website endpoints.

```
$tmpProfileName = "$PrimaryName";
$tmpDnsName = "$PrimaryName";

Get-AzureRmTrafficManagerProfile -Name $tmpProfileName `
  -ResourceGroupName $PrimaryName -ev tmpNotPresent -ea 0
if($tmpNotPresent) {
    Write-Host "Creating Traffic Manager Profile $tmpProfileName.."

    New-AzureRmTrafficManagerProfile -Name $tmpProfileName `
      -ResourceGroupName $PrimaryName -TrafficRoutingMethod Performance `
      -RelativeDnsName $tmpDnsName -Ttl 30 -MonitorProtocol HTTP `
      -MonitorPort 80 -MonitorPath "/"

    Write-Host "Traffic Manager Profile created successfully.."
}
else {
    Write-Host "Traffic Manager $tmpProfileName already exists.."
}
```

The script checks if the traffic manager profile exists using the [Get-AzureRmTrafficManagerProfile](#) cmdlet and if not, it creates it using [New-AzureRmTrafficManagerProfile](#). At this point you don't have any App Services to add. The App Service endpoints are added to the Traffic Manager profile at the time being provisioned as disabled (*more on this in Child Resources section*).

Create SQL Server and Database for ASP.NET Identity (*optional*)

If you pass the parameter `CreateIdentityDatabase` as `$true`, the script will create a logical SQL Server named `<PrimaryName>-sql`.

```

$serverName = "$PrimaryName-$sqlServerPrefix";
$resourceGroupName = $PrimaryName;

$serverInstance = Get-AzureRmSqlServer -ServerName $serverName `
                                     -ResourceGroupName $resourceGroupName `
                                     -ErrorAction SilentlyContinue

if ($serverInstance) {
    Write-Host "SQL Server $serverName already exists..."
}
else {
    Write-Host "Trying to create SQL Server $serverName.."

    New-AzureRmSqlServer -ResourceGroupName $resourceGroupName `
        -ServerName $serverName `
        -Location $ResourceGroupLocation `
        -SqlAdministratorCredentials $(New-Object `
            -TypeName System.Management.Automation.PSCredential `
            -ArgumentList $SqlServerLogin, $(ConvertTo-SecureString `
            -String $SqlServerPassword -AsPlainText -Force))

    Write-Host "SQL Server $serverName successfully created..."

    # Allow access to Azure Services
    Write-Host "Allowing access to Azure Services..."

    New-AzureSqlDatabaseServerFirewallRule -ServerName $serverName `
        -AllowAllAzureServices
}

```

The script checks if the SQL Server exists using the `Get-AzureRmSqlServer` cmdlet and if not it creates it using `New-AzureRmSqlServer`. At the end adds a new **Firewall Rule** so that Azure Services such as App Services can access the server. In case `CreateIdentityDatabase` is `$true` the script will continue by checking the `identitydb` database.

```

$Database = "identitydb";

if($CreateIdentityDatabase) {
    $azureDatabase = Get-AzureRmSqlDatabase `
        -ResourceGroupName $resourceGroupName `
        -ServerName $serverName -DatabaseName $Database `
        -ErrorAction SilentlyContinue

    if ($azureDatabase) {
        Write-Host "Azure SQL Database $Database already exists..."
    }
    else {
        Write-Host "Creating SQL Database $Database at Server $serverName.."

        New-AzureRmSqlDatabase -ResourceGroupName $resourceGroupName `
            -ServerName $serverName `
            -DatabaseName $Database `
            -RequestedServiceObjectiveName "Basic" `
            -MaxSizeBytes 524288000

        Write-Host "Azure SQL Database $Database successfully created..."
    }
}

```

First it checks if identitydb database exists on the previous created server using the `Get-AzureRmSqlDatabase` cmdlet and if not, it creates it using `New-AzureRmSqlDatabase`.

Running the script

While in PowerShell make sure that your working directory is the `Online.Store/App_Data/devops` folder where all the scripts exist.



Create a devops script

To make it easier for you running the scripts in the `Online.Store/App_Data/devops` folder, create a PowerShell script named **devops.ps1** inside that folder. Don't worry, this file will not be tracked by Git (*check .gitignore*). There you can write all the calls to your scripts you want to run for the `Online.Store` application. Check the **devops-template.ps1** file for reference to understand how your **devops** file should like at the end

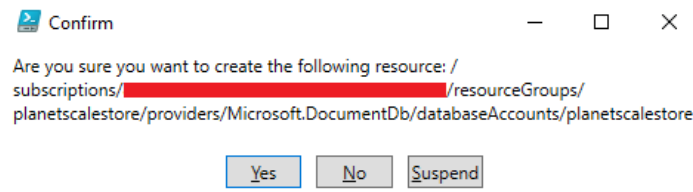
Run the `init-primary-resources.ps1` script as follow:

```
.\init-primary-resources.ps1 -PrimaryName "<primary-name>" `
                             -ResourceGroupLocation "<region>"
```

Make sure to use your primary name and the region you want the resources to provision to. For primary name `planetscalestore` and region `westeurope` the call would look like this:

```
.\init-primary-resources.ps1 -PrimaryName "planetscalestore" `
                             -ResourceGroupLocation "westeurope"
```

During the creation of the Cosmos DB account you will be asked to confirm the creation. Click Yes to continue.



Create Azure Cosmos DB account confirmation



Creating Azure Cosmos DB for the first time

The **first time** you create an Azure Cosmos DB in a subscription using the Azure portal, the portal registers the `Microsoft.DocumentDB` namespace for that subscription. If you attempt to create the first Cosmos DB account in a subscription using PowerShell, you must first register the namespace using the following command:








```
Register-AzureRmResourceProvider -ProviderNamespace "Microsoft.DocumentDB"
```

otherwise the script will fail to create the account. If you have followed along with the book, you have already created at least one Azure Cosmos DB account so you don't need to worry about that

At the end of each script you will here a beep sound indicating that the script has been executed. In case you want to use ASP.NET Identity for authentication in `Online.Store` application run the script as follow:


```
.\init-primary-resources.ps1 -PrimaryName "<primary-name>" `
                             -ResourceGroupLocation "<region>" `
                             -CreateIdentityDatabase $true `
                             -SqlServerLogin "<sql-server1-login>" `
                             -SqlServerPassword "<sql-server-password>"
```

Go back in the portal and confirm that all resources have been provisioned properly.

NAME ↑↓	TYPE ↑↓	LOCATION ↑↓
 planetscalestore	Traffic Manager profile	global
 planetscalestore-cdn	CDN profile	West Europe
 planetscalestore-endpoint	Endpoint	West Europe
 planetscalestore-cosmosdb	Azure Cosmos DB account	West Europe
 planetscalestore-sql	SQL server	West Europe
 identitydb	SQL database	West Europe
 planetscalestorestorage	Storage account	West Europe

Primary Resource Group

The script will create the SQL Server plus the identitydb database. Don't forget to run the **identity-migrations.sql** SQL script to update the schema in the identitydb database. This database will be a stand-alone database used by all regions for authenticating users.