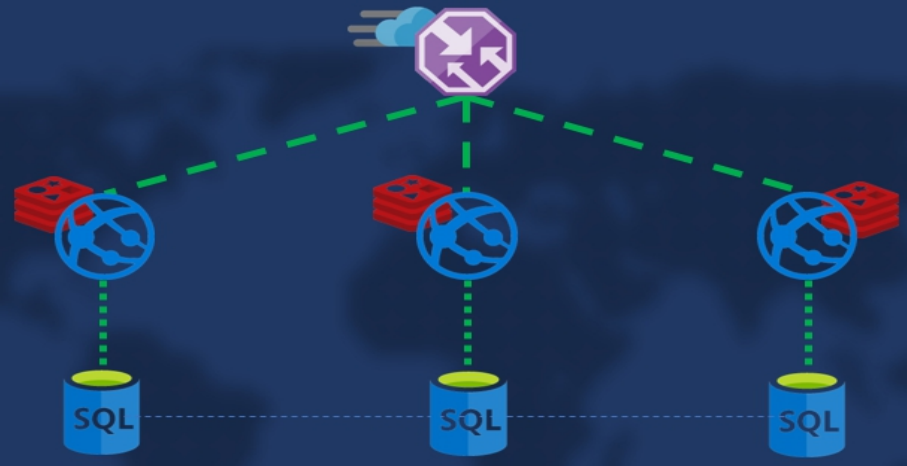
CHRISTOS SAKELLARIOS

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

chsakell's blog

Globally-Distributed Applications with Microsoft Azure

For developers and architects

Christos Sakellarios

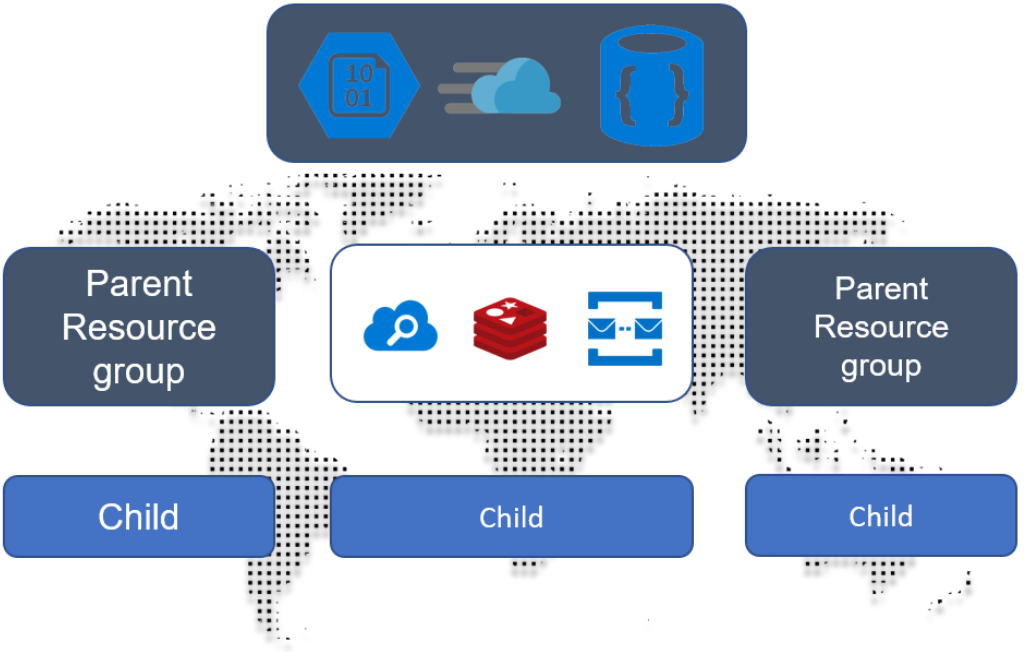
This book is for sale at
http://leanpub.com/globally-distributed-applications-with-microsoft-azure

This version was published on 2019-04-04

Contents

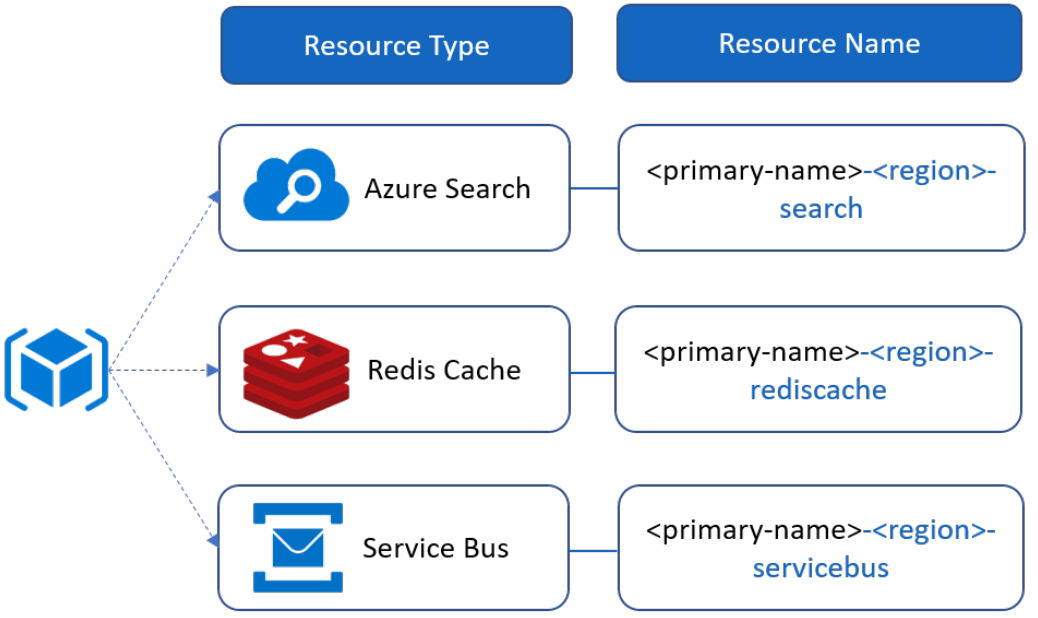
Performance Optimization

In the following sections you are going to create your first `Parent Resource Group` in `West Europe`. A `parent resource group` contains the `Azure Search`, `Redis Cache` and `Service Bus` services for a specific Azure region and acts as the second level in the resource groups hierarchy.



Parent Resource Groups

Before creating and start adding resources in the parent resource group, recall the naming convention you should follow.



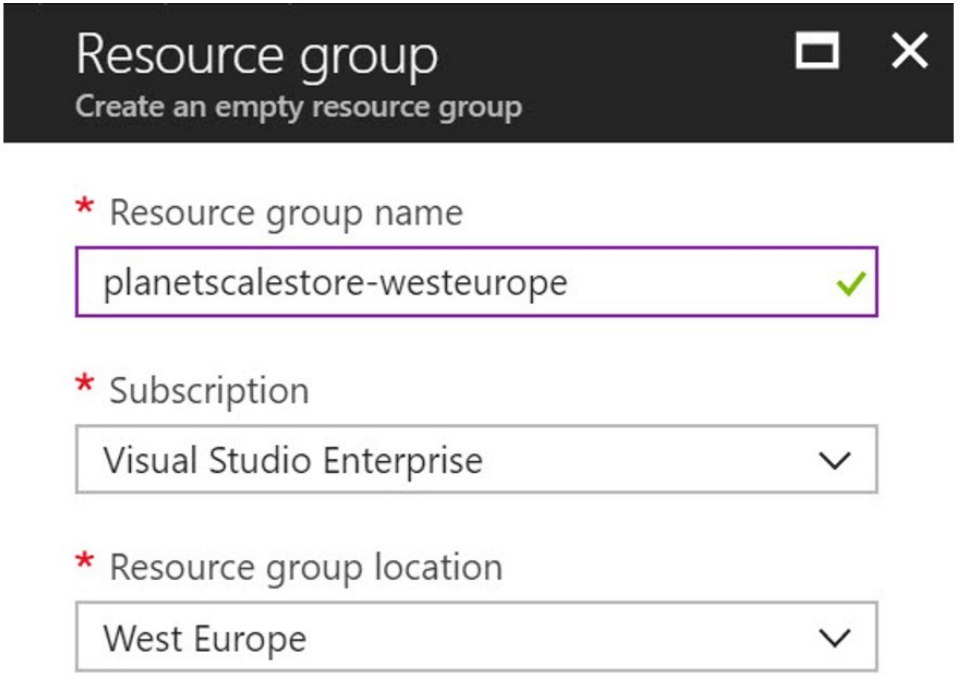
Parent Resource Group - Resources Names

Create Parent Resource Group

In Azure Portal click the `Resource groups` menu item on the left panel and next click the `Add` button. Fill the form as follow:

1. **Resource group name**: <Primary-Name>-westeurope
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group location**: West Europe

Since this is a parent resource group, you name it by the name you selected for primary resource group plus '-' and the region with no spaces and lowercase.



Create Parent Resource Group

Code Overview

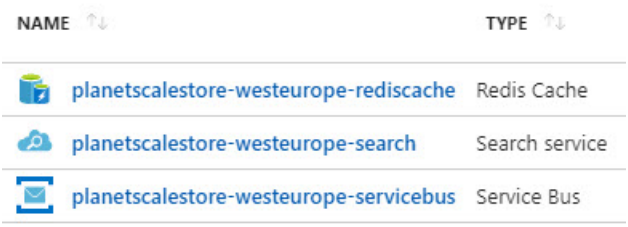
At this point you should have deployed the following Azure Resources:

- Azure Storage Account *(previous part)*
- CDN Profile *(previous part)*
- Azure Cosmos DB Account *(previous part)*
- Azure Search Service
- Redis Cache
- Service Bus

Before running again the `Online.Store` application verify that during this part, you have set the following properties in the configuration files:

- **appsettings.json**
  1. `SearchService:Name`: <parent-resource-group>-search
  2. `RedisCache:Endpoint`: <parent-resource-group>-rediscache
  3. `ServiceBus:Namespace`: <parent-resource-group>-servicebus
  4. `ServiceBus:Queue`: "orders"
- **secrets.json**
  1. `SearchService:ApiKey`: <search-service-primary-key>
  2. `RedisCache:Key`: <redis-cache-primary-key>
  3. `ServiceBus:WriteAccessKeyName`: "Write"
  4. `ServiceBus:WriteAccessKey`: <write-access-policy-key>

In the example images you have seen so far, `planetscalestore` has been used as the Primary Resource Group name. Here is how the resources looks like in the `parent resource group` created in `West Europe`.



Parent Resource Group Resources

Now that you have set the `Search Service` when you run `Online.Store` it will automatically **bind** the search engine with your **Cosmos DB** account. More specifically the app will create an index named *product-index* with a data-source pointing the `Items` collection where the product documents exist. The library project for `Search Service` is the `Online.Store.AzureSearch`. You can access and manage an Azure `Search Service` using the Microsoft.Azure.Search NuGet package. The `ISearchRepository` interface has 3 methods:

```
public interface ISearchRepository
{
    Task CreateOrUpdateDocumentDbDataSourceAsync(string dataSourceName,
                                        string dataSourceColletion);

    Task CreateOrUpdateIndexAsync<T>(string indexName, string suggesterName,
                                          List<string> suggesterFields);

    Task<List<T>> SearchAsync<T>(string searchIndex, string term,
                      List<string> returnFields) where T : class;
}
```

`SearchRepository` is an abstract class implementing the `ISearchRepository`. It also has the properties for concrete classes such as `SearchStoreRepository`, to initialize clients.

```
public SearchStoreRepository(IConfiguration configuration)
{
    _documentDbEndpoint = string.Format("https://{0}.documents.azure.com:443/",
                    configuration["DocumentDB:Endpoint"]);
    _documentDbAccountKey = configuration["DocumentDB:Key"];
    _documentDbDatabase = configuration["DocumentDB:DatabaseId"];

    string _azureSearchServiceName = configuration["SearchService:Name"];
    string _azureSearchServiceKey = configuration["SearchService:ApiKey"];

    _serviceClient = new SearchServiceClient(_azureSearchServiceName,
        new SearchCredentials(_azureSearchServiceKey));
}
```

There are 3 parts for binding a search service to a Cosmos DB collection:

1. Create the **datasource**
2. Create the **index**
3. Create the **indexer**

Open `SearchRepository` class to see the implementations for these one by one.

```
public async Task CreateOrUpdateDocumentDbDataSourceAsync(
            string dataSourceName, string dataSourceColletion)
{
    string connectionString =
        string.Format("AccountEndpoint={0};AccountKey={1};Database={2}",
        _documentDbEndpoint, _documentDbAccountKey, _documentDbDatabase);
    DataSourceCredentials credentials =
        new DataSourceCredentials(connectionString);
    DataContainer container = new DataContainer(dataSourceColletion);
    DataSource docDbSource =
        new DataSource(dataSourceName, DataSourceType.DocumentDb,
                        credentials, container);
    await _serviceClient.DataSources.CreateOrUpdateAsync(docDbSource);
}
```

`CreateOrUpdateDocumentDbDataSourceAsync` method needs the Cosmos DB connection string to build the data source. `CreateOrUpdateIndexAsync` method creates the index which will be used for querying the Search Service.

```
public async Task CreateOrUpdateIndexAsync<T>(
                string indexName, string suggesterName,                    \
            List<string> suggesterFields)
{
    List<Suggester> suggesters = new List<Suggester>();
    suggesters.Add(new Suggester(suggesterName,
        SuggesterSearchMode.AnalyzingInfixMatching, suggesterFields));
    var definition = new Index()
    {
        Name = indexName,
        Fields = FieldBuilder.BuildForType<T>(),
        Suggesters = suggesters
    };

    await _serviceClient.Indexes.CreateOrUpdateAsync(definition);
}
```

This method is generic and can create indexes for any type of component in Cosmos DB. Mind though that type `<T>` should have certain attributes from `Microsoft.Azure.Search` namespace that informs the SDK how to build the index. Check `ProductInfo` in `Online.Store.Core` to view some of those attributes. The following table comes directly from the official documentation for creating indexes.

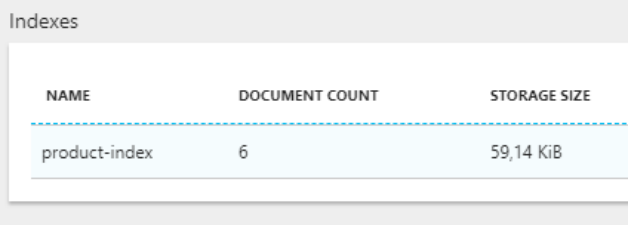
Attribute	Description
searchable	Full-text searchable, subject to lexical analysis such as word-breaking during indexing. If you set a searchable field to a value like "sunny day", internally it will be split into the individual tokens "sunny" and "day".
filterable	Referenced in **$filter** queries. Filterable fields of type `Edm.String` or `Collection(Edm.String)` do not undergo word-breaking, so comparisons are for exact matches only. For example, if you set such a field f to "sunny day", `$filter=f eq 'sunny'` will find no matches, but `$filter=f eq 'sunny day'` will.
sortable	By default the system sorts results by score, but you can configure sort based on fields in the documents. Fields of type `Collection(Edm.String)` cannot be **sortable**.
facetable	Typically used in a presentation of search results that includes a hit count by category (for example, hotels in a specific city). This option cannot be used with fields of type `Edm.GeographyPoint`. Fields of type `Edm.String` that are **filterable**, **sortable**, or **facetable** can be at most 32 kilobytes in length. For details, see Create Index (REST API).

Attribute	Description
key	Unique identifier for documents within the index. Exactly one field must be chosen as the key field and it must be of type `Edm.String`.
retrievable	Determines whether the field can be returned in a search result. This is useful when you want to use a field (such as *profit margin*) as a filter, sorting, or scoring mechanism, but do not want the field to be visible to the end user. This attribute must be `true` for `key` fields.

The last method is the one that creates the **indexer** and completes an index in Azure Service.

```
public async Task CreateOrUpdateIndexerAsync(string indexerName,
                              string dataSource, string index)
{
    await _serviceClient.Indexers.CreateOrUpdateAsync(
            new Indexer(indexerName, dataSource, index));
}
```

After running the project again, go back in the portal and verify that the index has been created successfully. Click the `Search Service` resource in your parent resource group and select `Overview`. You should see the `product-index` having indexed 6 product documents.



Product index

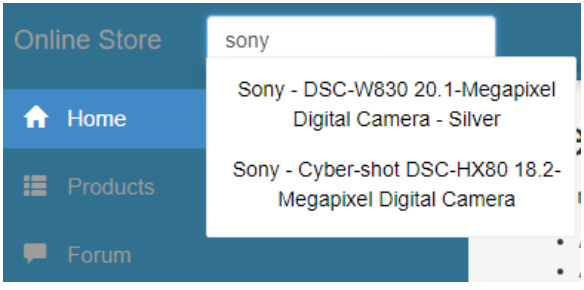
Click on the index and you should see the configuration have been set through code in `ProductInfo` class.

Fields

FIELD NAME	TYPE	ATTRIBUTES
id	Edm.String	Key, Filterable, Retrievable
title	Edm.String	Searchable, Filterable, Retrievable
model	Edm.String	Searchable, Filterable, Retrievable
sku	Edm.String	Retrievable
price	Edm.Double	Retrievable
image	Edm.String	Retrievable
description	Edm.String	Searchable, Retrievable
rating	Edm.Double	Sortable, Retrievable
rates	Edm.Int32	Retrievable
created	Edm.DateTimeOffset	Retrievable

Product index configuration

You should be able to search for products in the app. Try searching the term `sony`.



Search Service results

The library project for `Redis Cache` is the `Online.Store.RedisCahce`. It uses the Caching.Redis NuGet package to connect and access the Azure Redis Cache. The `IRedisCacheRepository` interface declares methods to save items of type *string* or *object*, a generic method to get an item and last but not least a method to remove an item from cache. Items in Cache are saved in key-value pairs.

```
public interface IRedisCacheRepository
{
    Task SetStringAsync(string key, string value);
    Task SetStringAsync(string key, string value, int expirationMinutes);

    Task SetItemAsync(string key, object item);
    Task SetItemAsync(string key, object item, int expirationMinutes);

    Task<T> GetItemAsync<T>(string key);
    Task RemoveAsync(string key);
}
```

`RedisCacheRepository` is the concrete class that implements the `IRedisCacheRepository` interface. It uses an instance of `IDistributedCache` for accessing the cache. These are the methods available in `IdistributedCache` interface:

```
public interface IDistributedCache
{
    byte[] Get(string key);
    Task<byte[]> GetAsync(string key, CancellationToken token
        = default(CancellationToken));
    void Refresh(string key);
    Task RefreshAsync(string key, CancellationToken token
        = default(CancellationToken));
    void Remove(string key);
    Task RemoveAsync(string key, CancellationToken token
        = default(CancellationToken));
    void Set(string key, byte[] value,
            DistributedCacheEntryOptions options);
    Task SetAsync(string key, byte[] value,
        DistributedCacheEntryOptions options,
        CancellationToken token = default(CancellationToken));
}
```

`SetStringAsync` in `RedisCacheReposistory` saves a `string` value in cache.

```
public async Task SetStringAsync(string key, string value, int expirationMinutes)
{
    var options = new DistributedCacheEntryOptions
    {
        AbsoluteExpiration = DateTime.Now.AddMinutes(expirationMinutes)
    };

    await _distributedCache
            .SetAsync(key, Encoding.UTF8.GetBytes(value), options);
}
```

`SetItemAsync` method uses internally the same method to save a serializable object.

```
public async Task SetItemAsync(string key, object item)
{
    string json = JsonConvert.SerializeObject(item);

    await SetStringAsync(key, json);
}
```

`GetItemAsync<T>` and `RemoveAsync` methods retrieves and removes an item from cache respectively.

```
public async Task<T> GetItemAsync<T>(string key)
{
    string json = await _distributedCache.GetStringAsync(key);

    if (json == null)
        return default(T);

    return JsonConvert.DeserializeObject<T>(json);
}

public async Task RemoveAsync(string key)
{
    await _distributedCache.RemoveAsync(key);
}
```

The configuration for `Redis Cache` exist in the `Online.Store` **Startup** class.

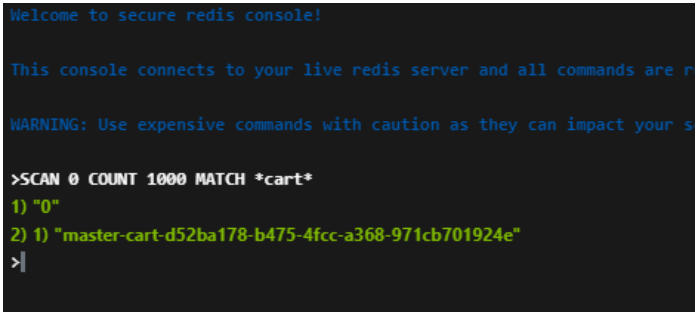
```
services.AddDistributedRedisCache(option =>
{
    option.Configuration = string.Format(RedisCacheConnStringFormat,
                Configuration["RedisCache:Endpoint"] +
                    ".redis.cache.windows.net:6380",
                Configuration["RedisCache:Key"]);

    option.InstanceName = "master-";
});
```

The instance name will prefix all keys stored in this cache. `Online.Store` application uses the cache to store `cart items`. Check the `AddProductToCart` method inside the `StoreService` class.

```
// code omitted
cart = await _cacheRepository.GetItemAsync<CartDTO>("cart-" + cardId);
// code omitted
await _cacheRepository.SetItemAsync("cart-" + cardId, cart);
```

The *cartId* is a **Guid** generated and saved in browser's cookie to track the cart. Run the web app and add products to cart. Back in the portal open your `Redis Cache` instance and in the main `Overview` blade click `>_Console`. Run the following command to list all cart keys stored in the cache.

```
SCAN 0 COUNT 1000 MATCH *cart*
```



Redis Cache - List cart keys

You can use the `FLUSHALL` command to clear all items in the cache.

The library project for `Service Bus` operations is the `Online.Store.ServiceBus`. It uses the Microsoft.Azure.ServiceBus NuGet package and has a simple `IServiceBusRepository` interface.

```
public interface IServiceBusRepository
{
    void InitQueueClient(string serviceBusAccessKeyName,
                string serviceBusAccessKey, string queue);

    Task SendAsync(object message);
}
```

`ServiceBusRepository` class is the concrete class that implements the interface. It uses a instance of `IQueueClient` to send messages to queues. Following is the implementation for `IServiceBusRepository` repository.

```
public void InitQueueClient(string serviceBusAccessKeyName,
                  string serviceBusAccessKey, string queue)
{
  var connectionString =
    $"Endpoint=sb://{_serviceBusNamespace}" +
    $".servicebus.windows.net/;SharedAccessKeyName={serviceBusAccessKeyName};"+
    $"SharedAccessKey={serviceBusAccessKey}";

    _queueClient = new QueueClient(connectionString, queue);
}

public async Task SendAsync(object body)
{
    var serializedBody = JsonConvert.SerializeObject(body);
    var message = new Message(Encoding.UTF8.GetBytes(serializedBody));
    await _queueClient.SendAsync(message);
}
```

`Online.Store` application uses Service Bus to submit *orders* to the orders queue you created before. It hasn't have to wait till the order is actually saved in database, a **Web Job** will be responsible to complete this task asynchronously. Check the `ServiceBusService` class inside `Online.Store.Azure.Services` project.

```
public async Task SubmitOrderAsync(Order order)
{
    _serviceBusRepository
        .InitQueueClient(_configuration["ServiceBus:WriteAccessKeyName"],
                         _configuration["ServiceBus:WriteAccessKey"],
                         _configuration["ServiceBus:Queue"]);

    await _serviceBusRepository.SendAsync(order);
}
```

At this point you cannot test service bus operations because you haven't setup **Authentication**, the required **Azure SQL Database** for saving orders and of course the **WebJob** that listens for and processes messages from the *orders* queue. The part that follows covers all these and when you are done you will have a full operational website. Next you will proceed with the **Global Scale** part where you will actually deploy the website at least at two Azure Regions.

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.

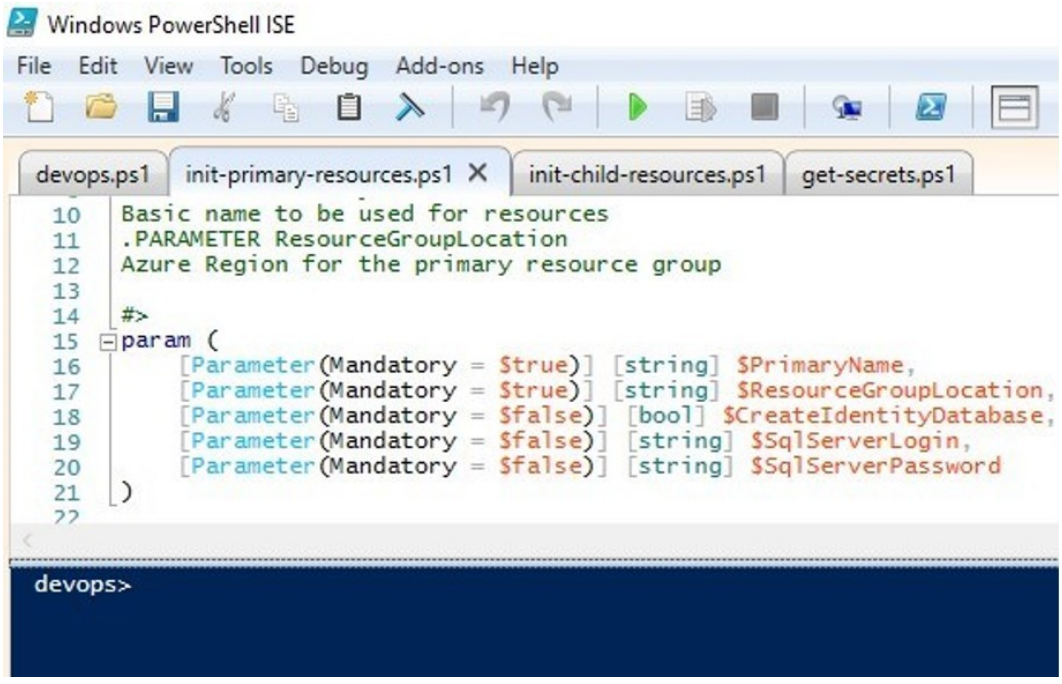
You could catch some of those errors before starting the provisioning process by checking the Products available by region page. The following regions have been tested many times during the development of `Online.Store` app and are highly recommended

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 useful 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 powerful intellisense experience.



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
    $cdnEnpointName = "$PrimaryName-$endpointPrefix";
    $endpointHost =  "$storageAccountName.blob.core.windows.net"

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

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

        New-AzureRmCdnEndpoint -ProfileName $cdnProfileName `
         -ResourceGroupName $PrimaryName `
         -Location $ResourceGroupLocation -EndpointName $cdnEnpointName `
         -OriginName "$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 successfully 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. In case you run the scripts in Mac or Linux make sure to change the references to the files inside the `mac-linux` folder

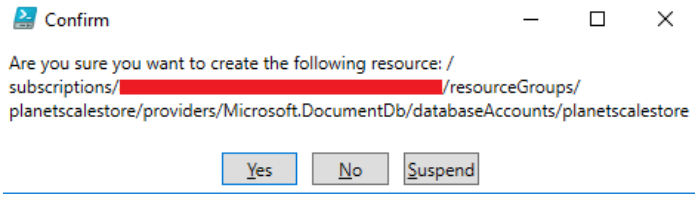
Run the `init-primary-resources.ps1` script as follow:

```
.\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.