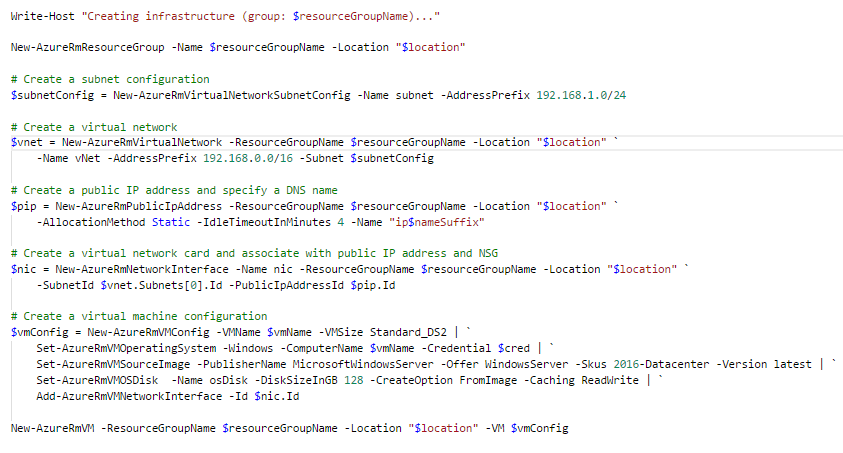
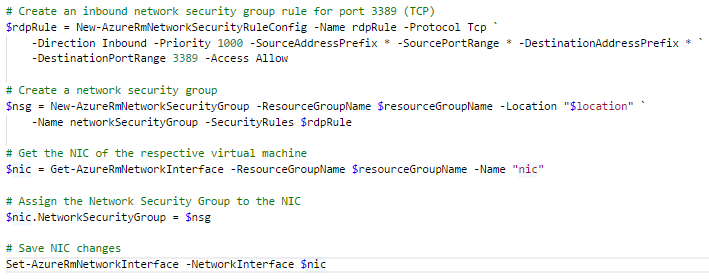
# Powershell commands to Create VM:

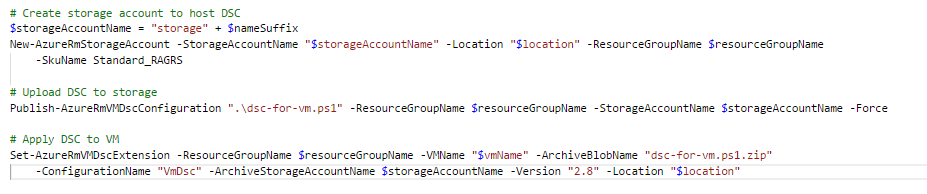
## Create VM



## Create NSG



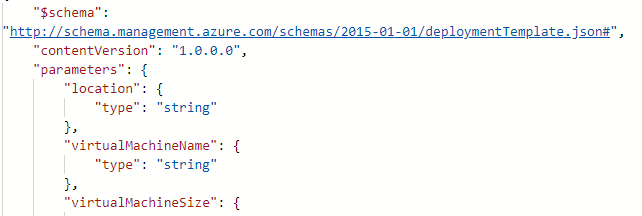
## Execute DSC

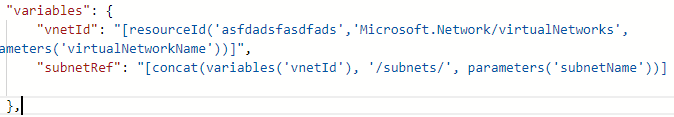


# General ARM Template



There is also a variables and outputs.





Variables are dependent here on parameters. That you can remember.

ex: 

Here ConnectionString is concat of 2 parameters.

"resources": [

{

"name": "[parameters('virtualMachineName')]",

"type": "Microsoft.Compute/virtualMachines",

"apiVersion": "2018-04-01",

"location": "[parameters('location')]",

"dependsOn": [

"[concat('Microsoft.Network/networkInterfaces/', parameters('networkInterfaceName'))]",

"[concat('Microsoft.Storage/storageAccounts/', parameters('diagnosticsStorageAccountName'))]"

],

"properties": {

"osProfile": {

"computerName": "[parameters('virtualMachineName')]",

"adminUsername": "[parameters('adminUsername')]",

"adminPassword": "[parameters('adminPassword')]",

"windowsConfiguration": {

"provisionVmAgent": "true"

}

},

"hardwareProfile": {

"vmSize": "[parameters('virtualMachineSize')]"

},

"storageProfile": {

"imageReference": {

"publisher": "MicrosoftWindowsServer",

"offer": "WindowsServer",

"sku": "2016-Datacenter",

"version": "latest"

},

"osDisk": {

"createOption": "fromImage",

"managedDisk": {

"storageAccountType": "Premium\_LRS"

}

},

"dataDisks": []

},

"networkProfile": {

"networkInterfaces": [

{

"id": "[resourceId('Microsoft.Network/networkInterfaces', parameters('networkInterfaceName'))]"

}

]

},

"diagnosticsProfile": {

"bootDiagnostics": {

"enabled": true,

"storageUri": "[concat('https://', parameters('diagnosticsStorageAccountName'), '.blob.core.windows.net/')]"

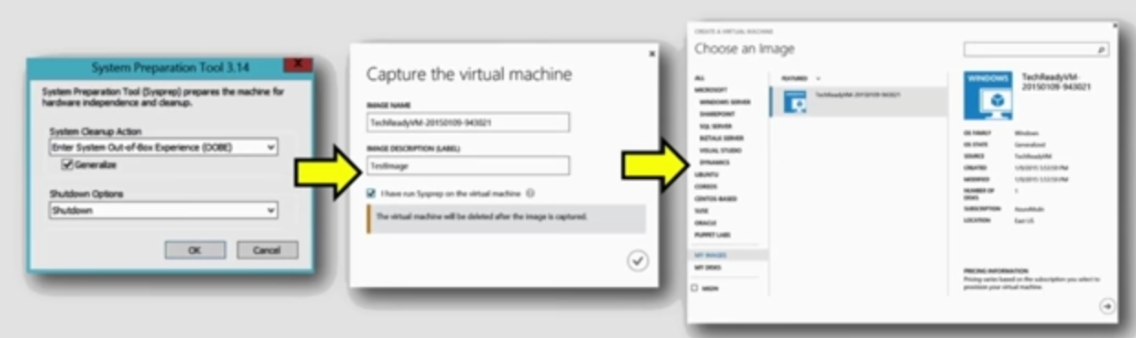
}

}

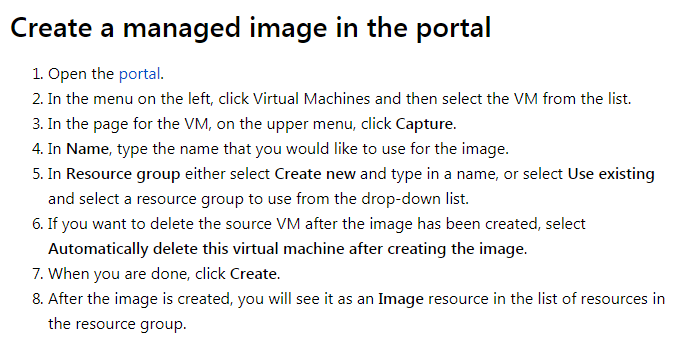
}

},

# Uploding Images



<https://docs.microsoft.com/en-us/azure/virtual-machines/windows/capture-image-resource>



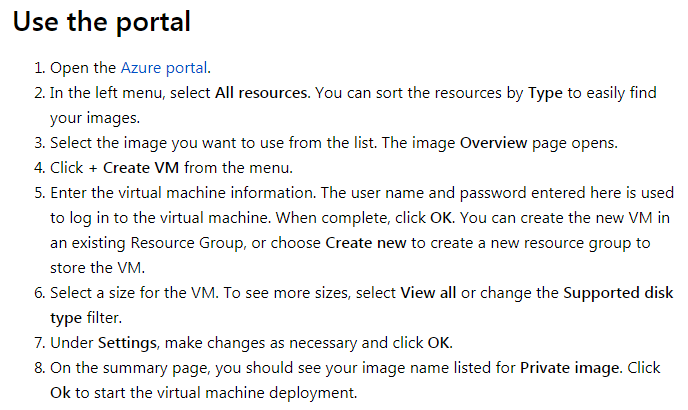
$image = New-AzureRmImageConfig -Location $location -SourceVirtualMachineId $vm.ID

New-AzureRmImage -Image $image -ImageName $imageName -ResourceGroupName $rgName

And then use the image like this:



OR



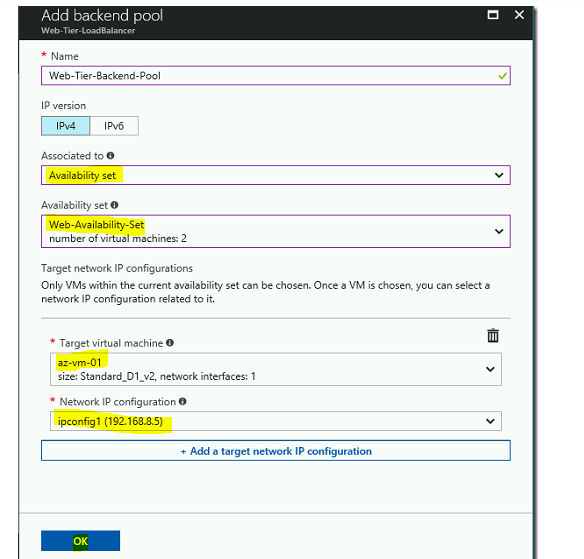
Creating an image directly from the VM ensures that the image includes all of the disks associated with the VM, including the OS Disk and any data disks.

We can just get OS image also.

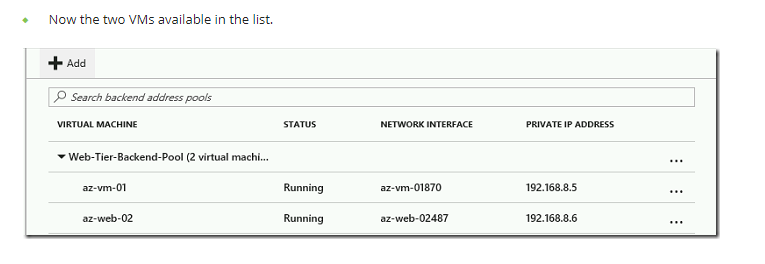
# Azure Availability Sets and Load Balancer

<https://www.assistanz.com/create-azure-web-farm-using-load-balancer-and-high-availability-set/>

NSG is for blocking ports. remember that.



The first one of choosing Avaliability set is only for filtering VM. After that chose the VM in Target Virtual machine there.



# Storage Replication Options

LRS: 3 nodes in same data center.

ZRS: upto 3 data centers in same region.

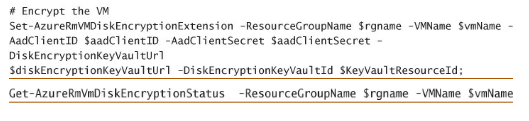
GRS: data center in different region.

Read-Access: Data center in diff region and secondary can be read.

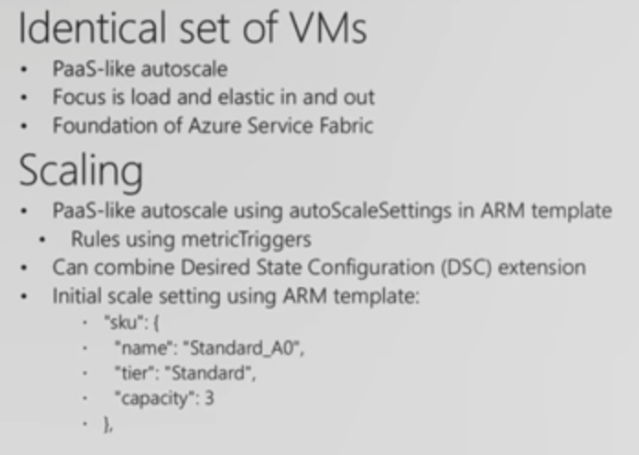
# Encrypting a disc in Azure

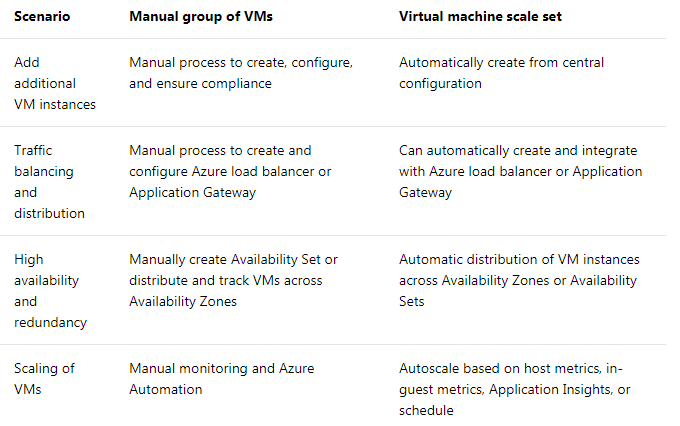
Create a Key vault

AAD app with write secrets permission to keyvault.



# VM Scale Sets





# SAS Token Creation

static void Main(string[] args)

{

//Parse the connection string and return a reference to the storage account.

CloudStorageAccount storageAccount = CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting("StorageConnectionString"));

//Create the blob client object.

CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

//Get a reference to a container to use for the sample code, and create it if it does not exist.

CloudBlobContainer container = blobClient.GetContainerReference("sascontainer");

container.CreateIfNotExists();

//Insert calls to the methods created below here...

//Require user input before closing the console window.

Console.ReadLine();

}

static string GetContainerSasUri(CloudBlobContainer container)

{

//Set the expiry time and permissions for the container.

//In this case no start time is specified, so the shared access signature becomes valid immediately.

SharedAccessBlobPolicy sasConstraints = new SharedAccessBlobPolicy();

sasConstraints.SharedAccessExpiryTime = DateTimeOffset.UtcNow.AddHours(24);

sasConstraints.Permissions = SharedAccessBlobPermissions.List | SharedAccessBlobPermissions.Write;

//Generate the shared access signature on the container, setting the constraints directly on the signature.

string sasContainerToken = container.GetSharedAccessSignature(sasConstraints);

//Return the URI string for the container, including the SAS token.

return container.Uri + sasContainerToken;

}

//Generate a SAS URI for the container, without a stored access policy.

Console.WriteLine("Container SAS URI: " + GetContainerSasUri(container));

Console.WriteLine();

 For powershell

# Append, Page and Block Blobs

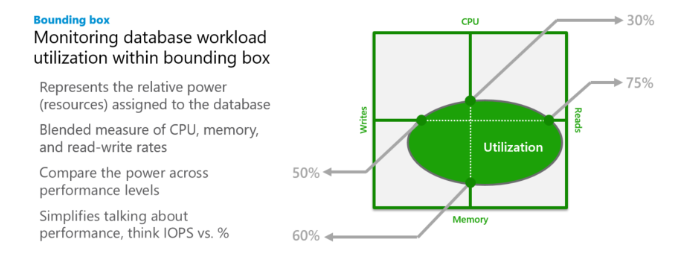
Block blobs let you upload large blobs efficiently. Block blobs are comprised of blocks, each of which is identified by a block ID. You create or modify a block blob by writing a set of blocks and committing them by their block IDs. Each block can be a different size, up to a maximum of 100 MB (4 MB for requests using REST versions before 2016-05-31), and a block blob can include up to 50,000 blocks. The maximum size of a block blob is therefore slightly more than 4.75 TB (100 MB X 50,000 blocks). For REST versions before 2016-05-31, the maximum size of a block blob is a little more than 195 GB (4 MB X 50,000 blocks). If you are writing a block blob that is no more than 256 MB (64 MB for requests using REST versions before 2016-05-31) in size, you can upload it in its entirety with a single write operation; see [Put Blob](https://docs.microsoft.com/en-us/rest/api/storageservices/put-blob).

## Azure Blob VS Files

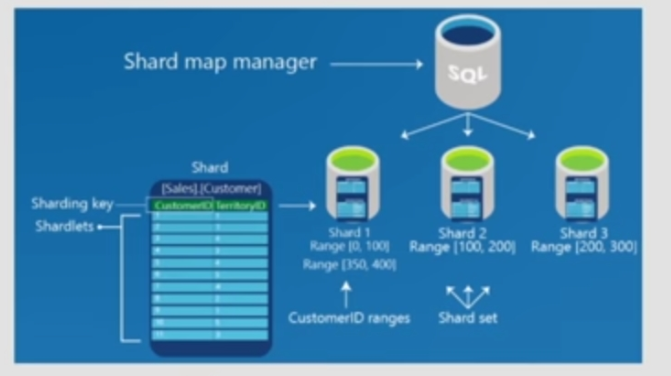


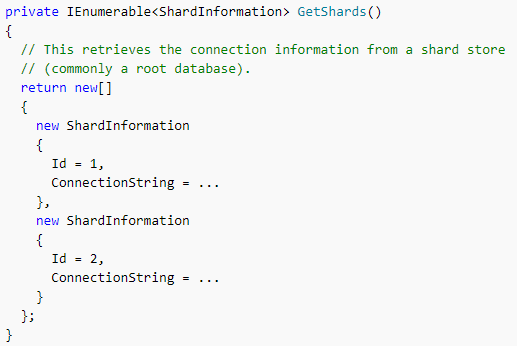
# SQL Database

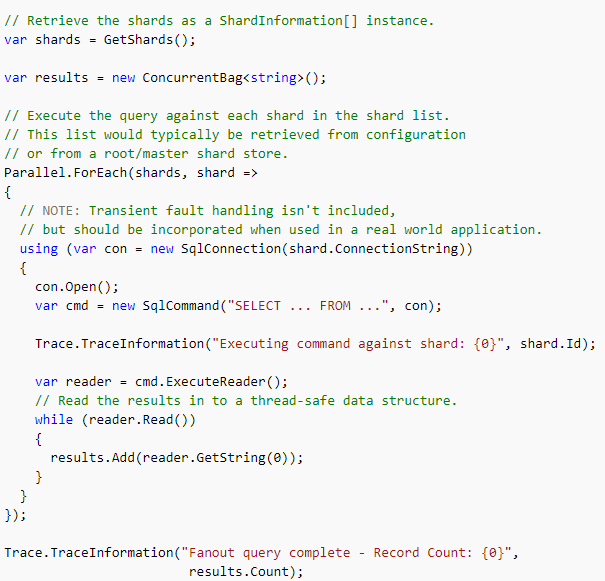


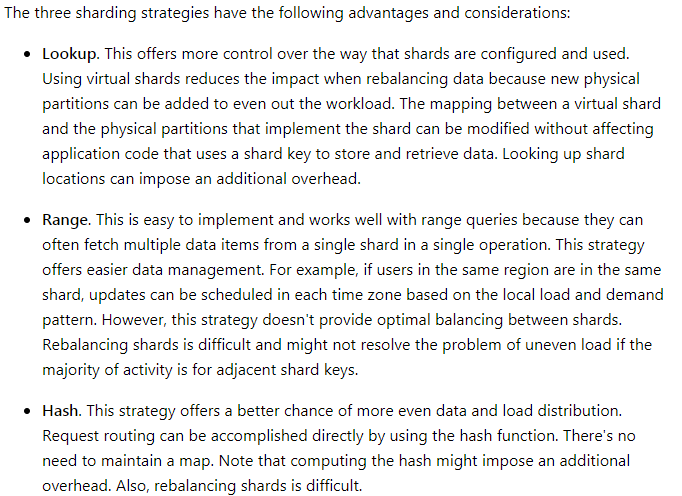


# Sharding



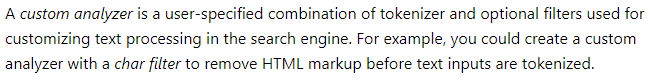


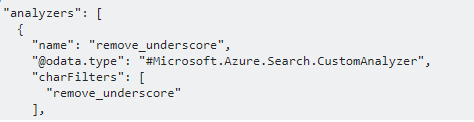




# Azure Search

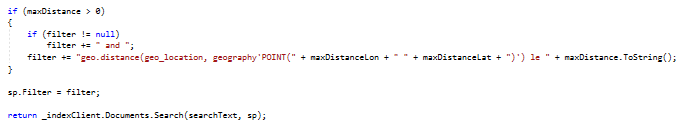
We will create an index, with fields, analysers etc



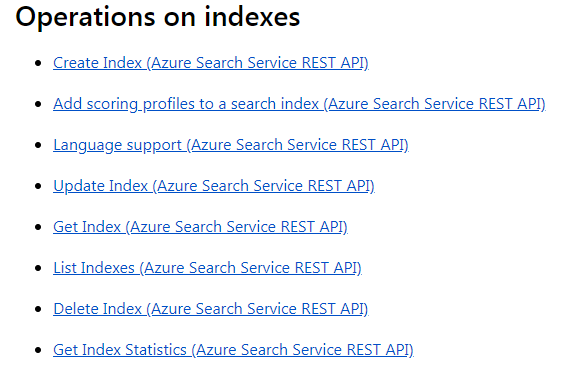








There can be a fuzzy search also for cases where its not misspelled strings.



Understand the above. Map it to the Code we wrote in other article.

{

"name": (optional on PUT; required on POST) "name\_of\_index",

"fields": [

{

"name": "name\_of\_field",

"type": "Edm.String | Collection(Edm.String) | Edm.Int32 | Edm.Int64 | Edm.Double | Edm.Boolean | Edm.DateTimeOffset | Edm.GeographyPoint",

"searchable": true (default where applicable) | false (only Edm.String and Collection(Edm.String) fields can be searchable),

"filterable": true (default) | false,

"sortable": true (default where applicable) | false (Collection(Edm.String) fields cannot be sortable),

"facetable": true (default where applicable) | false (Edm.GeographyPoint fields cannot be facetable),

"key": true | false (default, only Edm.String fields can be keys),

"retrievable": true (default) | false,

"analyzer": "name of the analyzer used for search and indexing", (only if 'searchAnalyzer' and 'indexAnalyzer' are not set)

"searchAnalyzer": "name of the search analyzer", (only if 'indexAnalyzer' is set and 'analyzer' is not set)

"indexAnalyzer": "name of the indexing analyzer" (only if 'searchAnalyzer' is set and 'analyzer' is not set)

}

],

"suggesters": [

{

"name": "name of suggester",

"searchMode": "analyzingInfixMatching" (other modes may be added in the future),

"sourceFields": ["field1", "field2", ...]

}

],

"scoringProfiles": [

{

"name": "name of scoring profile",

"text": (optional, only applies to searchable fields) {

"weights": {

"searchable\_field\_name": relative\_weight\_value (positive #'s),

...

}

},

"functions": (optional) [

{

"type": "magnitude | freshness | distance | tag",

"boost": # (positive number used as multiplier for raw score != 1),

"fieldName": "...",

"interpolation": "constant | linear (default) | quadratic | logarithmic",

"magnitude": {

"boostingRangeStart": #,

"boostingRangeEnd": #,

"constantBoostBeyondRange": true | false (default)

},

"freshness": {

"boostingDuration": "..." (value representing timespan leading to now over which boosting occurs)

},

"distance": {

"referencePointParameter": "...", (parameter to be passed in queries to use as reference location)

"boostingDistance": # (the distance in kilometers from the reference location where the boosting range ends)

},

"tag": {

"tagsParameter": "..." (parameter to be passed in queries to specify a list of tags to compare against target fields)

}

}

],

"functionAggregation": (optional, applies only when functions are specified)

"sum (default) | average | minimum | maximum | firstMatching"

}

],

"analyzers":(optional)[ ... ],

"charFilters":(optional)[ ... ],

"tokenizers":(optional)[ ... ],

"tokenFilters":(optional)[ ... ],

"defaultScoringProfile": (optional) "...",

"corsOptions": (optional) {

"allowedOrigins": ["\*"] | ["origin\_1", "origin\_2", ...],

"maxAgeInSeconds": (optional) max\_age\_in\_seconds (non-negative integer)

}

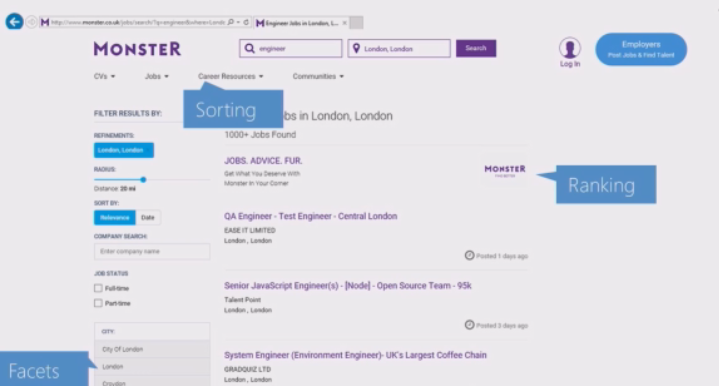
}

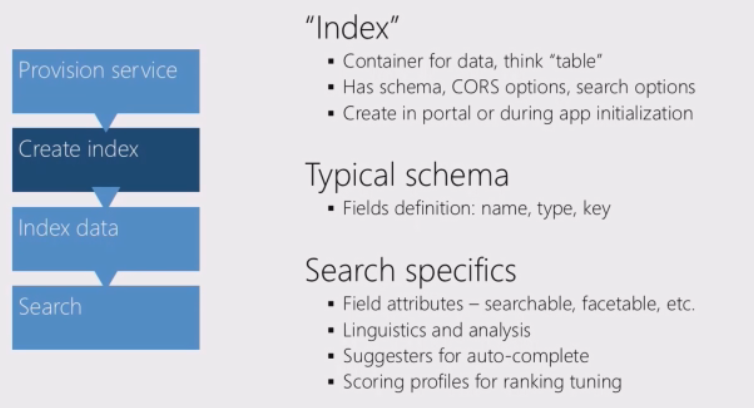
A typical scoring profile looks like this:

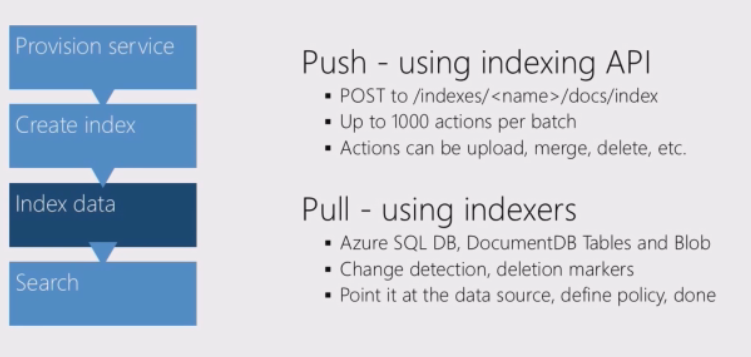




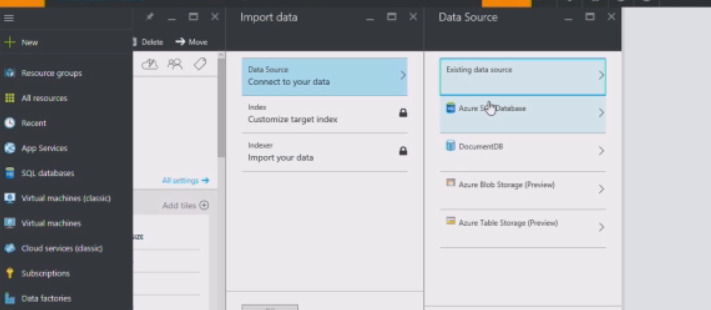
Scoring Profile will have to specified as part of search parameter we wrote above.



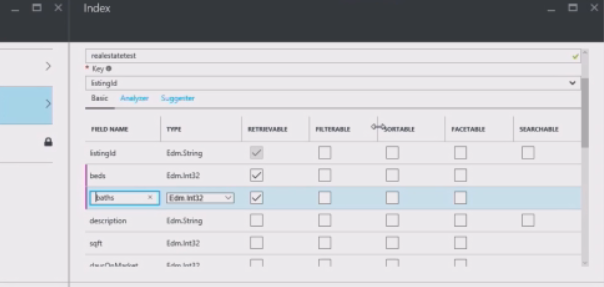


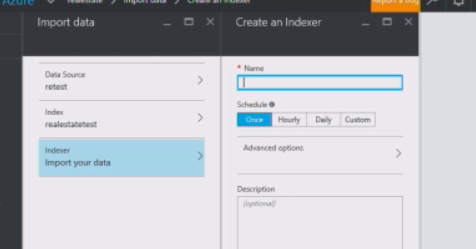


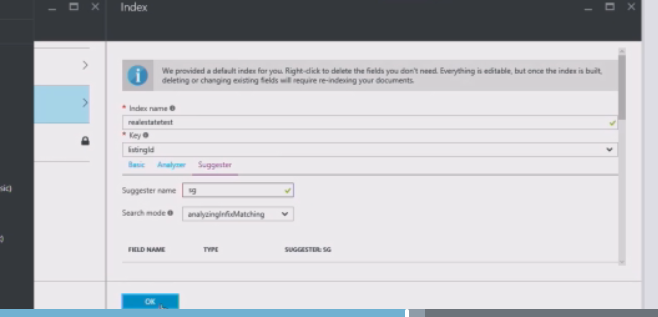
Indexer is nothing predefined sources of data. Like user indexer to pull data from SQL server.



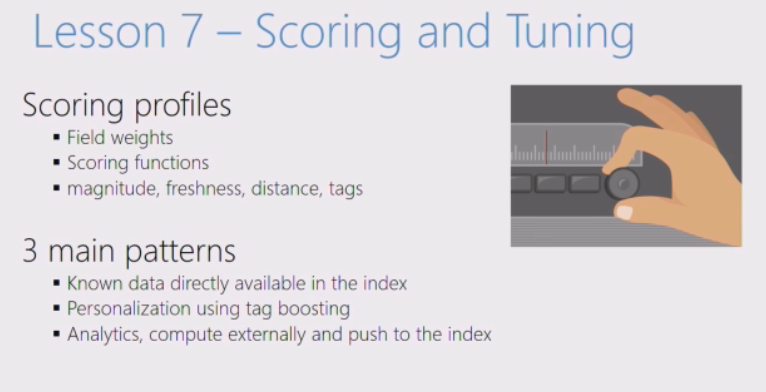
After importing do the ONE Below:

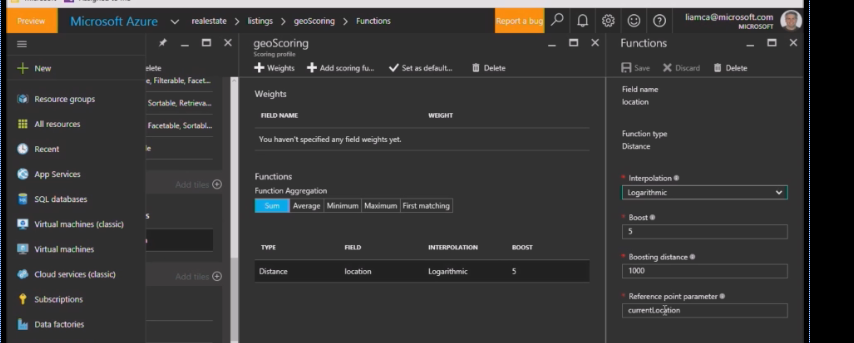


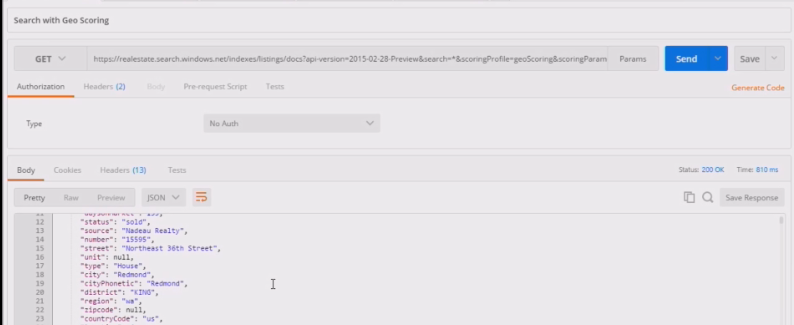




Analyzer is for language analyzing and Suggester if for type ahead functionality.







# Redis Cache

static void Main(string[] args)

{

InitializeConfiguration();

// Connection refers to a property that returns a ConnectionMultiplexer

// as shown in the previous example.

IDatabase cache = lazyConnection.Value.GetDatabase();

// Perform cache operations using the cache object...

// Simple PING command

string cacheCommand = "PING";

Console.WriteLine("\nCache command : " + cacheCommand);

Console.WriteLine("Cache response : " + cache.Execute(cacheCommand).ToString());

// Simple get and put of integral data types into the cache

cacheCommand = "GET Message";

Console.WriteLine("\nCache command : " + cacheCommand + " or StringGet()");

Console.WriteLine("Cache response : " + cache.StringGet("Message").ToString());

cacheCommand = "SET Message \"Hello! The cache is working from a .NET Core console app!\"";

Console.WriteLine("\nCache command : " + cacheCommand + " or StringSet()");

Console.WriteLine("Cache response : " + cache.StringSet("Message", "Hello! The cache is working from a .NET Core console app!").ToString());

// Demostrate "SET Message" executed as expected...

cacheCommand = "GET Message";

Console.WriteLine("\nCache command : " + cacheCommand + " or StringGet()");

Console.WriteLine("Cache response : " + cache.StringGet("Message").ToString());

// Get the client list, useful to see if connection list is growing...

cacheCommand = "CLIENT LIST";

Console.WriteLine("\nCache command : " + cacheCommand);

Console.WriteLine("Cache response : \n" + cache.Execute("CLIENT", "LIST").ToString().Replace("id=", "id="));

lazyConnection.Value.Dispose();

}

private static Lazy<ConnectionMultiplexer> lazyConnection = new Lazy<ConnectionMultiplexer>(() =>

{

string cacheConnection = Configuration[SecretName];

return ConnectionMultiplexer.Connect(cacheConnection);

});

public static ConnectionMultiplexer Connection

{

get

{

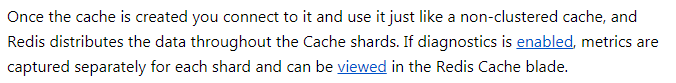
return lazyConnection.Value;

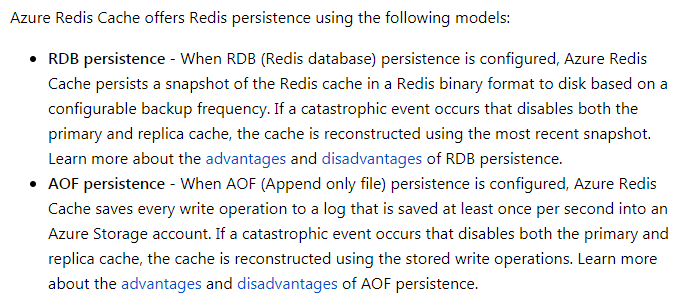
}

}

Clustering:

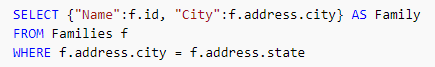
After completing the clustering,



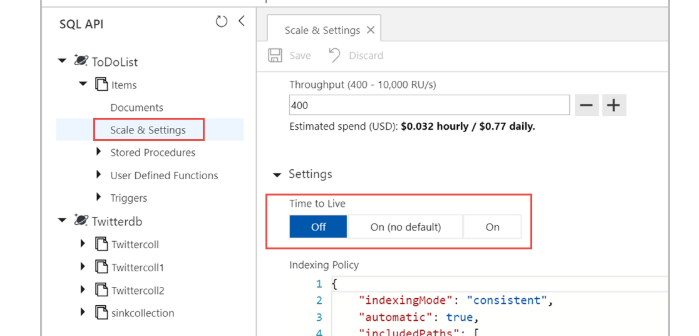




# Cosmos DB



TTL can be set per document or a collection basis like this:



DocumentCollection collectionDefinition = new DocumentCollection();

collectionDefinition.Id = "orders";

collectionDefinition.PartitionKey.Paths.Add("/customerId");

collectionDefinition.DefaultTimeToLive =-1; //never expire by default

Database = await Client.CreateDatabaseIfNotExistsAsync(new Database { Id = CosmosSettings.DatabaseId });

DocumentCollection ttlEnabledCollection = await client.CreateDocumentCollectionAsync(

UriFactory.CreateDatabaseUri(databaseName),

collectionDefinition,

new RequestOptions { OfferThroughput = 20000 });

public async Task<string> UploadEventRegistrationAsync(dynamic registration)

{

ResourceResponse<Document> response = await Client.CreateDocumentAsync(Collection.SelfLink, registration);

return response.Resource.Id;

}

IDocumentQuery<int> query = Client.CreateDocumentQuery<int>(Collection.SelfLink, "SELECT VALUE COUNT(1) FROM registrants", options).AsDocumentQuery();

int count = 0;

while (query.HasMoreResults)

{

FeedResponse<int> results = await query.ExecuteNextAsync<int>();

count += results.Sum();

}

return count;

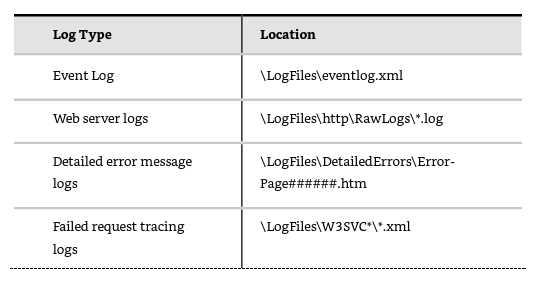
# Azure App Service Monitoring

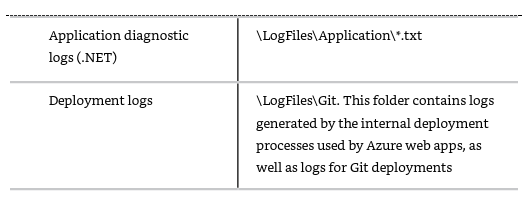
Event Log: capturing unhandled excepition

web server logs: entry for each http request

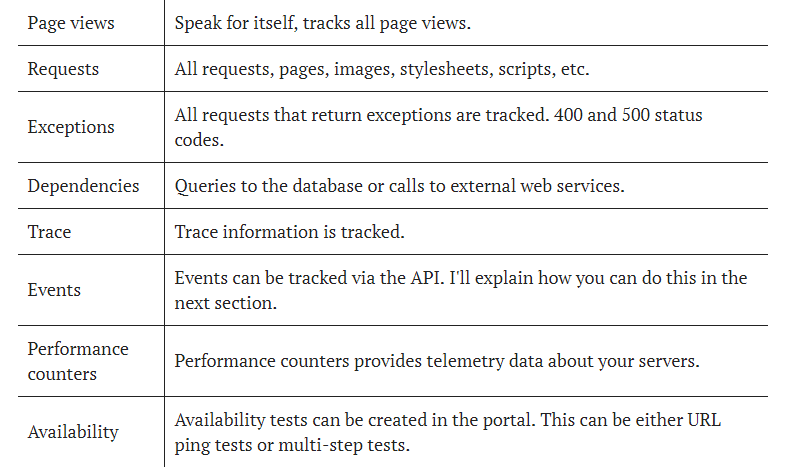
detailed error log: http status 400 or higher

failed request tracing logs: has stack trace

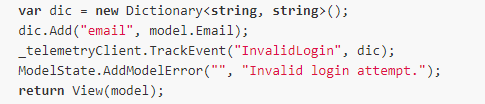




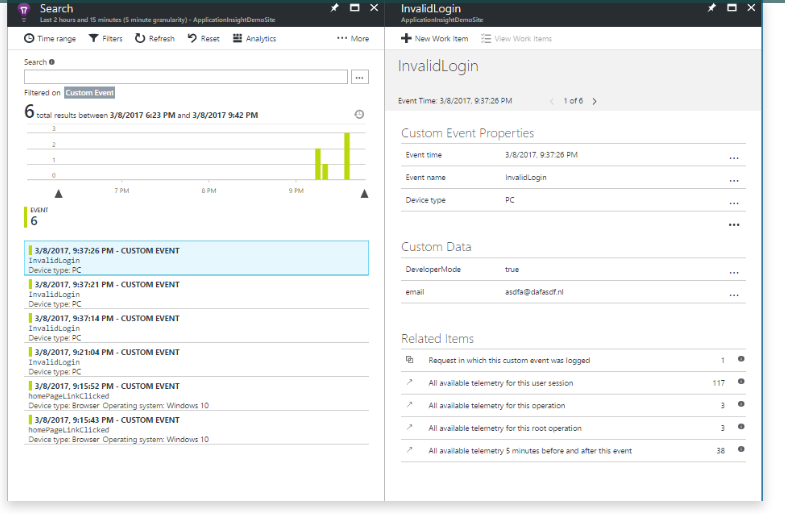
# Application Insights

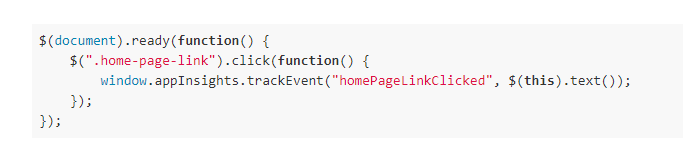


Telemetry data is held in memory and pushed in batches to Application Insights. Expect a short delay before the telemetry data appear in the portal. When you're in development mode data is send immediately. You could also persist the telemetry data. This can be useful when you're developing a mobile application. If the app crash the data is still available and can be pushed when the app restarts.



A custom event to tract invalid login like above.





The above case always add a 1 to telementry. We can also do a custom value like this:

Metric itemsInDatastructure = client.GetMetric(

"ItemsInDatastructure",

new Microsoft.ApplicationInsights.Metrics.MetricConfiguration(

1000,

100,

new Microsoft.ApplicationInsights.Metrics.TestUtility.MetricSeriesConfigurationForTestingAccumulatorBehavior()));

int itemsAdded = AddItemsToDataStructure();

itemsInDatastructure.TrackValue(itemsAdded);

int itemsRemoved = AddItemsToDataStructure();

itemsInDatastructure.TrackValue(-itemsRemoved);

or event this:

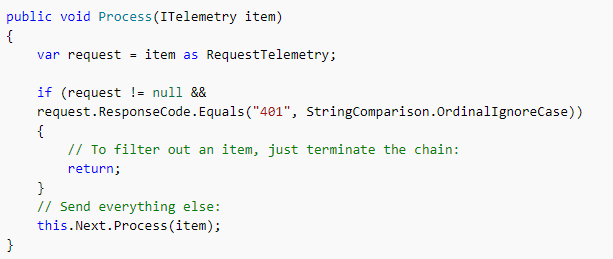
Metric animalsSold = client.GetMetric("AnimalsSold", "Species");

animalsSold.TrackValue(42, "Pigs");

animalsSold.TrackValue(24, "Horses");

Second one is example of Custom metrics.

Filtering of data:



<https://www.patrickvankleef.com/2017/03/12/track-telemetry-data-with-application-insights/>

# Azure Functions

[FunctionName("ProcessDocuments")]

public static async Task Run([BlobTrigger("signinsheets-pending/{name}")] Stream input, string name, [Blob("signinsheets/{name}", FileAccess.Write)] Stream output, TraceWriter log)

{

log.Info($"Request received to generate sign-in sheet for event: {name}");

string eventKey = Path.GetFileNameWithoutExtension(name);

using (MemoryStream stream = await ProcessStorageMessage(eventKey))

{

byte[] byteArray = stream.ToArray();

await output.WriteAsync(byteArray, 0, byteArray.Length);

}

log.Info($"Request received to generate sign-in sheet for event: {name}");

}

[FunctionName("GenericWebhookFunction")]

public static async Task<object> Run([HttpTrigger(WebHookType =

"genericJson")]HttpRequestMessage req, TraceWriter log)

{

log.Info($"Webhook was triggered!");

string jsonContent = await req.Content.ReadAsStringAsync();

dynamic data = JsonConvert.DeserializeObject(jsonContent);

if (data.first == null || data.last == null)

{

return req.CreateResponse(HttpStatusCode.BadRequest, new

{

error = "Please pass first/last properties in the input

object"

});

}

return req.CreateResponse(HttpStatusCode.OK, new

{

greeting = $"Hello {data.first} {data.last}!"

});

}

}

}

Exam Tips:

New-AzureRmResourceGroupDeployment -Name <deployment-name> -ResourceGroupName

<resource-group-name> -TemplateUri https://raw.githubusercontent.com/azure/azure-

quickstart-templates/master/101-hdinsight-hbase-replication-geo/azuredeploy.json

------------------------------------------------------------

App Insights

TelemetryClient client = new TelemetryClient();

client.TrackEvent("SomethingInterestingHappened");

client.GetMetric("CowsSold").TrackValue(42);

// By default, metrics are aggregated as Measurements. Here is how you can define a metric to be aggregated as an Accumulator instead:

Metric itemsInDatastructure = client.GetMetric("ItemsInDatastructure", MetricConfigurations.Common.Accumulator());

int itemsAdded = AddItemsToDataStructure();

itemsInDatastructure.TrackValue(itemsAdded);

int itemsRemoved = AddItemsToDataStructure();

itemsInDatastructure.TrackValue(-itemsRemoved);

------------------------------------------------------------

Securing Logic Apps: Logic APP url will have SAS token. to secure this, we can do:

1. Use the API management and update the rewrite rule to copy the sas token from header.

2. Use function app proxy to rewrite the URL accordingly.

https://blog.mexia.com.au/securing-sas-token-from-azure-logic-apps

-------------------------------------------------------------------------------

https://docs.microsoft.com/en-gb/azure/active-directory/develop/authentication-scenarios#native-application-to-web-api

For Native Apps at server,

1. Register the app with sign-on URL as : https://localhost:44321/

2. https://<your\_tenant\_name>/TodoListService-NativeDotNet' (replacing <your\_tenant\_name> with the name of your Azure AD tenant)

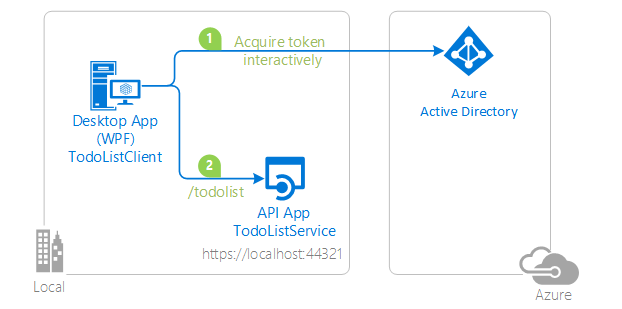
At Client:

In the Azure Active Directory pane, click on App registrations and choose New application registration.

Enter a friendly name for the application, for example 'TodoListClient-NativeDotNet' and select 'Native' as the Application Type.

For the Redirect URI, enter https://<your\_tenant\_name>/TodoListClient-NativeDotNet, replacing <your\_tenant\_name> with the name of your Azure AD tenant.

Click Create to create the application.



<https://docs.microsoft.com/en-gb/azure/virtual-machines/windows/upload-generalized-managed?toc=%2Fazure%2Fvirtual-machines%2Fwindows%2Fclassic%2Ftoc.json>

<https://docs.microsoft.com/en-gb/azure/service-bus-relay/relay-hybrid-connections-node-get-started>