Issue 46 | MARCH - APRIL 2020

DNC Magazine

www.dotnetcurry.com

AI and FACT FICTION

Automated Actions on Azure Monitor Alerts

Blazor, Tensorflow and ML.NET

Coding Practices: The most important ones

Unit Testing Angular Services

CLOUD Applications in .NET

Azure Cognitive Search APIs

LETTER FROM

THE EDITOR

COVID-19 - NAVIGATING THE UNCHARTED!

The stock market has tanked, there is a drop in the overall hiring sentiment, a recession looks likely, and the world is staring at the biggest pandemic of the century. There isn't a single soul, industry, or geography that has not been affected by Covid-19.

We have all heard a similar narrative in the past few days!

But once we get past this pandemic, how will we recover from it? How will it affect the way we design, build and communicate? Will we depend on technology more than ever to continue with the same pace of productivity?

Is there a silver lining?

In times of crisis, it can be hard to see past our own noses. So I may not know a definite answer to most of these questions, but I know of one thing that has never failed me during crisis.

Learning.

Learning provides insights, comfort and confidence. It ensures your foot is on the accelerator of your career.

Thankfully, in this connected world, learning resources are easy to access without leaving your safe abode. There are plenty of free online resources available, including this magazine, to help you accelerate your career and growth.

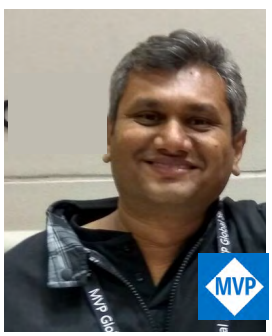
It's hard to focus during a pandemic, but don't give in to emotions. Use this lockdown to learn some technical, mental, behavioural and communication skills. Once you have acquired them, teach others. When this is all over, these skills will create new opportunities for you, and will help you outshine others.

So Stay safe, Stay Connected and don't worry.

..as this, too, shall pass!

I hope you enjoy this edition! Do write back and share your stories and learnings of how are you coping in these tough times.

P.S: We could not release the January edition due to a shortage of funds. If you would like to help your magazine, here's how you can.



Suprotim Agarwal
Editor in Chief

THE TEAM

Contributing Authors :
Benjamin Jakobus
Damir Arh
Daniel Jimenez Garcia
Mahesh Sabnis
Ravi Kiran
Subodh Sohoni
Yacoub Massad

Technical Reviewers :
Vikram Pendse
Suprotim Agarwal
Subodh Sohoni
Gouri Sohoni
Daniel Jimenez Garcia
Damir Arh

Next Edition : July 2020

Art Director : Minal Agarwal

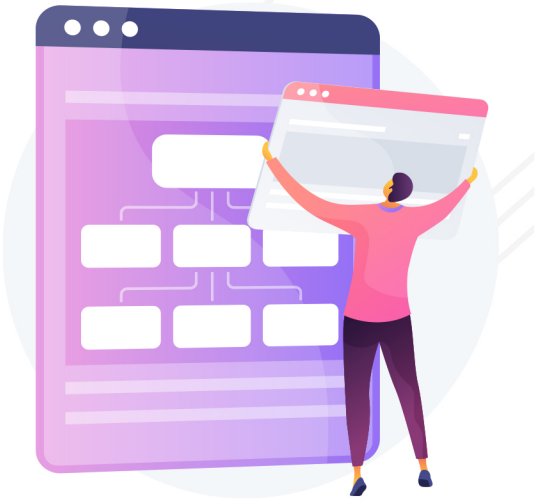
Editor In Chief : Suprotim Agarwal (suprotimagarwal@dotnetcurry.com)

a2z | Knowledge Visuals

et curry.com

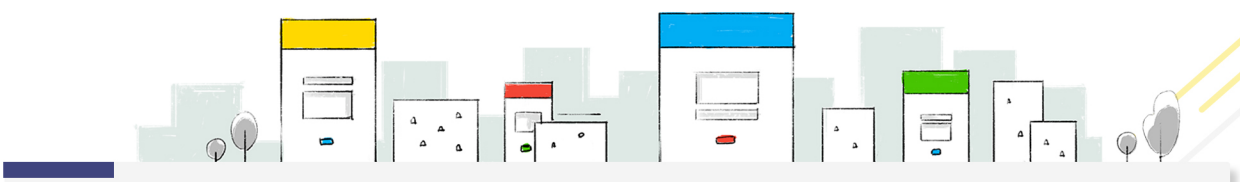
CONTENTS

ManageEngine
Applications Manager



Accelerate your migration to Microsoft Azure Cloud with enterprise-grade performance monitoring

- Get consistent monitoring regardless of environment – traditional, hybrid or Azure cloud.
- Runtime application dependency mapping.
- Code-level visibility.
- Pinpoint end user impact.
- Troubleshoot easily during each stage of migration Predictive analysis.
- Real-time dashboards.



> Applications Manager enables us to act very quickly to fix performance problems and our end users don't even notice any changes.

- Vadim Rapoport,
Technical Computing Advisor, DBA Sempra Utilities

Try us today for free!

www.manageengine.com/apm

Daniel Jimenez Garcia

The world of .NET has changed rapidly in the recent years, becoming a modern cross-platform environment aware of modern challenges. These challenges in the past were solved traditionally with the knowledge and expertise of other languages and frameworks. But today, .NET developers can apply their strengths and skills to solve complex problems.

For example, today one can use .NET to build a website without the need for JavaScript or to classify images with a Machine Learning model. This is made possible by two recent additions to the .NET ecosystem, Blazor and ML.NET.

USING **BLAZOR, TENSORFLOW AND ML.NET** TO IDENTIFY IMAGES

Blazor is a framework that lets .NET developers build client web applications entirely in .NET, without the need for a JavaScript framework. It leverages technologies such as SignalR and WebAssembly in order to support different hosting models, of which the server-side one based on SignalR is now part of ASP.NET Core (The client-side WebAssembly model is still experimental).

On the other hand, ML.NET is a new offering from Microsoft that provides an open source and cross platform framework for Machine Learning. It allows .NET developers to leverage their existing knowledge and skills while supporting popular existing Machine Learning technologies such as TensorFlow and ONNX.

Through the rest of the article we will explore how Blazor and ML.NET can be used to build a sample website that lets users upload images which will be classified by a pre-trained TensorFlow model using ML.NET. Enjoy!

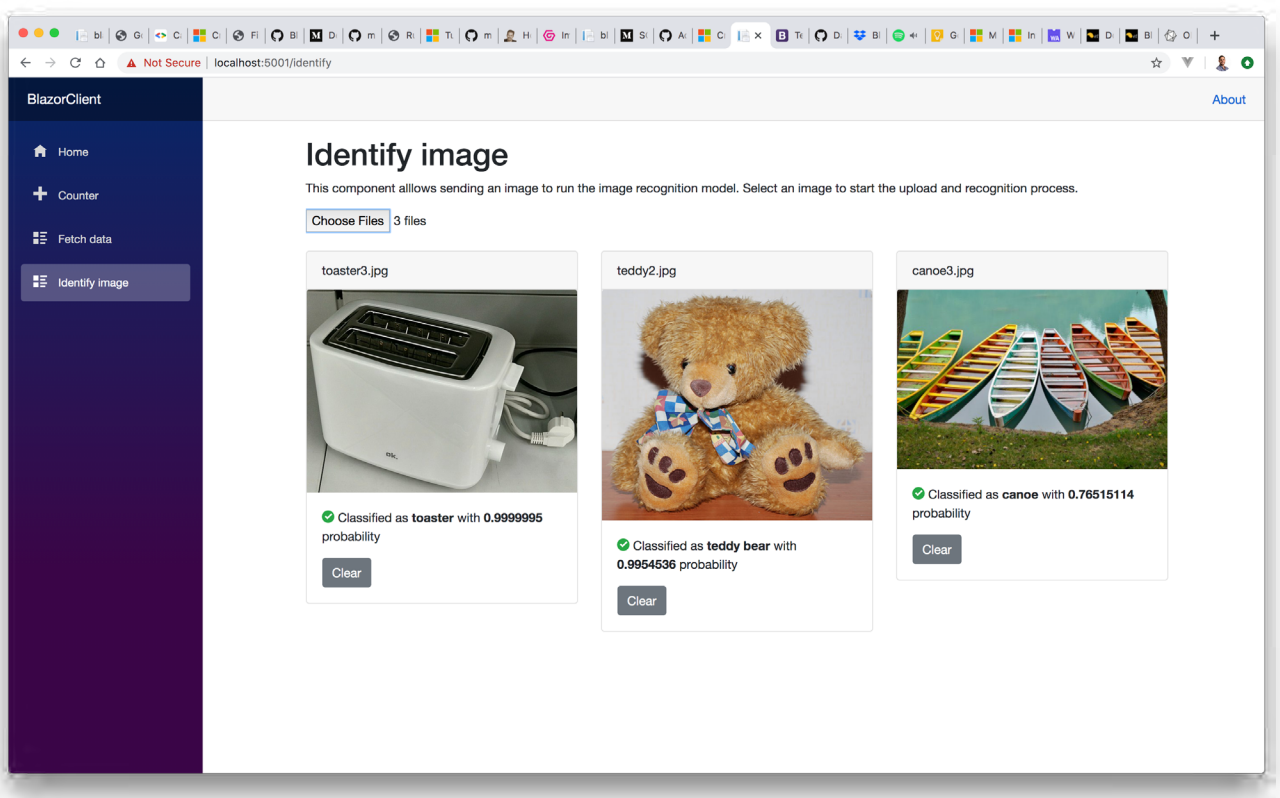


Figure 1, our target, a Blazor website classifying images with ML.NET

*You can find the **entire source code** on GitHub.*

Creating the ML.NET model

Machine Learning lets you use existing well-known data in order to create a model, which can later be used with previously unseen data to make predictions.

For example, one can use a pre-classified set of images with objects in them (a canoe, a teddy bear, a coffee pot, etc.) and create a model that can be used to identify the object in an image. The act of building a model is called *training a model*, and the data used to train it is called the *training dataset*.

The ML.NET framework provides an API that lets developers implement the following workflow:

- Define the data schema, for example a bitmap image
- Define transformations over the initial data, for example resizing the image and extracting the pixels
- **Train** the model using a training dataset, the defined transformations and one of the available algorithms.
- **Evaluate** the accuracy and precision of the model using a second dataset, the *evaluation dataset* (which is different from the training dataset!)
- Save the model so it can be used to make future predictions

You can see a simple example that puts together all these concepts in the official docs.

Creating an ML.NET model using an existing TensorFlow model

As mentioned in the introduction, ML.NET is compatible with TensorFlow, one of the most popular frameworks for building Machine Learning models. You can use an existing TensorFlow model as the starting point to derive further knowledge, as in this article of the docs. However, you can also use ML.NET to simply load an existing TensorFlow model and use it to make predictions.

In this article, we will implement an application that follows an idea similar to this excellent article by Cesar de la Torre. We will use an existing TensorFlow model that has been trained for image recognition so it can identify the object in a given image and classify it into one of 1000 different categories. This model follows the Google's Inception architecture, and has been trained on a popular academical dataset for image recognition called ImageNet. While you could train the model yourself, for example following the instructions from TensorFlow's official Github, you can also download a fully trained model file from one of Microsoft's examples here or from Google. (The important files are the **.pb** with the model and the **.txt** with labels.)

Let's begin to create our application. Create an empty solution, then add a console project named **ModelBuilder** to the solution. Later in the article, we will add a second project to the solution with the Blazor application.

Once the solution and project are created, we need to install several NuGet packages that will be used to build the model. Make sure to install all of:

- Microsoft.ML
- Microsoft.ML.ImageAnalytics
- Microsoft.ML.TensorFlow
- SciSharp.TensorFlow.Redist

Next, create a folder named **TFInceptionModel** and download the pre-trained TensorFlow model files from Microsoft's example.

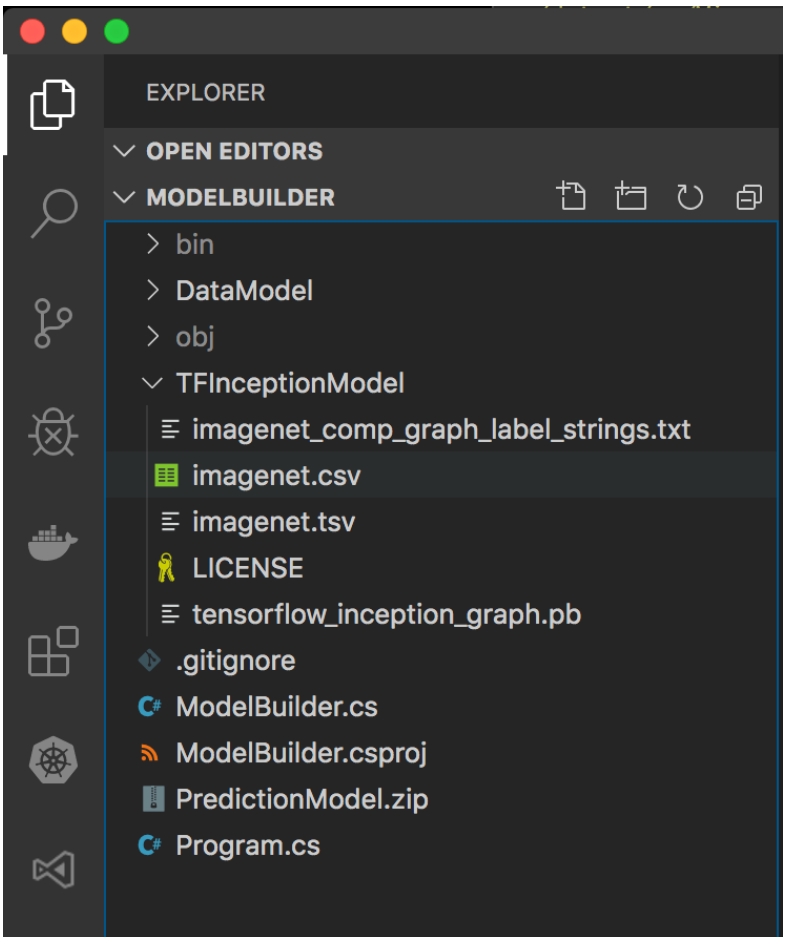


Figure 2, creating the ModelBuilder console application

As the final setup step, we will make sure that the working directory when launching the project is set to the project folder. This will allow us to reference folders and files relative to the project root in a way that works regardless whether the project is run with Visual Studio or `dotnet run` in the console (More info on this GitHub issue). Update the **ModelBuilder.csproj** file adding the following entry to the property group:

```
<RunWorkingDirectory>$(MSBuildProjectDirectory)</RunWorkingDirectory>
```

Let's finally begin to add some code. Start by adding a **DataModel** folder. Inside, create a new class `ImageInputData` with the following contents:

```
public class ImageInputData
{
    [ImageType(224, 224)]
    public Bitmap Image { get; set; }
}
```

This will be our entry to the model we will begin creating next, it simply tells ML.NET that we will use a bitmap image of 224x224 pixels. This matches the size used to train the downloaded TensorFlow model.

Next add a new `ModelBuilder` class to the project. This is where we will define the ML.NET model that will:

1. load a bitmap image
2. resize it as 224x224 and extract its pixels
3. run it through the downloaded TensorFlow model

Following ML.NET's API, we would define a pipeline with the loading and transformation steps, then train the model using a training dataset and finally evaluate its accuracy. Since we will be directly using the pre-trained TensorFlow model, we can skip the training and evaluation steps. The steps to build the model would then look like:

```
public ModelBuilder(string tensorFlowModelFilePath, string mlnetOutputZipFilePath)
{
  _tensorFlowModelFilePath = tensorFlowModelFilePath;
  _mlnetOutputZipFilePath = mlnetOutputZipFilePath;
}

public void Run()
{
  // Create new model context
  var mlContext = new MLContext();

  // Define the model pipeline:
  //    1. loading and resizing the image
  //    2. extracting image pixels
  //    3. running pre-trained TensorFlow model
  var pipeline = mlContext.Transforms.ResizeImages(
    outputColumnName: "input",
    imageWidth: 224,
    imageHeight: 224,
    inputColumnName: nameof(ImageInputData.Image)
  )
  .Append(mlContext.Transforms.ExtractPixels(
    outputColumnName: "input",
    interleavePixelColors: true,
    offsetImage: 117)
  )
  .Append(mlContext.Model.LoadTensorFlowModel(_tensorFlowModelFilePath)
    .ScoreTensorFlowModel(
    outputColumnNames: new[] { "softmax2" },
    inputColumnNames: new[] { "input" },
    addBatchDimensionInput: true));

    // Train the model
    // Since we are simply using a pre-trained TensorFlow model,
    // we can "train" it against an empty dataset
    var emptyTrainingSet =
        mlContext.Data.LoadFromEnumerable(new List<ImageInputData>());
    ITransformer mlModel = pipeline.Fit(emptyTrainingSet);

    // Save/persist the model to a .ZIP file
    // This will be loaded into a PredictionEnginePool by the
    // Blazor application, so it can classify new images
```

```
    mlContext.Model.Save(mlModel, null, _mlnetOutputZipFilePath);
}
```

As you can see, we are mostly resizing the images and extracting the pixels into the format used to train the TensorFlow model (Same size, offset and pixel interleave order), then we simply use the pre-trained TensorFlow model. If you are curious about the name of the input and output columns, these also need to match the names used by the TensorFlow model. You can check this out yourself by loading the .pb file into a TensorFlow explorer such as netron:

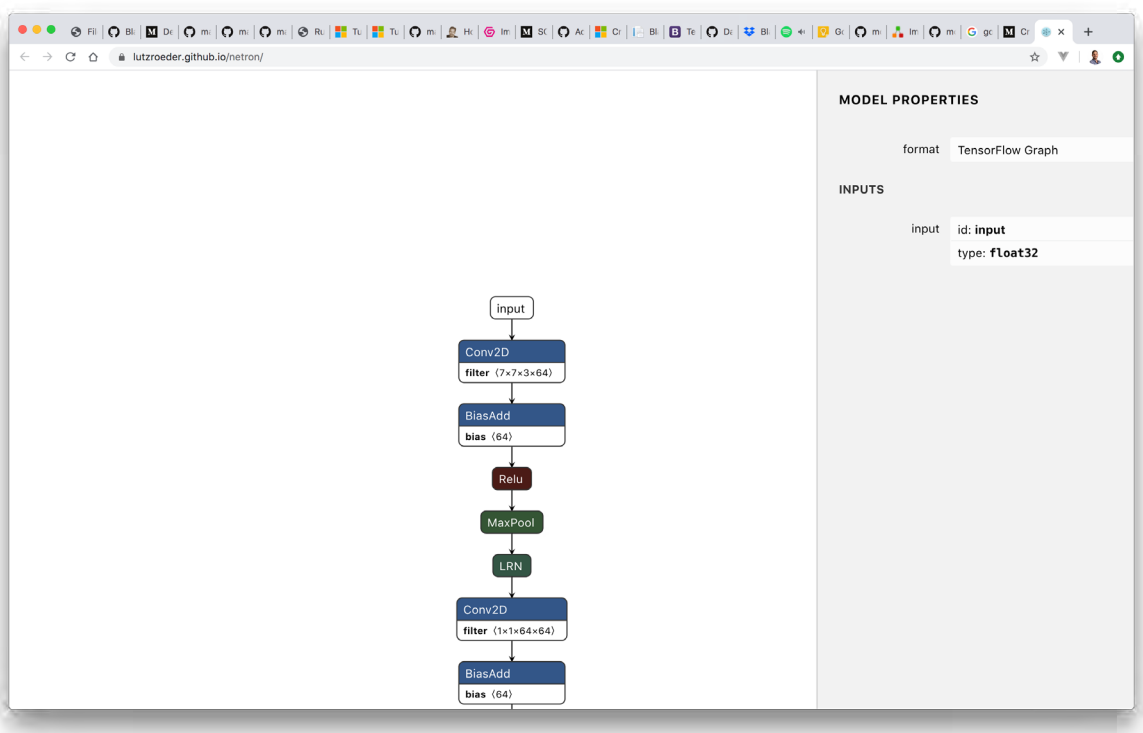


Figure 3, inspecting TensorFlow model with Netron to get the input name “input”

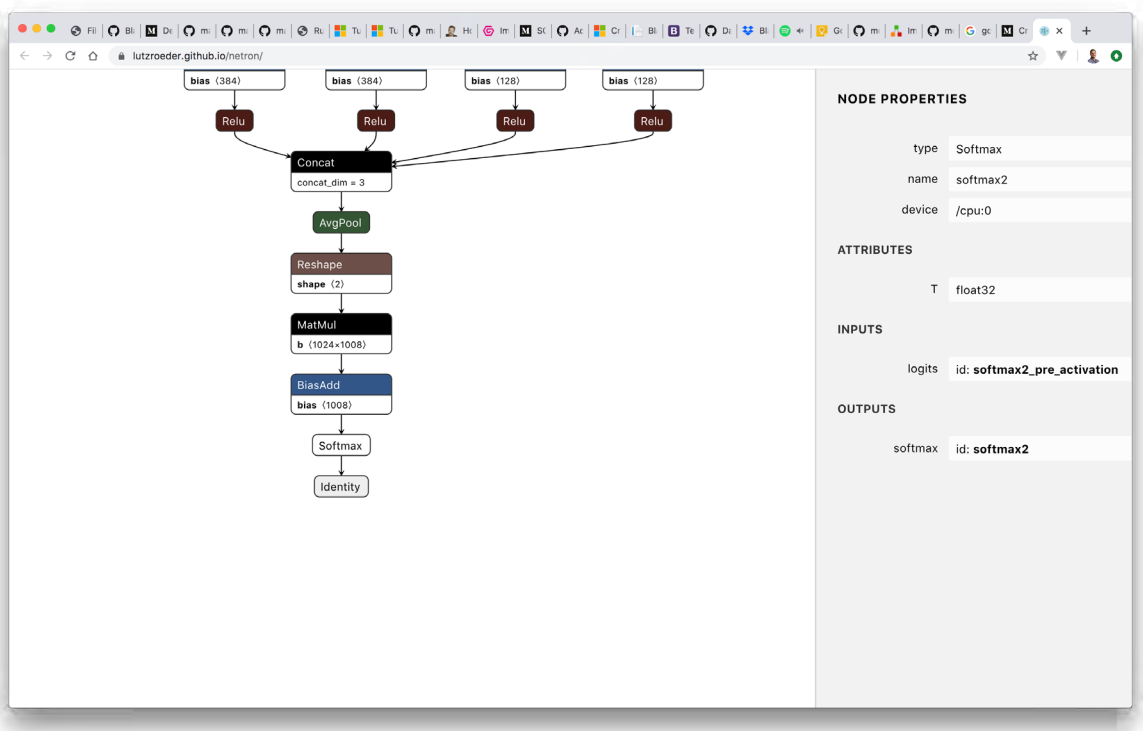


Figure 4, inspecting TensorFlow model with Netron to get the output name "softmax2"

All we have to do now is to modify **Program.cs** so it calls our `ModelBuilder` class in order to generate the model and save it to a .zip file:

```
static void Main(string[] args)
{
  var tensorFlowModelPath = "TFInceptionModel/tensorflow_inception_graph.pb";
  var mlnetOutputZipFilePath = "PredictionModel.zip";
  var modelBuilder = new ModelBuilder(tensorFlowModelPath, mlnetOutputZipFilePath);
  modelBuilder.Run();

  Console.WriteLine($"Generated {Path.GetFullPath(mlnetOutputZipFilePath)}");
}
```

You should now be able to build and run the application using either Visual Studio or `dotnet run`, which will generate a file named **PredictionModel.zip** at the root of the project folder.

USING THE ML.NET MODEL TO MAKE PREDICTIONS

So far, we have created a model and saved it to a .zip file, but we haven't used it yet to try and identify the object inside a given image. Let's see how we can use our model to make predictions!

To begin with, create a new folder **SampleImages** at the solution root and download the following images from Microsoft's example. We will ask the model to classify the first image broccoli.jpg:

Figure 5, a sample broccoli image to try our model

Next, create a new class `ImageLabelPredictions` inside the **DataModel** folder. This class simply represents the output of the model, the array of probabilities assigned to each of the 1000 labels the model was trained on (Remember we downloaded a txt file with the names of each of the labels):

```
public class ImageLabelPredictions
{
  [ColumnName("softmax2")]
  public float[] PredictedLabels { get; set; }
}
```

Now we can follow the instructions available in the ML.NET official docs that explains how to use the ML.NET model to make predictions. We will add some code at the end of the `ModelBuilder.Run` method.

Let's begin by loading the model from the zip file we just created:

```
DataViewSchema predictionPipelineSchema;
mlModel = mlContext.Model.Load(_mlnetOutputZipFilePath, out
predictionPipelineSchema);
```

Next we need to create a Prediction Engine from our trained model. This lets us classify one image at a time, which is enough for our purposes. Check the ML.NET docs for a batch mode example.

```
var predictionEngine = mlContext.Model
    .CreatePredictionEngine<ImageInputData, ImageLabelPredictions>(mlModel);
```

As you can see, the engine receives an `ImageInputData` instance as input and returns an `ImageLabelPredictions` output. In order to classify an image, we need to load the image into an instance of `ImageInputData` and call the `Predict` method of the engine:

```
var image = (Bitmap)Bitmap.FromFile("../SampleImages/broccoli.jpg");
var input = new ImageInputData{ Image = image };
var prediction = predictionEngine.Predict(input);
```

The output `prediction` variable is an instance of the `ImagePredictionLabel` class, containing an array of 1000 elements. Each element of the array represents the probability assigned by the model to each of the labels. We can then find the maximum probability and its associated label name:

```
var maxProbability = prediction.PredictedLabels.Max();
var labelIndex = prediction.PredictedLabels.AsSpan().IndexOf(maxProbability);
var allLabels = System.IO.File.ReadAllLines("TFInceptionModel/imagenet_comp_graph_
label_strings.txt");
var classifiedLabel = allLabels[labelIndex];
Console.WriteLine($"Test image broccoli.jpg predicted as '{classifiedLabel}' with
probability {100 * maxProbability}%");
```

That's it! If you build and run the project, you should see an output in the console similar to the following one:

```
$ dotnet run
...
Test image broccoli.jpg predicted as 'broccoli' with probability 99.99056%
Generated PredictionModel.zip
```

Using System.Drawing on Mac or Linux

The code above relies on `System.Drawing` in order to convert the image into a `Bitmap`. While `System.Drawing` is now part of .NET Core, and thus cross-platform, you may still need to install its dependencies.

More specifically, you will need to install the GDI+ libraries for your system. You can try the following commands:

```
# Linux
sudo apt install libc6-dev
sudo apt install libgdiplus
# Mac
brew install mono-libgdiplus
```

Integrating the ML.NET model in a Blazor website

So far, we have created a console application which creates a Machine Learning model and saves it as a zip file. ML.NET provides the necessary APIs to integrate the model in any application in order to make predictions with it. In this second half of the article, we will integrate the model within a Blazor application so users can upload images which will be identified using the previously generated model.

Begin by adding a new server-side Blazor project to the solution, named **BlazorClient**. Once generated, install the following NuGet packages. We will use them to upload files and to use the ML.NET model:

- Microsoft.Extensions.ML
- BlazorInputFile (It's in prerelease so it won't show in the Visual Studio NuGet unless you check "Include prerelease")

After installing BlazorInputFile, update the **_Host.cshtml** file located inside the **Pages** folder. You will need to add its JavaScript file right before the closing `</body>` tag:

```
<script src="_content/BlazorInputFile/inputfile.js"></script>
```

We will also need to add a reference to the **ModelBuilder** project. From the BlazorClient, we will use the `ImageInputData` and `ImagePredictionLabels` classes as the input/output of the model prediction engine respectively.

Deploy model and label files to the bin folder

In order to use the generated model, we will need to use both the zip and label files. ML.NET provides an API to load models both from the local file system or a remote network location. To keep things simple, we will load it from the local file system.

Update the **.csproj** file of the **ModelBuilder** project so both the generated zip model and labels file are copied to the output folder:

```
<ItemGroup>
  <None Update="PredictionModel.zip">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
  <None Update="TFInceptionModel\imagenet_comp_graph_label_strings.txt">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

After this change, whenever we build the BlazorClient, the model and labels will be copied to the output folder alongside the ModelBuilder DLL.

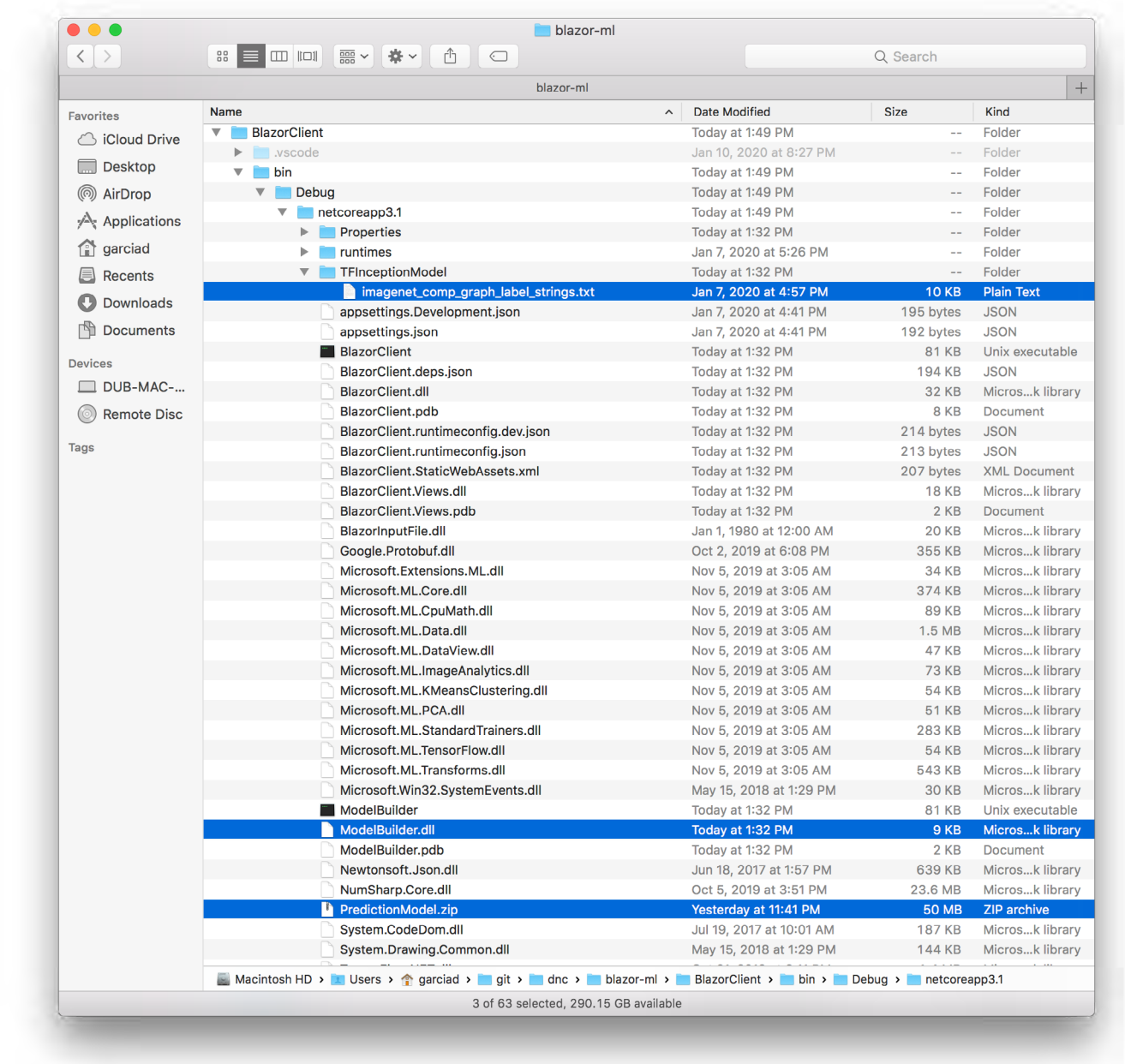


Figure 6, ModelBuilder DLL, model zip file and labels copied to the output folder

This way we will read both model and labels file from the local file system. The only thing we need is a little utility to get the full path to a file inside the current bin folder (as opposed to the entry folder where dotnet run was called). Create a small utility class `PathUtilities` since we will need it in a couple of places:

```
public class PathUtilities
{
  public static string GetPathFromBinFolder(string relativePath)
  {
    FileInfo _dataRoot = new FileInfo(typeof(Program).Assembly.Location);
    string assemblyFolderPath = _dataRoot.Directory.FullName;

    string fullPath = Path.Combine(assemblyFolderPath, relativePath);
    return fullPath;
  }
}
```

Setting up a Prediction Engine Pool

In the previous section we saw how we could create a Prediction Engine and use it to make predictions, identifying objects in new images. The approach worked fine while we had a console application and used the model with one image at a time.

However, the Prediction Engine **is not thread safe**. This means we need a slightly different approach in the context of a Blazor application, which is inherently multi-threaded like any other ASP.NET web application. Luckily, ML.NET already provides a thread safe solution designed to be used in the context of web applications and services, the Prediction Engine Pool.

Setting up the Prediction Engine Pool is very straightforward, with ML.NET providing the necessary dependency injection extensions. Add the following registration to the `ConfigureServices` method of the `Startup` class:

```
services.AddPredictionEnginePool<ImageInputData, ImageLabelPredictions>()
        .FromFile(PathUtilities.GetPathFromBinFolder("PredictionModel.zip"));
```

It is as simple as it looks. We register a Prediction Engine Pool by telling ML.NET where to find the zip file with the model and which specific classes are to be used as input/output when making predictions.

When registering the engine pool, ML.NET lets you load the model both from the local file system (using the `FromFile` extension method) or a network location (using the `FromUri` extension method). In both cases, optional parameters let you automatically reload the model whenever a new version of the model is published, either using a `FileSystemWatcher` or polling the network location.

Once registered, we can inject instances of `PredictionEnginePool<ImageInputData, ImageLabelPredictions>`, the class which exposes the `Predict` method that will let us identify images.

Creating a prediction service

Now that we have registered the `PredictionEnginePool`, we could directly use it from a Blazor page/component via the `@inject` directive. However, as we saw earlier when testing the generated model, there is a bit of boilerplate needed to use the `Predict` method. We need to load the image into a Bitmap, create the `ImageInputData` instance, call the `Predict` method and finally find the label with the highest probability and map it to the actual label name.

Let's instead create a service class where we can hide these details. Begin by creating a new class `ImageClassificationResult` inside the existing Data folder. Rather than using the raw model output, which contains an array of 1000 probabilities (one per label) which has to be interpreted, our service will return a simpler class with the name of the highest probability label and its actual probability:

```
public class ImageClassificationResult
{
    public string Label { get; set; }
    public float Probability { get; set; }
}
```

Now let's create the service class that encapsulates the boilerplate needed to run the model. Add an `ImageClassificationService` class also inside the Data folder with the following contents:

```
public ImageClassificationService(PredictionEnginePool<ImageInputData,
ImageLabelPredictions> predictionEnginePool)
{
  _predictionEnginePool = predictionEnginePool;
  // Read the labels from txt file available in the output bin folder
  string labelsFileLocation = PathUtilities.GetPathFromBinFolder(
    Path.Combine("TFInceptionModel", "imagenet_comp_graph_label_strings.txt"));
  _labels = System.IO.File.ReadAllLines(labelsFileLocation);
}

public ImageClassificationResult Classify(MemoryStream image)
{
  // Convert to image to Bitmap and load into an ImageInputData
  Bitmap bitmapImage = (Bitmap)Image.FromStream(image);
  ImageInputData imageInputData = new ImageInputData { Image = bitmapImage };

  // Run the model
  var imageLabelPredictions = _predictionEnginePool.Predict(imageInputData);

  // Find the label with the highest probability
  // and return the ImageClassificationResult instance
  float[] probabilities = imageLabelPredictions.PredictedLabels;
  var maxProbability = probabilities.Max();
  var maxProbabilityIndex = probabilities.AsSpan().IndexOf(maxProbability);
  return new ImageClassificationResult()
  {
    Label = _labels[maxProbabilityIndex],
    Probability = maxProbability
  };
}
```

The service receives the `PredictionEnginePool` through dependency injection. It also reads the labels file once as part of the constructor, so they are already loaded in memory by the time an image needs to be classified.

The implementation of the `Classify` method follows the very same steps we took when testing the generated model as part of the **ModelBuilder** project. The main difference is that we receive the image as an instance of `MemoryStream`, which is then transformed into a Bitmap using the System.Drawing utilities (with the system dependencies we already saw in the case of Mac and Linux). Using a `MemoryStream` will let us easily integrate with the code that will receive user uploaded images from a Blazor page.

Finally, let's register the service within the dependency injection container so it can be later injected into the Blazor pages/components. Add the following line to the `ConfigureServices` method of the `Startup` class:

```
services.AddSingleton<ImageClassificationService>();
```

Uploading images

It is time to start building the user interface using Blazor. Let's add a new Razor component named **Identify.razor** inside the Pages folder. This page will let users upload one or multiple images using the BlazorInputFile component, so if you haven't yet installed it using NuGet, do so now.

```
@page "/identify"
@using System.Collections.Generic
@using BlazorClient.Data
@using BlazorInputFile

<div class="container">
  <h1>Identify image</h1>

  <p>
    This component allows sending an image to run the image recognition model.
    Select an image to start the upload and recognition process.
  </p>
  <form>
    <InputFile multiple OnChange="OnImageFileSelected" accept="image/*"/>
  </form>

  <div class="row my-4">
    @foreach (var image in selectedImages)
    {
      <div class="col-4">
        <p>@image.Name</p>
      </div>
    }
  </div>
</div>


@code {
    List<IFileListEntry> selectedImages = new List<IFileListEntry>();

    void OnImageFileSelected(IFileListEntry[] files)
    {
        selectedImages.AddRange(files);
    }
}
```

You can also add a link to your new page in the left-hand menu of the website. Simply edit the **NavMenu.razor** file located inside the **Shared** folder, adding a new item that navigates to the `/identify` route associated with the page you just created:

```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="identify">
    <span class="oi oi-list-rich" aria-hidden="true"></span> Identify image
  </NavLink>
</li>
```

Right now, this results in a very uninteresting page that lets users select the image files, the names of which are then rendered in the page.

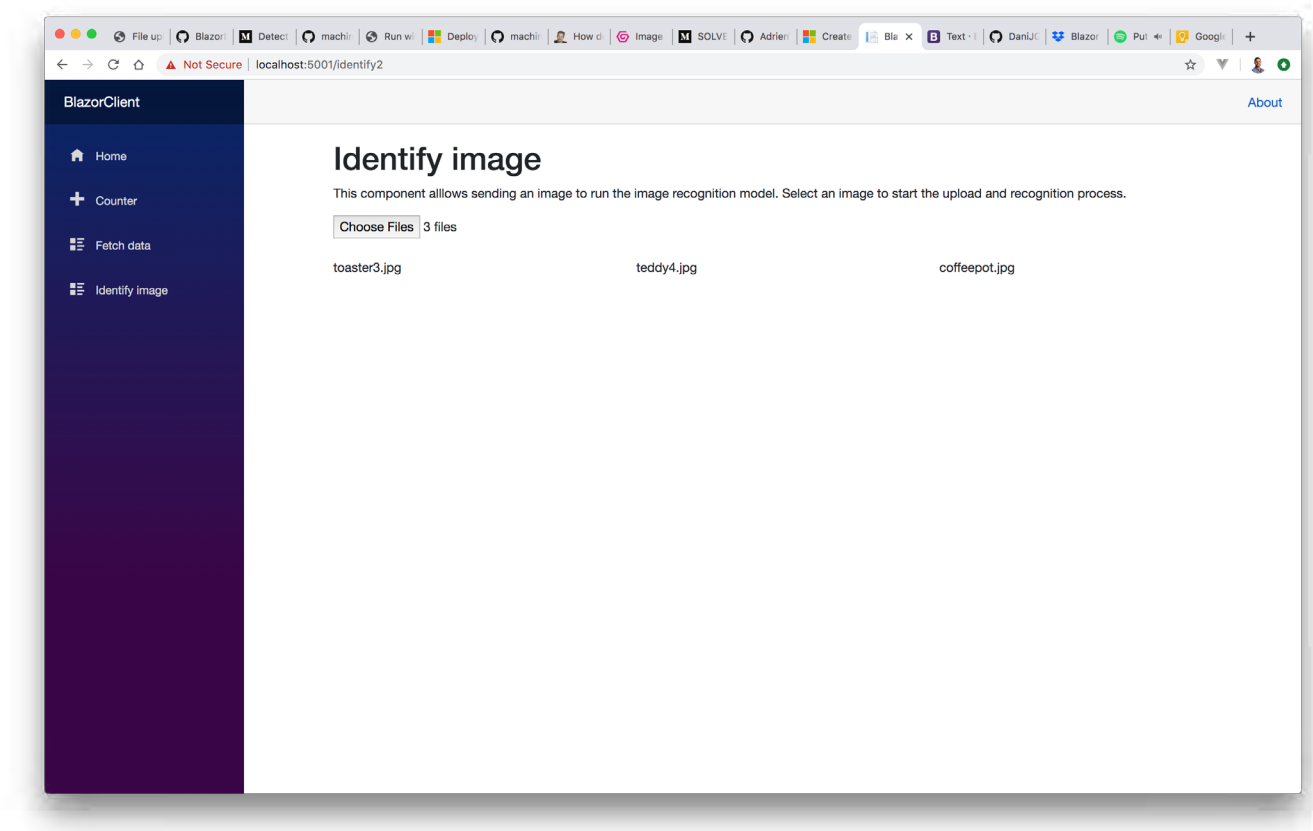


Figure 7, the first unimpressive iteration of our page

There are a few interesting details though!

As we already mentioned, we use the BlazorInputFile component for users to select files which will then be uploaded to the server.

Notice how we have added the `accept="image/*"` attribute to the `InputFile` component exposed by **BlazorInputFile**. While this isn't a property specifically allowed by the Blazor component, it uses a property dictionary in order to capture all the additional unknown properties which are then added directly to the HTML input element. Read more about this technique in the Blazor docs.

We have also added an event handler for its `OnChange` event, where we have access to an array of `IFileListEntry`. Each `IFileListEntry` instance contains information about an individual file selected by the user, including a Data property of type `Stream`. We will later use this property in order to upload the image to the server into a `MemoryStream` instance that can be then used with the `ImageClassificationService`. This piece, which we will be needing in a moment, looks as follows:

```
var file = files.FirstOrDefault();
var downloadedFileData = new MemoryStream();
await file.Data.CopyToAsync(downloadedFileData);
```

Let's now create a model class that identifies each selected image. This model will contain the code seen in the previous paragraph that lets us upload the image from the `IFileListEntry` into a `MemoryStream`, as well as a property with the classification result.

Note: *Apart from keeping the page/component code cleaner, using a specific model class will let us extract/process the data we need from the `IFileListEntry`. We will later see how this will come really handy as we add a button for users to remove one of the uploaded files, which can cause Blazor to destroy and re-initialize*

some components.

Add a new class `SelectedImage` inside the **Data** folder:

```
public class SelectedImage
{
  private IFileListEntry _file;
  public ImageClassificationResult ClassificationResult { get; set; }
  public string Name => _file.Name;

  public SelectedImage(IFileListEntry file)
  {
    _file = file;
  }

  public async Task<MemoryStream> Upload()
  {
    var fileStream = new MemoryStream();
    await _file.Data.CopyToAsync(fileStream);
    return fileStream;
  }
}
```

Now update the Blazor page so it stores a list of `SelectedImage` instead of `IFileListEntry`:

```
List<SelectedImage> selectedImages = new List<SelectedImage>();

void OnImageFileSelected(IFileListEntry[] files)
{
  selectedImages.AddRange(
    files.Select(f => new SelectedImage(f)));
}
```

Creating a child component for individual images

We are now ready to create a specific Blazor component that will render an individual image and its ML.NET classification result. This will follow a classic parent-child relationship between the previous **Identify.razor** page and the new child component.

Add a new **IdentifyImage.razor** component inside the **Shared** folder. For now, the component will receive a `SelectedImage` as parameter and will render a bootstrap card (The Blazor project template comes with Bootstrap4 CSS framework preconfigured) with the name of the file:

```
@using BlazorClient.Data

<div class="card mb-2">
  <div class="card-header">
    @Image.Name
  </div>
  <div class="card-body">
    We will render here the classification result
  </div>
</div>

@code {
  [Parameter]
```

```
    public SelectedImage Image { get; set; }
}
```

This way we can update the template of the parent **Identify.razor** page so it renders this component for each of the images:

```
@foreach (var image in selectedImages)
{
  <div class="col-4">
    <IdentifyImage Image="image" />
  </div>
}
```

You should see something like the following screenshot:

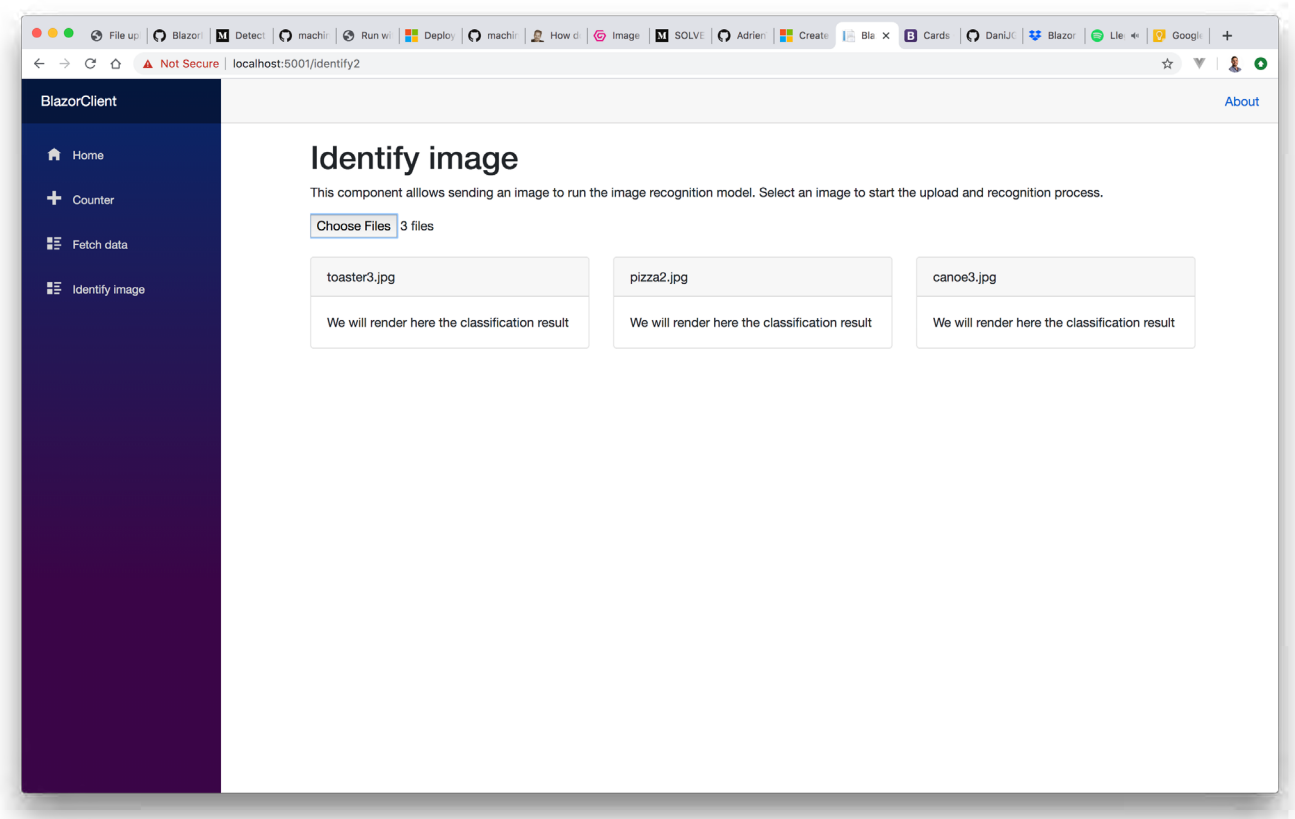


Figure 8, the page now using a child component for each individual file

Let's now allow users to remove one of the uploaded images. This is a good excuse to study how a child can communicate back to its parent, since the child **IdentifyImage.razor** component needs to let the parent **Identify.razor** page know which image should be removed.

Update the **IdentifyImage.razor** component with a new `EventCallback` parameter and a method to trigger it:

```
[Parameter]
public EventCallback<SelectedImage> OnClear { get; set; }

async Task TriggerOnClear()
{
    await OnClear.InvokeAsync(Image);
}
```

Now update the component template with a button whose click event will execute the `TriggerOnClear` method:

```
<p class="card-text">
  <button class="btn btn-secondary" @onclick="TriggerOnClear">Clear</button>
</p>
```

The changes in the child **IdentifyImage.razor** component are finished. Now we need to listen to the `EventCallback` in the parent component, which should remove the image from the list of selected images.

First add a new method to be triggered by the callback:

```
void OnClear(SelectedImage image)
{
  selectedImages.Remove(image);
}
```

Then update the template in order to bind the method to the EventCallback.

```
<IdentifyImage @key="image" Image="image" OnClear="OnClear" />
```

Notice the usage of the `@key` special directive. This directive is critical for Blazor to be able to minimize the work needed when removing elements, as described in its official docs.

Note: *Even with this directive, I have been surprised by Blazor recreating component instances which were not affected. For example, given a list of 3 images, removing the second image will result on the component for the 3rd image to be removed and recreated! Make sure to take this into account when relying on component lifecycle events.*

If you build and run, the page should look similar to the following screenshots, where users are able to upload multiple files, which can later be removed:

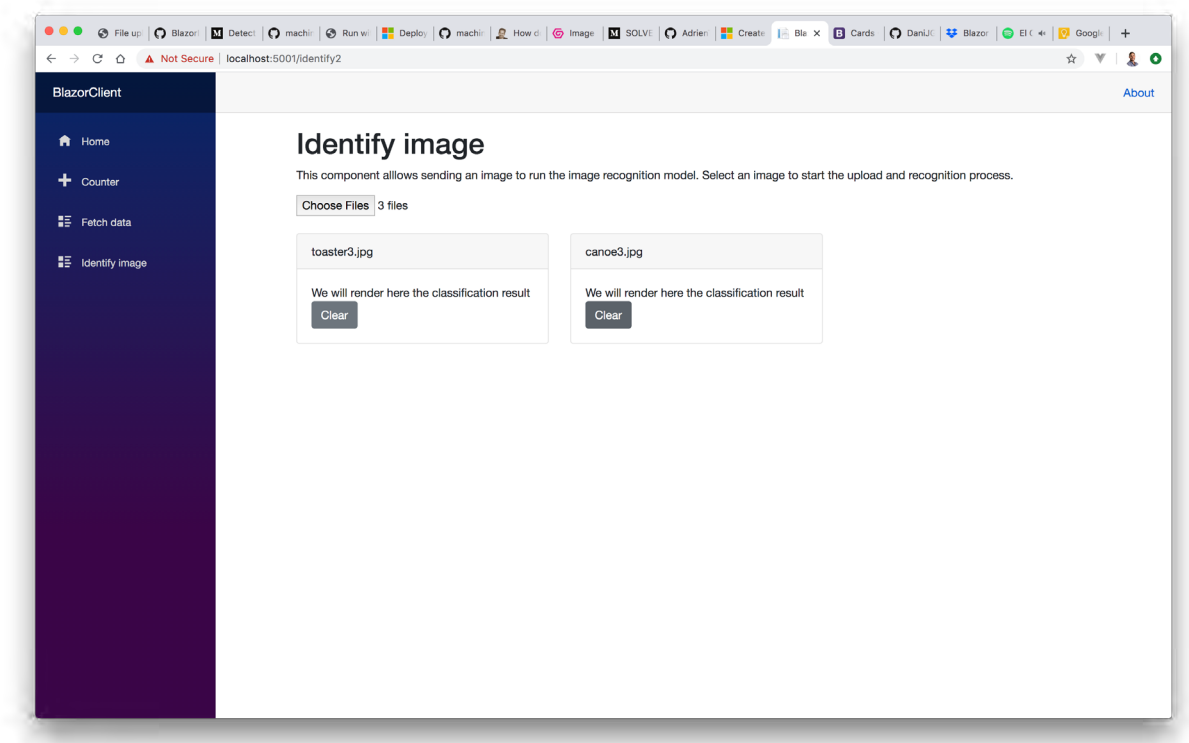


Figure 9, updated page and component allowing users to remove uploaded images

Using the Classification service from the individual image component

It is now time to join all the pieces and run the image classification model for each of the uploaded images. All we need to do is:

- Inject the `ImageClassificationService` into the image component
- Upload the image into a `MemoryStream`, using the Upload method of the `SelectedImage` class
- Run the `Classify` method of the service, rendering the result in the card body

Begin by injecting the service into the **IdentifyImage.razor** component:

```
@inject ImageClassificationService ClassificationService
```

Then we will run the image classification during the component initialization, one of its lifecycle events. We will store the result in the `ClassificationResult` property of the `SelectedImage` class.:

```
protected override async Task OnInitializedAsync()
{
  if (Image.ClassificationResult != null) return;
  using(var fileStream = await Image.Upload())
  {
    Image.ClassificationResult = ClassificationService.Classify(fileStream);
  }
}
```

Notice how we do nothing when the image has already been classified. This prevents us from re-running the model if Blazor recreates the component.

Finally, update the component template in order to render the classification result. In case the result isn't ready, we will render some text letting the user know that the upload and classification process is in progress. Add the following elements inside the `card-body` element of the template:

```
<p class="card-text text-center my-2">
  @if (Image.ClassificationResult != null)
  {
    @:Classified as <strong>@Image.ClassificationResult.Label</strong>
    @:with <strong>@Image.ClassificationResult.Probability</strong> probability
  }
  else
  {
    <em>Processing...</em>
  }
</p>
```

Rebuild and run the Blazor application. You should finally be able to upload and classify images, as in the following screenshot:

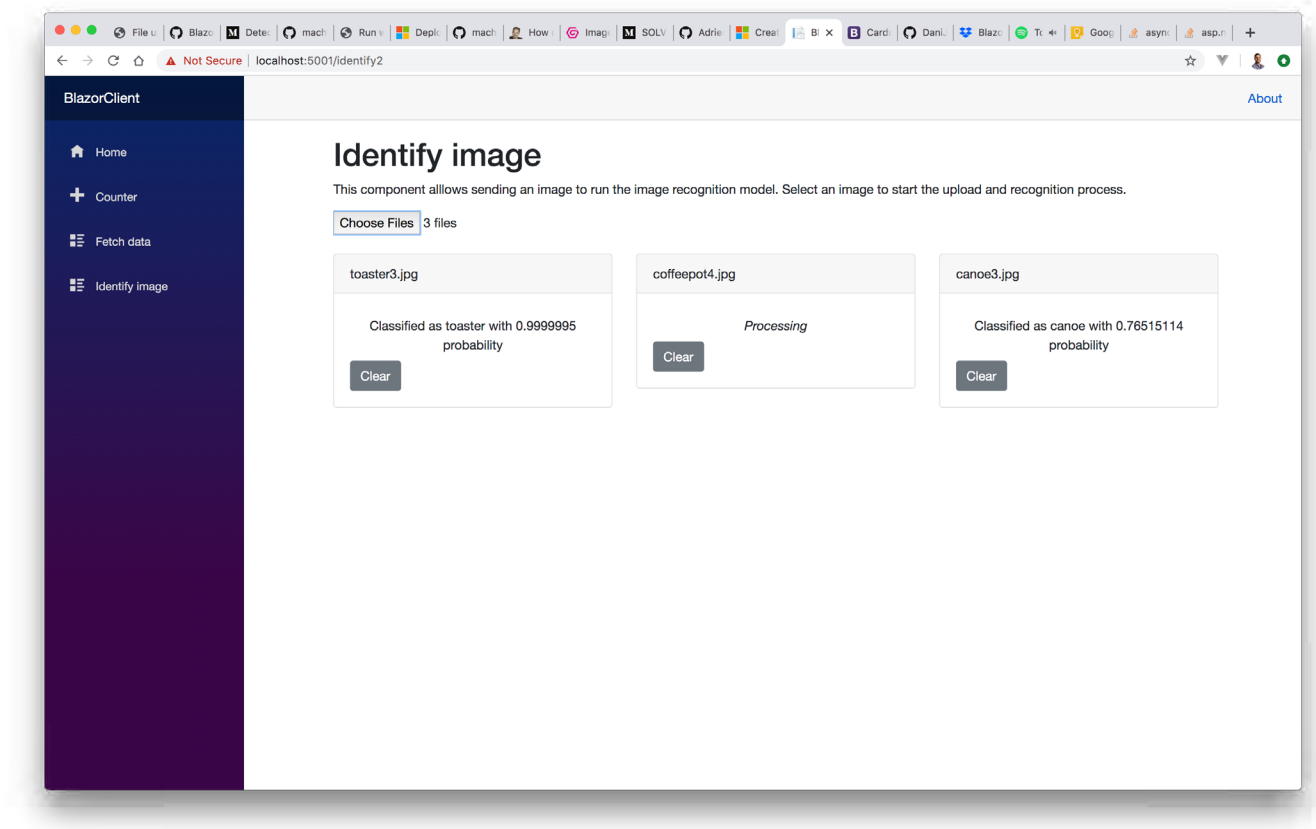


Figure 10, classifying uploaded images

Polishing the UX by rendering each image and adding and upload progress indicator

While the app is perfectly functional, the UX is very rough. Fortunately, we can improve it with some simple changes.

First, we can render the uploaded image, which will make the UX much nicer. We can update the `SelectedImage` class so we capture the base64 string of the uploaded image. This way, we will be able to add an HTML `img` element that renders said base64 string.

```
public string Base64Image { get; private set; }

public async Task<MemoryStream> Upload()
{
  ... earlier method contents ...

  // Get a base64 so we can render an image preview
  Base64Image = Convert.ToBase64String(fileStream.ToArray());
  return fileStream;
}
```

Now update the template of the **IdentifyImage.razor** component and add the following `img` element right before the `card-body` element:

```
@if (Image.Base64Image != null)
{
  <img src=" @Image.Base64Image" class="card-img-top" alt="@
Image.Name">
}
```

That's it, the uploaded images will now be rendered as well.

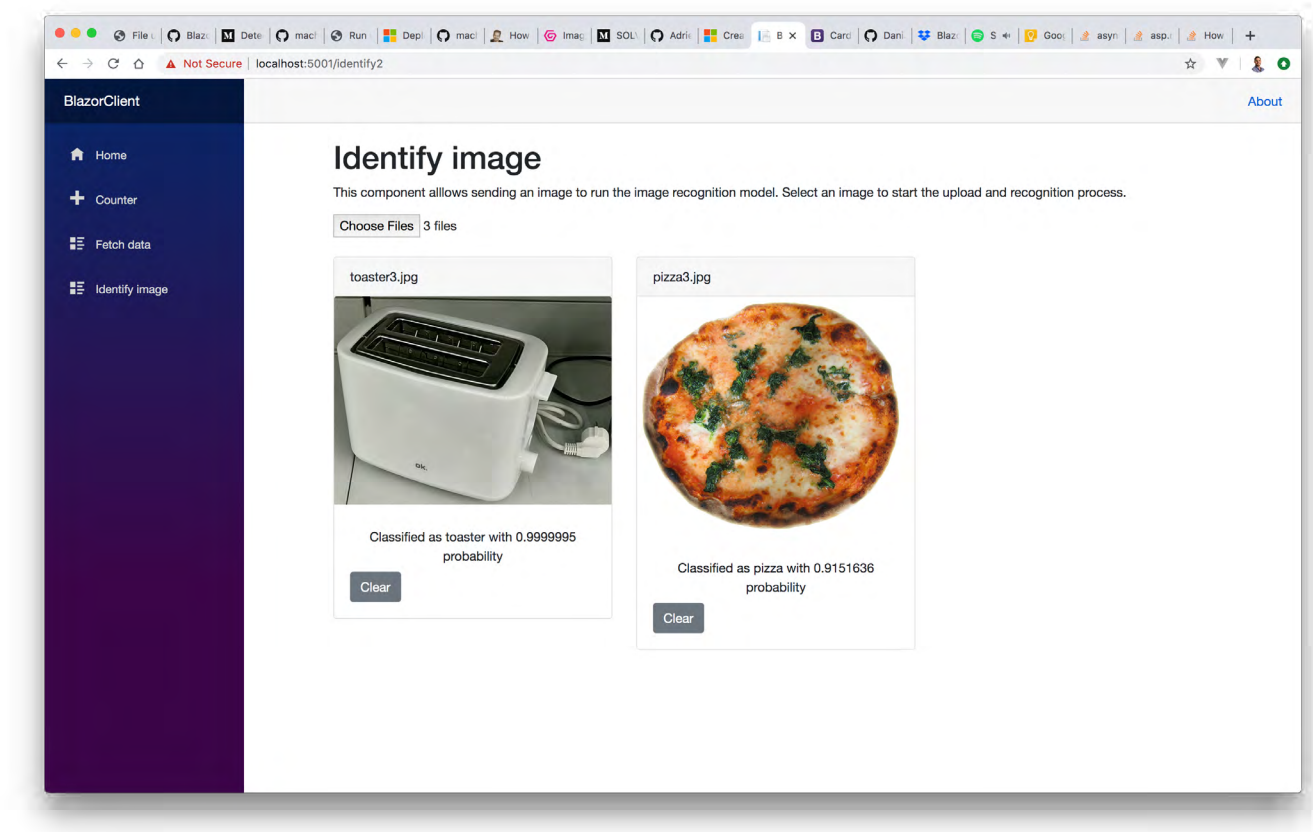


Figure 11, rendering uploaded images

The final improvement we will make is to show a Bootstrap progress bar that gives the user feedback on the upload progress.

Update the `SelectedImage` class with a new read-only property that returns the uploaded percentage by calculating how many of the total file bytes have been read so far:

```
public double UploadedPercentage => 100.0 * _file.Data.Position / _file.Size;
```

Next update the template of the `IdentifyImage` component so it renders the progress bar using the calculated percentage:

```
<div class="progress">
  <div class="progress-bar" role="progressbar"
   style="width: @Image.UploadedPercentage%"
   aria-valuenow="@Image.UploadedPercentage"
   aria-valuemin="0"
   aria-valuemax="100"/>
</div>
```

However, if you build and run the project, you will notice that the bar is stuck at 0 during a file upload! That is because Blazor doesn't know it needs to update the UX during the file upload. In cases like this, we need to notify Blazor that component dependent state has changed by invoking the StateHasChanged method.

Update the `Upload` method of the `SelectedImage` class so we can provide an Action to be invoked to report progress:

```
public async Task<MemoryStream> Upload(Action OnDataRead)
{
  EventHandler eventHandler = (sender, eventArgs) => OnDataRead();
  _file.OnDataRead += eventHandler;

  ... existing code ...

  _file.OnDataRead -= eventHandler;
  return fileStream;
}
```

Then update the `OnInitializedAsync` method of the **IdentifyImage.razor** component so it invokes the `StateHasChanged` method during the upload process:

```
await Image.Upload(() => InvokeAsync(StateHasChanged))
```

This way, Blazor now has to update the UX during the upload process. If you build and run the project, you should finally see a working progress bar.

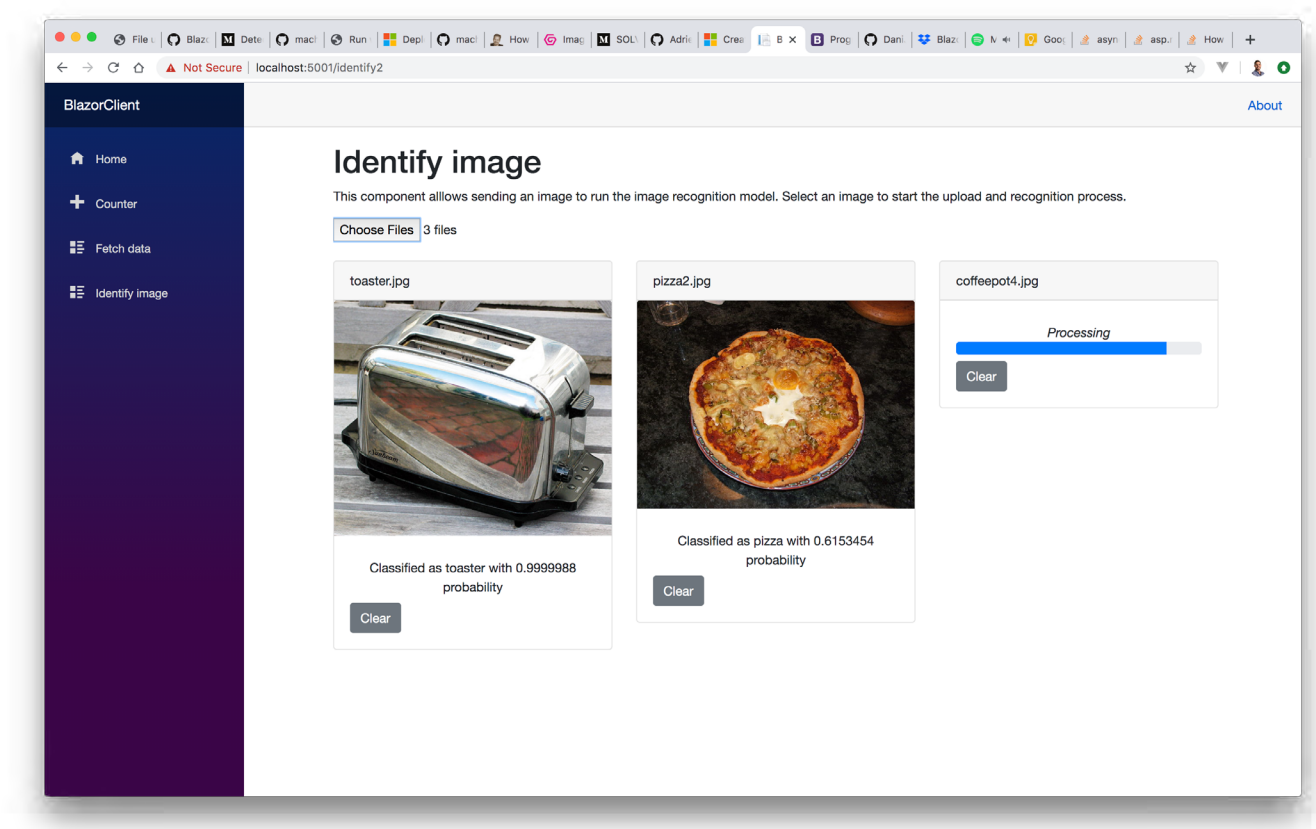


Figure 12, giving feedback on the upload process with a progress bar

Conclusion

It is a very interesting time to be a .NET developer. Thanks to Blazor and ML.NET, we have seen how leveraging your current C# and .NET skills, it is now possible to build a SPA web application and a Machine Learning model that lets you identify objects in images.

Microsoft seems really excited about both technologies, not without a reason. These were areas traditionally outside the scope of .NET developers, who at best had to master additional skills, and at worse would avoid them.

While currently they are not as fully featured as established SPA and Machine Learning solutions, they are already perfectly functional and have been designed so they are extensible and compatible with existing technologies.

I can't wait to see how far Microsoft and communities like ours, can push them!

• • • • • • •

Download the entire source code from GitHub at
bit.ly/dncm46-blazorml

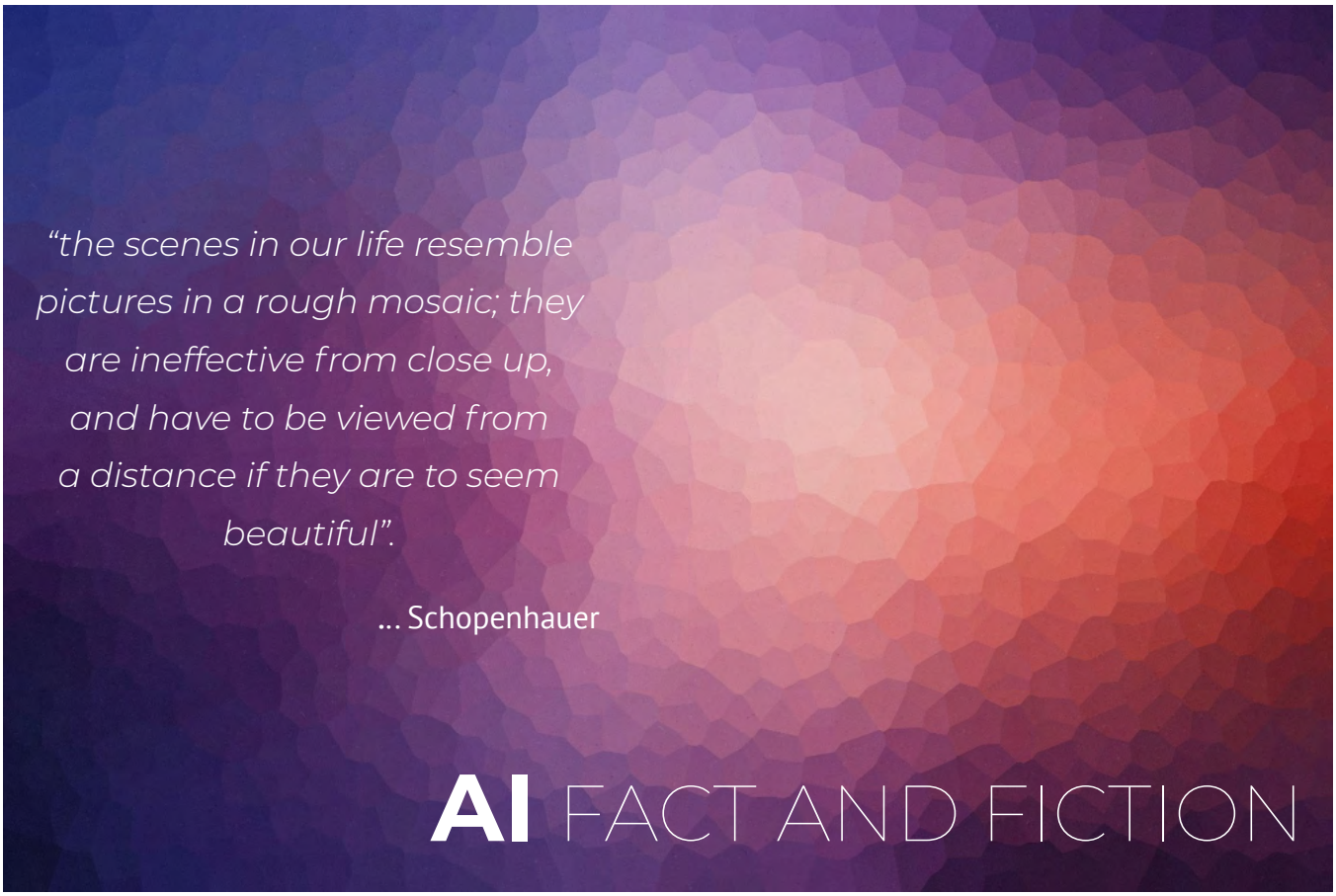
Daniel Jimenez Garcia

Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

*Thanks to **Damir Arh** for reviewing this article.*

Benjamin Jakobus



As beautiful as this quote is, when it comes to artificial intelligence, the opposite is true.

The mosaic seems rough and simple from far away. Only by moving up closer, do we see the intricacies and complexity of the individual stones that make up the bigger picture.

Maybe it was the treacherous illusion of simplicity, caused by standing far away from the mosaic, that gave rise to the exorbitant speculations about Artificial Intelligence (AI) back in the 40s and 50s.

Experts and IT moguls predicted that, within the span of a few years,

- computers would pass the Turing test,
- would be able to learn and use human language,
- and even become sentient.

Pundits and professionals alike, have since predicted

- the end of spam,
- the emergence of humanoids,
- mass unemployment due to automation or the rise of the singularity.

Yet, the more advances we make in the field of computing, the more obvious it becomes just how far away we are from achieving even the simplest of these wild "predictions".

Even if you have only ever dabbled your toes in programming, you will know that even the most "basic" of problems that humans face today, such as arriving at simple decisions or finding the shortest path between two points on a map, can become very complex very quickly.

The complexities that make a technical solution difficult, tend to only surface as we examine the problem in detail. Hence my reference to Schopenhauer's mosaic: *the individual problems that our human brain faces every single day* are infinitely more detailed and nuanced than we can see from afar.

The devil always lies in the detail.

This is precisely the reason as to why I am not worried about many of the scary predictions that we hear in the media today.

An all-powerful artificial intelligence won't become our overlord within the next few lifetimes, and we are still a long way away from producing machines that can compete with us on all levels of human existence. Sure, computing and artificial intelligence have made tremendous advances over the past decades and changed both industry and society at a fundamental level. But all of these changes still face the same constraints than they did when the digital computer was first invented.

Our processors have become faster since then, and the chip has become smaller, migrating into every pocket. We have therefore managed to apply technology to solve a wider range of problems and can use the technology in different ways. That is, our computer is no longer constrained to cracking Nazi submarine codes, or calculating the trajectory of ballistic missiles.

But the underlying theory and foundations of computing haven't changed since the 40s.

The likes of Google, Amazon and Microsoft still use machine learning techniques that were invented during the 50s, 60s and 70s. These techniques work better today only because we have access to more data, and can store and process this data. But they still face the same problems than they have before: the techniques apply only to very domain-specific problems.

At the same time, our understanding of the world, of intelligence and of how we look at and solve problems, hasn't drastically changed since the last major scientific breakthroughs of the 19th and 20th century.

In his book "Homo Deus", Noah Yuval Harari eloquently pointed out that humanity tends to describe and view the world in terms of the prominent technology of its age. When the steam engine was invented, we looked at the world in very mechanical terms. Biologists and doctors described the human body using terms such as "valves" and "pressure".

Today, everything is an algorithm or a computer program.

The human mind has become a "software"; our body is “the hardware”; by cooking using a recipe we are "following an algorithm", and suicide is no longer against our "nature" but "against our programming". We view the world through one lens, until we have exhausted all the possible problems that we can solve this way. Then the next dominant technology will come along, and we throw ourselves into its current, letting it carry our thought processes along like a log in a big river.

This is a natural process, and indeed is what drives progress. However, it would be foolish to think that, just because we view the world in the context of the digital computer, we will be able to solve all problems using it and create digital gods.

This is because some processes simply don’t lend themselves well to machine learning, statistical analysis or computing in general. Unless we make fundamental advances in the way we deconstruct problems and

look at the world, many problems will still remain outside the domain of completely autonomous artificial intelligences.

The AI Bubble

The recent increase of the use of artificial intelligence by tech giants such as Google is largely due to the massive expansion of the internet, and the emergence of mobile devices that allow us to remain connected 24/7.

With the generation of massive amounts of data, well-known machine learning techniques can suddenly be used in completely new ways. From recommending songs for us to listen to, to telling us what books we might like, notifying our homes of intruders, detecting weapons at airports or automatically adding entries to our online calendars, AI systems have simplified our lives, produced large amounts of wealth, but also changed the way we think, communicate and act.

And whilst there still exists much room for the expansion of artificial intelligence, its rise in popularity also created a new bubble in whose midst we now find ourselves.

Just like the possibility of the wide-spread adoption of the internet in the 1990s and early 2000s gave rise to the dot com bubble (during which CEOs of tech companies over-promised and under-delivered and analysts over-valued over-rated companies), the increased popularity of artificial intelligence has resulted in what an "AI bubble".

Whether you go to University research laboratories or talk a walk down any major city's business district, everybody is trying to cash in on the promise that "AI will make the world a better place". Academics fall

over themselves to try and fit the words "artificial intelligence", "machine learning" or "smart systems" into their government funding applications or research proposals. Startup founders promise to "automate" and "optimize" away your daily pains and worries, and established conglomerates move towards "smarter solutions".

Most of this is smoke and mirrors. Vaporware sold to customers.

Therefore, before buying a new "AI-powered product" or reading articles about "smart systems", ask yourself whether the problem in question is well suited for AI.

Problems that are effectively solved by AI often involve categorizing or classifying things, searching large amounts of data quickly or building up profiles of people or organizations by aggregating, cleaning and analyzing large amounts of data. AI techniques are also great for solving tricky problems such as scheduling or pathfinding.

And almost always, one requires data. Data that is well-structured and unambiguous along with a problem definition must be concise with specific objectives.

Benjamin

Author

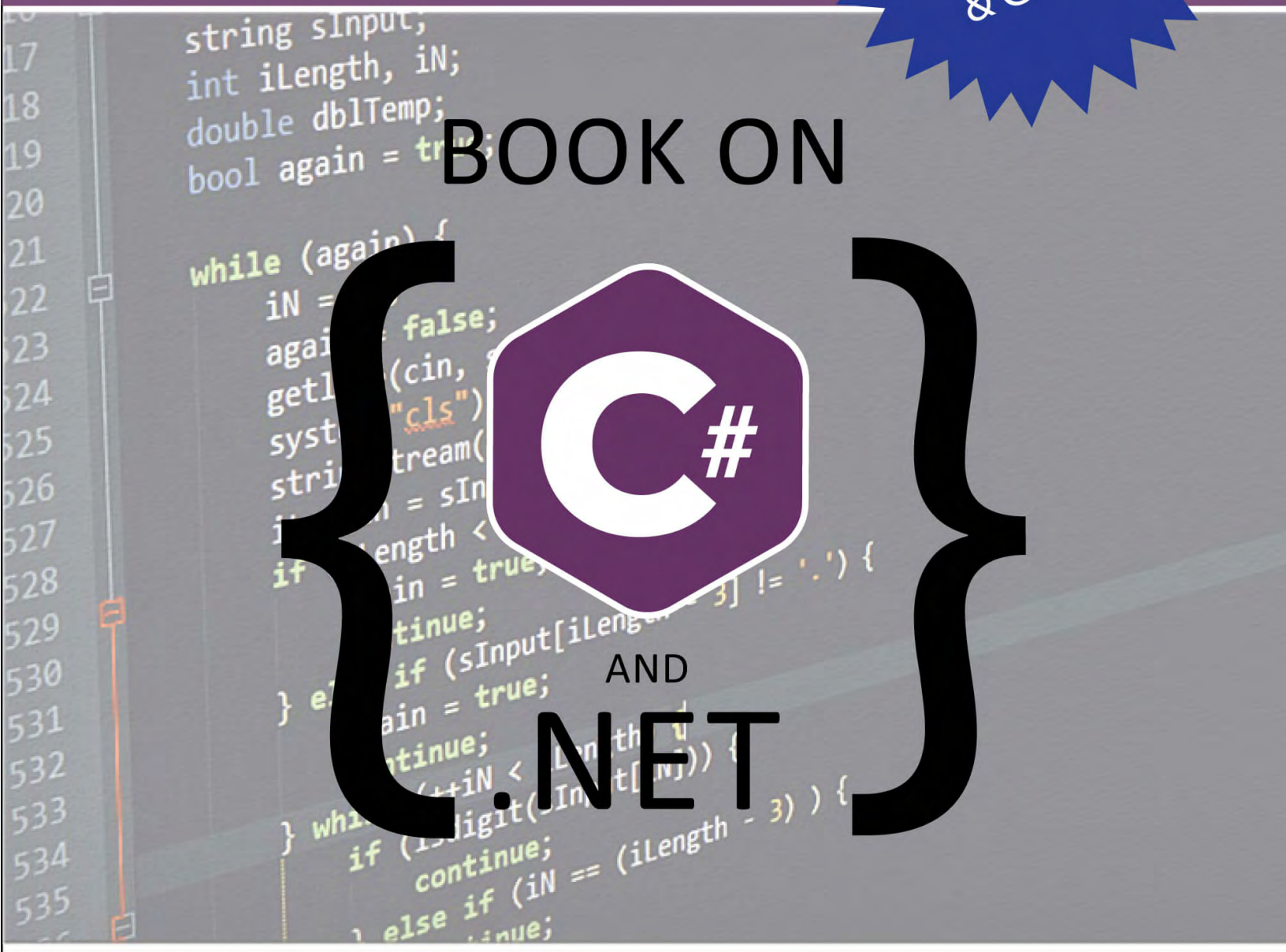
Benjamin Jakobus is a senior software engineer based in Rio de Janeiro. He graduated with a BSc in Computer Science from University College Cork and obtained an MSc in Advanced Computing from Imperial College, London. For over 10 years he has worked on a wide range of products across Europe, the United States and Brazil. You can connect with him on LinkedIn

Thanks to ***Suprotim Agarwal*** *for reviewing this article.*

COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY AWESOME BOOK ON { C# AND .NET }

Includes .NET Core 3.0 & C# 8.0



DAMIR ARH

ORDER NOW

Ravi Kiran

This tutorial will state the importance of unit testing Angular services. It also explains the process of unit testing services, HTTP calls and HTTP interceptors in an Angular application.

UNIT TESTING ANGULAR SERVICES, HTTP CALLS AND HTTP INTERCEPTORS

Angular services contain UI-independent reusable business logic of the application.

This logic could be used at multiple places in the application - say to receive or calculate data to be shown on the page. So, it is very important to make sure that the logic in the services is correct, or else this could result in issues at multiple places in the application.

Unit tests can be used to test the services by invoking the functionality directly.

As discussed in a previous article, unit testing can be used to invoke and test the behavior of a piece of code in isolation. The reusable logic written in services requires this kind of testing, as unit testing provides ways to test all possible scenarios by sending different types of data to the service methods.

Also, most applications use services to communicate with the backend APIs. It is important to make sure that the calls to these services are made correctly and their responses are correctly handled in the application. Unit tests help in checking the correctness in these calls.

Angular framework includes a testing module to test the API calls by providing mock responses. This setup can be used to effectively test whether the right set of APIs is called with correct parameters, and then test how the success and failures of the APIs are handled.

This tutorial will provide you with **enough knowledge on setting up a test file to unit test a service. And then it will show how the calls to backend APIs can be unit tested**.

Testing Services

The required setup to test any piece of Angular is already included with Angular CLI. My previous article on testing Angular component goes through details of the setup and explains it. Let's see how services can be tested by taking a simple example.

Consider the following service:

```
@Injectable({
  providedIn: "root"
})
export class CalculationsService {
  add(a: number, b: number): number {
    return a + b;
  }
}
```

This service is fairly easy to test, as it doesn't have any dependencies, and the logic executed in the *add* method, is adding two numbers. We need to perform the following tasks to test this service:

- Get an object of the service
- Call the methods to test
- Assert the results

The following code snippet gets an object of the `CalculationsService`:

```
describe('CalculationsService tests', () => {
```

```
  let calculationsSvc: CalculationsService;

  beforeEach(inject(
    [CalculationsService],
    (calcService: CalculationsService) => {
      calculationsSvc = calcService;
    }
  ));
});
```

The one thing to notice here is that we didn't add the *TestBed* setup here. We didn't do this as `CalculationsService` is provided in the root injector. Otherwise, if the service is provided in a module or in a component, we need to provide the service in the testing module configured with *TestBed*. The `beforeEach` block gets object of the service from the root injector. Now this object can be used to call the `add` method and test it.

The following snippet tests the `add` method:

```
it("should add two numbers", () => {
  let result = calculationsSvc.add(2, 3);
  expect(result).toEqual(5);
});
```

Testing a Service with HttpClient

A service with dependencies requires some more amount of setup for testing.

As unit testing is the technique for testing a piece of code in isolation, the dependencies of the service have to be mocked so the dependency doesn't become an obstacle while testing. One of the most common usages of the services is to interact with the backend APIs. It is needless to say that Angular applications use *HttpClient* to call the APIs, and Angular provides a mock implementation of this service to make it easier for the users of *HttpClient* to unit test their code.

Let's write unit tests for the `DataAccessService` used in the article Getting Started with HTTP Client. Here is the complete code of the service:

```
import { Injectable } from "@angular/core";
import { Traveller } from "./traveller";
import { HttpClient, HttpErrorResponse } from "@angular/common/http";
import { catchError } from "rxjs/operators";

import { Observable, throwError } from "rxjs";

const DATA_ACCESS_PREFIX: string = "api/travellers";

@Injectable({
  providedIn: 'root'
})
export class DataAccessService {
  constructor(private client: HttpClient) {}

  getTravellers(): Observable<Traveller[]> {
    return this.client.get<Traveller[]>(`${DATA_ACCESS_PREFIX}`).pipe(
      catchError((error: HttpErrorResponse) => {
        return throwError(
```

```
          `Error retrieving travellers data. ${error.statusText || "Unknown"} `
        );
      })
    );
  }

  deleteTraveller(id: number): Observable<any> {
    return this.client.delete<Traveller>(`${DATA_ACCESS_PREFIX}/${id}`).pipe(
      catchError((error: HttpErrorResponse) => {
        return throwError(
          `Error deleting travellers data. ${error.statusText || "Unknown"} `
        );
      })
    );
  }

  createTraveller(traveller: Traveller) {
    return this.client.post(`${DATA_ACCESS_PREFIX}`, traveller);
  }

  updateTraveller(traveller: Traveller, id: number) {
    return this.client.patch(`${DATA_ACCESS_PREFIX}/${id}`, traveller);
  }
}
```

As we can see, this service performs CRUD operations on a list of travelers that are made available through REST APIs. Let's set the environment for testing this.

The following snippet does this:

```
import { TestBed, inject } from "@angular/core/testing";
import {
  HttpClientTestingModule,
  HttpTestingController
} from "@angular/common/http/testing";

import { DataAccessService } from "./data-access.service";
import { Traveller } from "./traveller";

describe("DataAccessService", () => {
  let httpTestingController: HttpTestingController;
  let dataAccessService: DataAccessService;
  let baseUrl = "api/travellers";
  let traveller: Traveller;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule]
    });

    httpTestingController = TestBed.get(HttpTestingController);
    traveller = {
      id: 2,
      firstName: "John",
      lastName: "Kelly",
      city: "Boston",
      country: "USA",
      age: 18
    };
  });
```

```
  beforeEach(inject(
    [DataAccessService],
    (service: DataAccessService) => {
      dataAccessService = service;
    }
  ));
});
```

The above snippet does the following tasks:

- Imports the required objects. The `HttpClientTestingModule` and `HttpTestingController` are vital for testing the behavior of *HttpClient*.

  o The `HttpClientTestingModule` includes a mock implementation of `HttpClient` service, which doesn't make the actual XHR calls, instead it provides a way to inspect the calls attempted

  o The `HttpTestingController` provides APIs to make sure that the HTTP calls are made, to provide mock response to the calls and to flush the requests, so that the subscribers of the observables would be invoked

- Configures the testing module by importing the `HttpClientTestingModule` and gets the object of `HttpTestingController`
- Creates a mock traveler object which will be used in the tests
- Gets object of the `DataAccessService`

Now we have everything required to test the service. Let's write a test to check the correctness of `getTravellers` method.

```
it("should return data", () => {
  let result: Traveller[];
  dataAccessService.getTravellers().subscribe(t => {
    result = t;
  });
  const req = httpTestingController.expectOne({
    method: "GET",
    url: baseUrl
  });

  req.flush([traveller]);

  expect(result[0]).toEqual(traveller);
});
```

The above test calls the `getTravellers` method and expects a GET call to be made to the `baseUrl`. Then it flushes the request with the data to be returned. This is when the call is completed and the `subscribe` method is called. At the end, it inspects if the request returned the correct data.

The `getTravellers` method should throw an error when the API fails. The HTTP failure case can be emulated using `HttpTestingController`. For this, we need to flush the request with an error message and a failure HTTP status instead of returning the data. The following snippet tests the failure case of `getTravellers` method:

```
it("should throw error", () => {
```

```
  let error: string;
  dataAccessService.getTravellers().subscribe(null, e => {
    error = e;
  });

  let req = httpTestingController.expectOne("api/travellers");
  req.flush("Something went wrong", {
    status: 404,
    statusText: "Network error"
  });

  expect(error.indexOf("Error retrieving travellers data") >= 0).toBeTruthy();
});
```

Notice the difference in handling the observable returned from the `getTravellers` method. We passed `null` for a success callback, as we know that this observable will never succeed. The error callback assigns the error to a variable so that it can be asserted.

The other methods of `DataAccessService` can be tested in the same way. Let's test the `createTraveller` method to see how to test a POST call. This method should pass the object it receives to the REST API. The following snippet shows the unit test for this:

```
it("should call POST API to create a new traveller", () => {
  dataAccessService.createTraveller(traveller).subscribe();

  let req = httpTestingController.expectOne({ method: "POST", url: baseUrl });
  expect(req.request.body).toEqual(traveller);
});
```

The `updateTraveller` method invokes the PATCH API to update a traveler. We can test this method to check if the right parameter and body are sent to the API. The following snippet shows this test:

```
it("should call patch API to update a traveller", () => {
  dataAccessService.updateTraveller(traveller, traveller.id).subscribe();

  let req = httpTestingController.expectOne({
    method: "PATCH",
    url: `${baseUrl}/${traveller.id}`
  });
  expect(req.request.body).toEqual(traveller);
});
```

I leave the testing of `deleteTraveller` method as an assignment to the readers. You can also check the sample code to see how it is done.

Testing HTTP Interceptors

HTTP Interceptors are used to handle the tasks that have to be performed with every request going out of the application. Any mistake in the behavior of the interceptor may cause problems in every API request. Some of you may have started thinking how to do this, as interceptors are not directly invoked. They can be tested in the same way as they are used. We can make a request and see if the interceptor is invoked and performs the right action.

Let's consider the following interceptor:

```
@Injectable()
export class LoggingInterceptorService implements HttpInterceptor {
  constructor(private logger: LoggerService) {}

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    req.headers.set('Authorization', 'auth-token');
    this.logger.info(`Calling API: ${req.url}`);
    return next.handle(req).pipe(
      tap(
        (data: HttpEvent<any>) => {
          this.logger.success(`Call to the API ${req.url} succeeded`);
        },
        (error: HttpErrorResponse) =>
        this.logger.error(`Call to the API ${req.url} failed with status ${error.
status}`)
      )
    );
  }
}
```

The above interceptor logs messages when a request is made, a request succeeds or when a request fails. It uses the service `LoggerService` to log these messages. It also sets the Authorization token in the header of every request. For demo purpose, the above service assigns a hard-coded token; but in real life this token has to be read from *localstorage* or any place where the token is persisted before it is assigned to the header.

Setup for testing the `LoggingInterceptor` will involve the following:

- Creating a mock for `LoggingService`
- Configuring the testing module with:
  - `HttpTestingModule` imported to the test module
  - Providing the interceptor and the mock logging service
- Get the references of `HttpClient` and `HttpTestingController` to make requests and to inspect them

The following snippet shows this setup:

```
import { TestBed } from "@angular/core/testing";
import {
  HttpClientTestingModule,
  HttpTestingController
} from "@angular/common/http/testing";
import { HTTP_INTERCEPTORS, HttpClient } from "@angular/common/http";
import { LoggingInterceptorService } from "./logging-interceptor.service";
import { LoggerService } from "./logger.service";

describe("LoggingInterceptorService tests", () => {
  let httpTestingController: HttpTestingController,
    mockLoggerSvc: any,
    httpClient: HttpClient;
```

```
  beforeEach(() => {
    mockLoggerSvc = {
      info: jasmine.createSpy("info"),
      success: jasmine.createSpy("success"),
      error: jasmine.createSpy("error")
    };

    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [
        {
          provide: HTTP_INTERCEPTORS,
          useClass: LoggingInterceptorService,
          multi: true
        },
        {
          provide: LoggerService,
          useValue: mockLoggerSvc
        }
      ]
    });

    httpClient = TestBed.get(HttpClient);
    httpTestingController = TestBed.get(HttpTestingController);
  });
});
```

Every test will make an HTTP request using `httpClient` and then flush the request using `httpTestingController`, so the request is completed and then the test will assert the behavior.

For every request made, the interceptor logs an info message and sets the Authorization header. Let's write our first test to check if this is done correctly. The following snippet shows the test:

```
it("should log a message when an API is called and set the authorization header",
() => {
  httpClient.get("api/travellers").subscribe();

  let req = httpTestingController.expectOne("api/travellers");
  req.flush([]);

  expect(mockLoggerSvc.info).toHaveBeenCalled();
  expect(mockLoggerSvc.info).toHaveBeenCalledWith(
    "Calling API: api/travellers"
  );
  expect(req.request.headers.get("Authorization")).toBeDefined();
});
```

As we see, the interceptor is not directly invoked here, rather its behavior is tested in the same way as it would run in an actual application.

Let's write one more test to check if the interceptor logs the success message when an API succeeds. The following test shows this:

```
it("should log a success message when the API call is successful", () => {
  httpClient.get("api/travellers").subscribe();

  let req = httpTestingController.expectOne("api/travellers");
  req.flush([]);
```

```
  expect(mockLoggerSvc.success).toHaveBeenCalled();
  expect(mockLoggerSvc.success).toHaveBeenCalledWith(
    "Call to the API api/travellers succeeded"
  );
});
```

Testing the API failure case is much similar to the success case. The only difference will be that now we have to fail the request by passing an HTTP error code. The following snippet shows the failure case:

```
it("should log an error message when the API call fails ", () => {
  httpClient.get('api/travellers').subscribe();

  let req = httpTestingController.expectOne("api/travellers");
  req.error(null, { status: 404 });

  expect(mockLoggerSvc.error).toHaveBeenCalled();
  expect(mockLoggerSvc.error).toHaveBeenCalledWith(
    "Call to the API api/travellers failed with status 404"
  );
});
```

Conclusion

Unit testing is a vital part of software development. Because of its detailed and low-level nature, it helps in finding bugs early and fixing them.

Angular Services are created for reusability of data or business logic in an application, so it is important to make sure that the services work correctly. This tutorial explained how services, HTTP requests and the HTTP interceptors can be tested. I hope these techniques help in writing better tests in your Angular applications.

If you have any additional thoughts on Angular unit testing, please leave a comment if you are reading a web version of this article, or reach out to me on twitter.

Download the entire source code from GitHub at
bit.ly/dncm46-angularunittesting

Microsoft® Most Valuable Professional

Ravi Kiran

Author

Ravi Kiran is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (Visual Studio and Dev Tools) and DZone MVB awards for his contribution to the community. Ravi is also a Google Developer Expert.

*Thanks to **Damir Arh** for reviewing this article.*

Want this magazine delivered to your inbox ?

Subscribe here

www.dotnetcurry.com/magazine/

* No spam policy

Subodh Sohoni

Operations is an integral part of DevOps.

One of the main tasks of Operations is to monitor the health of the application and take corrective actions if it is not up to the mark. These activities are usually done by a set of people in the organization called the **Operations group**.

As per the guidelines of DevOps, the responsibility of "operations" should no longer be on an external group (like the Operations group), but should be a part of the responsibilities given to the DevOps team.

AUTOMATED ACTIONS ON AZURE MONITOR ALERTS

To make Operations more reliable, we can automate the monitoring of activities, and appropriate actions. The automation of operations may involve the following:

- monitor health of the application and the environment in which it is hosted.
- If any activity crosses a threshold (set for a condition of performance or quality), raise an alert. These conditions can be availability of application or quality of the application or performance of the hardware that is supporting it.
- Once such an alert is raised, an automated action should take place to bring the "out of bound" parameter back to normal.

Case Study – Monitoring an ASP.NET Web Application using Azure Monitor

We are going to study an application hosted on IIS on a VM in Azure. It is an ASP.NET web application that is already deployed and working. It has certain flaws that we need to find out and take appropriate actions until a bug-fix is found for those. We are going to make use of certain services in Azure to achieve the desired results.

The Azure services that we will use are:

a) Azure Monitor
   - Log Analytics
   - Application Insights
   - Alerts

b) Azure Automation

c) Azure Storage

In this case study, we are going to do the following:

1. **Setup Azure Monitor** – Log Analytics to observe the performance counters of a VM in Azure. This VM is hosting the application that we are monitoring.
2. **Setup Azure Application Insights**, which is a part of Azure Monitor, to observe availability of application.
3. **Setup Azure Automation** and a VM in Azure for running automation scripts.
4. **Create query in Azure Monitor** - Log Analytics to get the records where performance of hardware (e.g. %Processor Time) for the VM hosting our application, is below expected.
5. **Create Azure automation runbooks (scripts)** in Azure Automation which are hooked to alerts in Azure Monitor. These scripts will take the corrective action on the VM when a hooked alert is sent.
6. **Create alerts in Azure Monitor** to notify runbook so that appropriate actions are taken.

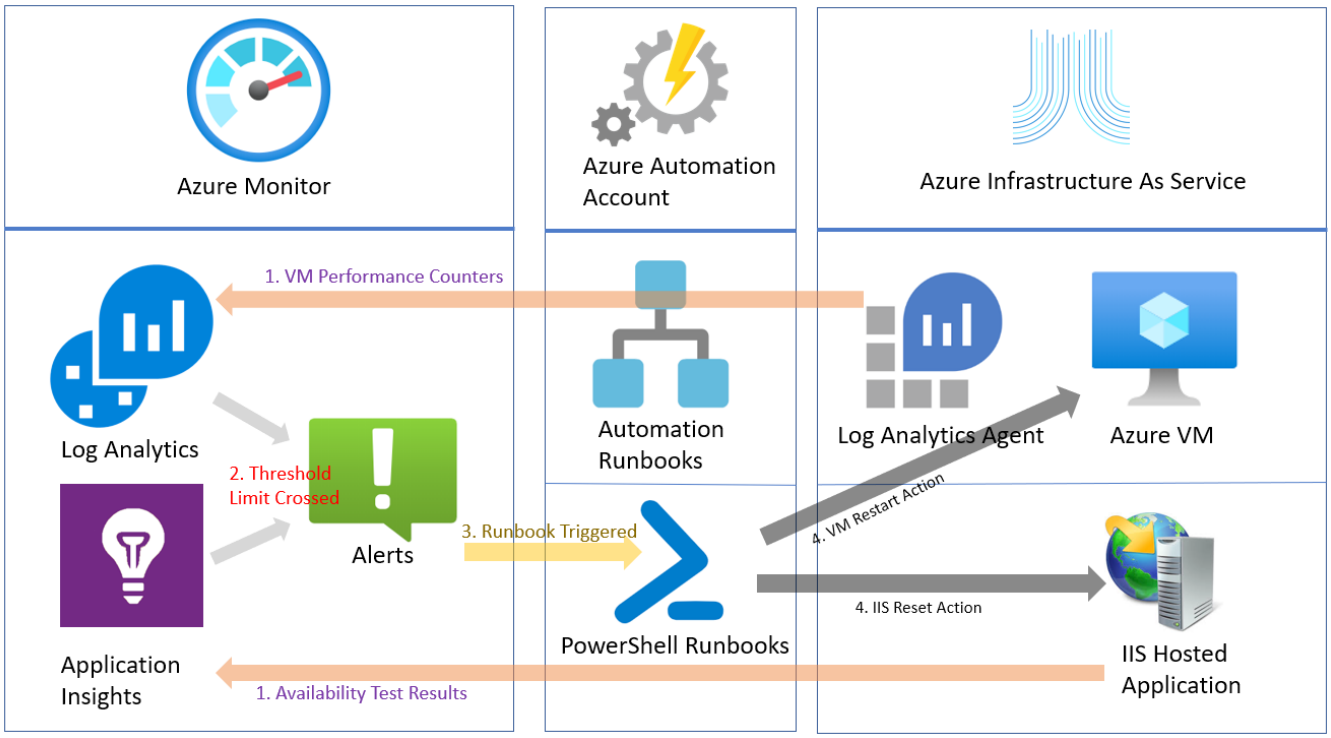


Figure: Case of integration of Azure Monitor with Azure Automation

Azure Monitor

Azure Monitor is a service offered by Azure to monitor and report various parameters related to health of the resources in Azure and on-premises. Set of services offered by Azure Monitor include Application Insights, Log Analytics, Alerts and predefined Metrics in a graphical form.

Azure Monitor can target various resources in Azure like Application Services, Virtual Machines, Storage Accounts, Containers, Networks and Cosmos DB.

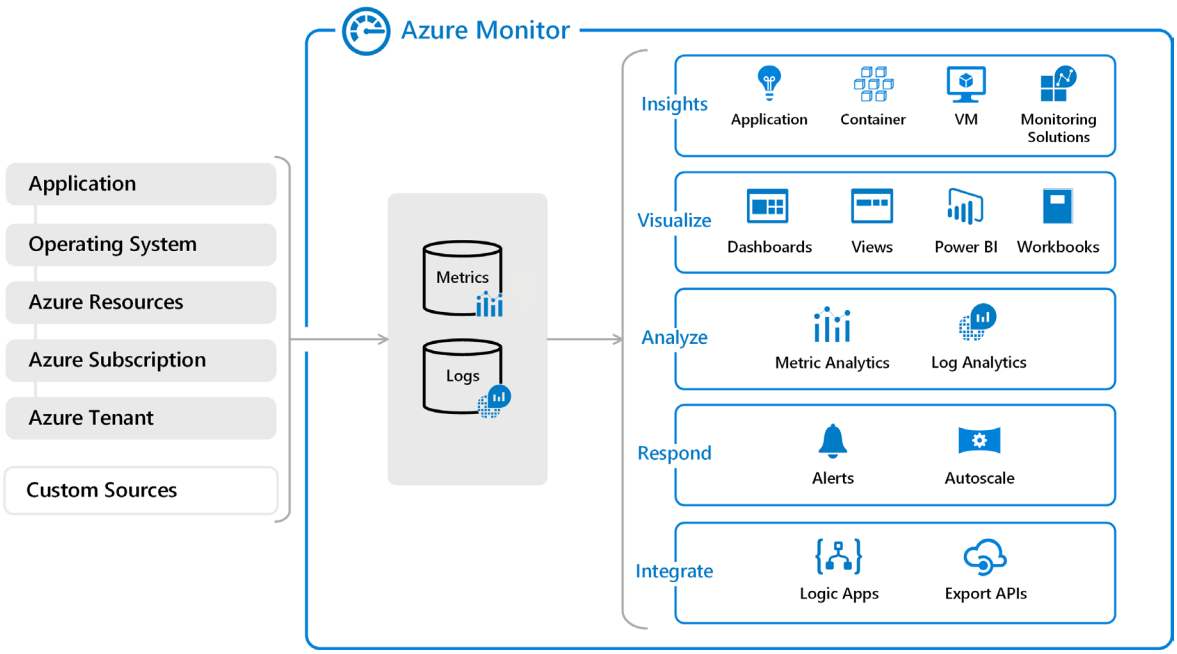
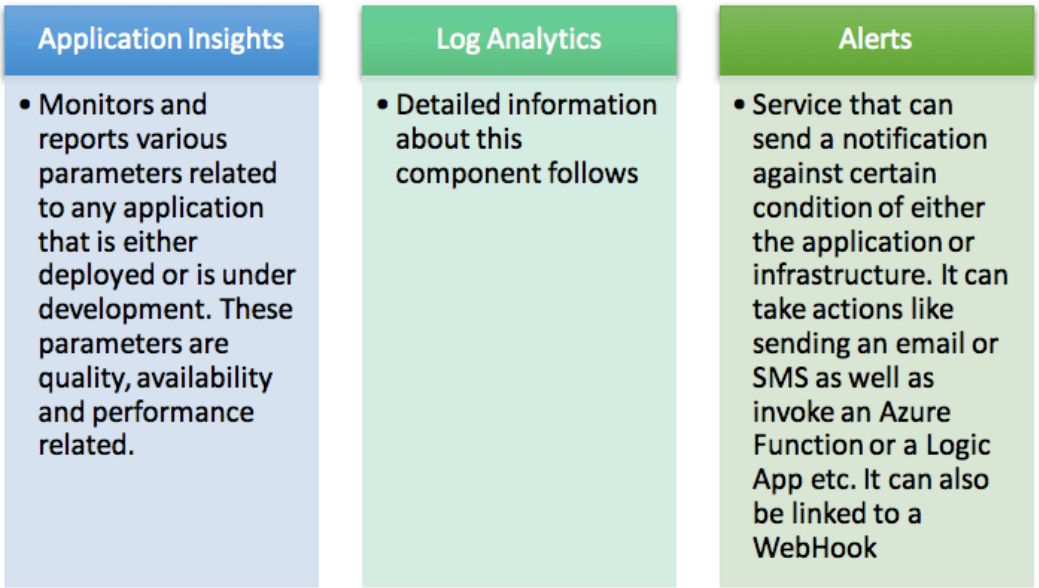


Figure: Azure monitor overview

Major components of Azure Monitor are:



Note: I have already written in details about Application Insights and alerts in my previous article. Let us now study **Log Analytics** in this article.

Log Analytics

Log analytics is a feature of Azure Monitor to query and display filtered results of various logs collected by Azure Monitor service. Application Insights data too can be queried with Log Analytics.

In this example, we will setup and query the performance counters of a virtual machine that is part of IaaS service of Azure. On this VM, I have installed an ASP.NET Web Application. What we want is when the application runs, if the performance counter of "% Processor Time" exceeds 90%, the VM should be restarted.

Setup Azure Monitor Log Analytics

To use the Log Analytics, we need to create a **Log Analytics Workspace**. This is easily done by searching for Log Analytics Workspaces in the search box (top) in the Azure portal and then use *+Add* to create a new Log Analytics Workspace.

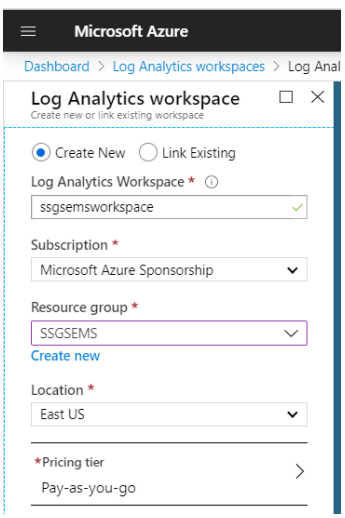


Figure: Create log analytics workspace

Once the Log Analytics Workspace is created, go to the section of "Get started with Log Analytics" and select Azure virtual machines (VMs) in it. This section shows a list of VMs in our account. We can select a VM from the list and connect to it.

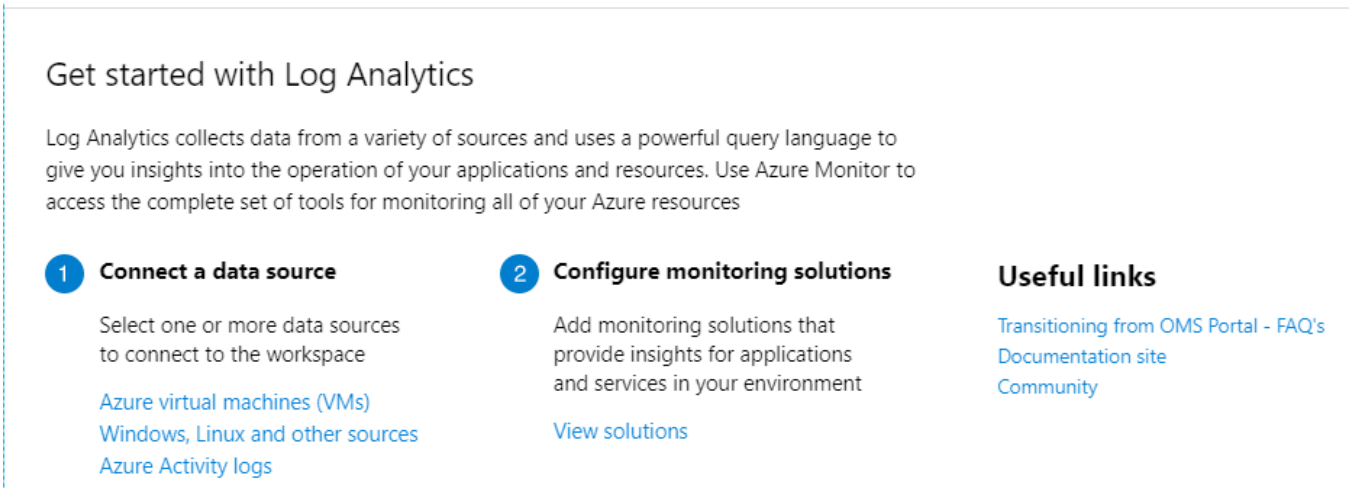


Figure: Get started with log analytics

Select a VM from the list and connect to it.

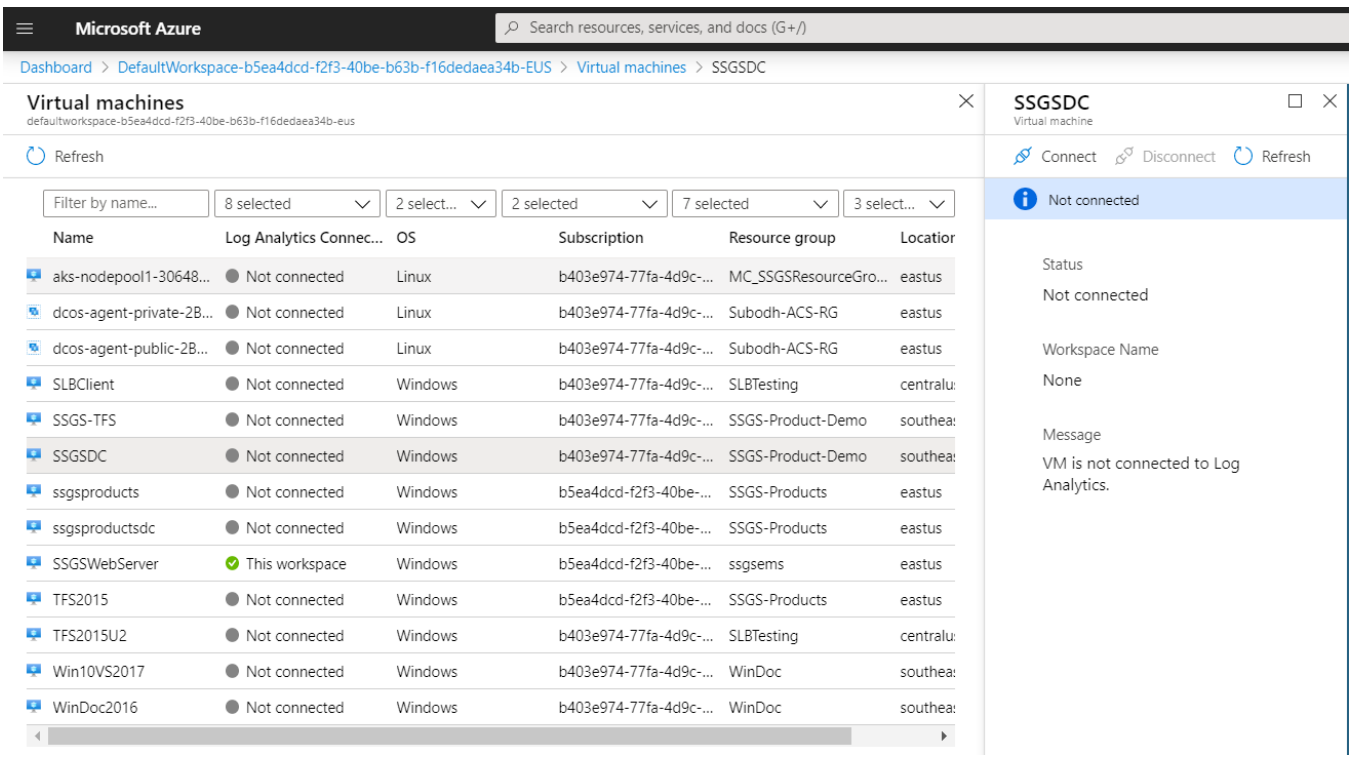


Figure: Connect log analytics to vm

When we click the connect button for a VM, a Log Analytics Agent is installed on the machine. Log analytics agent is a component of Log Analytics which resides on infrastructure hosting the application, like a VM. The agent starts collecting the logs from that machine and starts sending those to Log Analytics of Azure Monitor.

Log Analytics Agent does not automatically collect the performance counters of the VM. We have to configure it to do so. Let's see how we can do that.

1. To start collection of performance counters of VM, let's open the "Advanced Settings" of the Log Analytics Workspace that we created earlier.

2. On the page that opens, create a Data section. We will select the Windows Performance Counters from the list.

3. From the next list of various performance counters, we can retain the checks in the checkboxes of those performance counters that we want to collect.

4. Finally, by clicking the button of "Add the selected performance counters" we enable the agent to collect those performance counters.

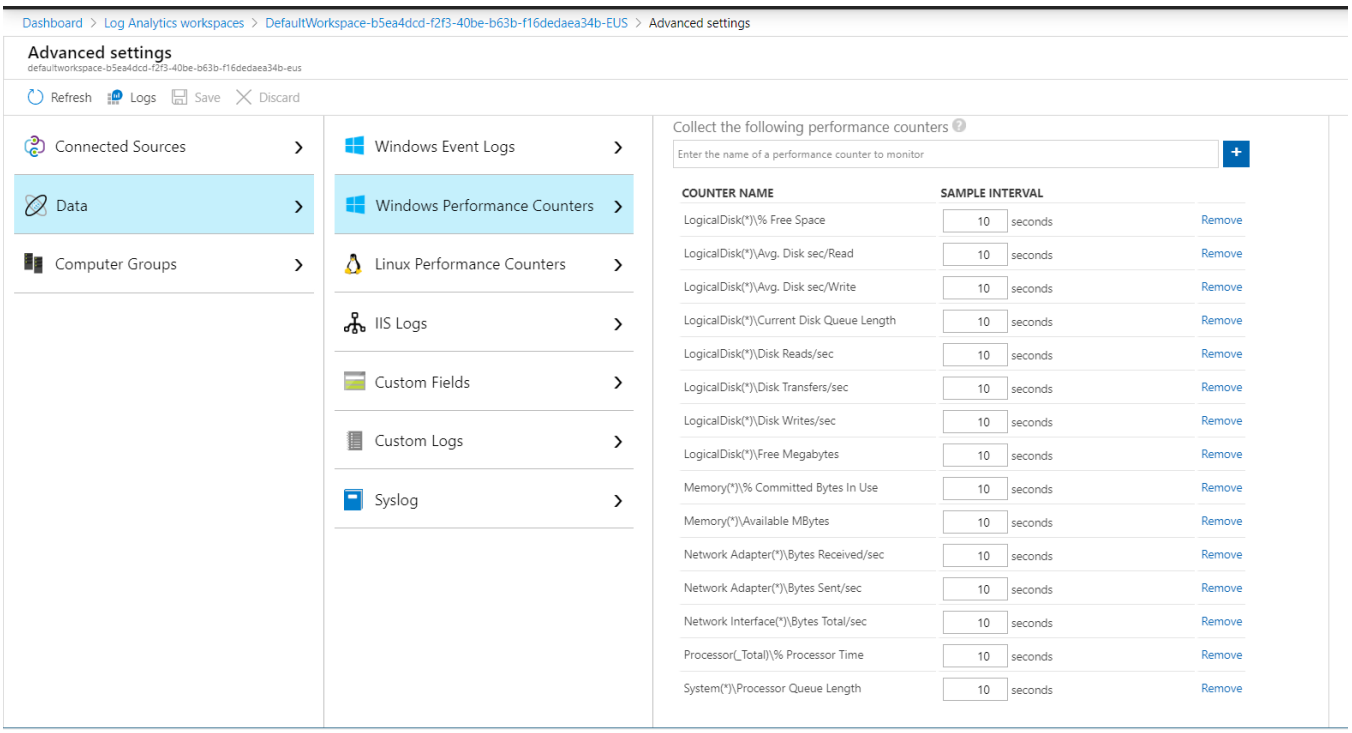


Figure: Add performance counters to log analytics workspace

Now that we have configured Log Analytics and the VM to collect the desired logs of performance counters of that VM, we can create a query that will filter and return the results. We want that only those log records should be considered where the "% Processor Time" is beyond 90%. This is the condition where the performance of the application is below expected and if that is true, we want to restart the VM.

Setup Azure Monitor – Application Insights

Application Insights is part of Azure Monitor that monitors the application that is deployed. One of the parameters that it can monitor is the **Quality of Application**.

We can measure quality of application in different ways. We can monitor exceptions thrown, unhandled exceptions, logical errors, usability and many more such pointers to quality of application.

Azure Automation Setup to Access Azure Resources

Setup Azure Automation to create PowerShell Runbooks under it. The first step in it is to create an Azure Automation Account from the Azure Portal. This is a simple step and does not need any parameters to be selected.

Once our Azure Automation account is created, configure the modules under it to run scripts that will allow us to authenticate to Azure Account, and to access resources under our Azure Subscription. For this, the following modules need to be present in our Azure Automation account:

1. Az.Accounts
2. Az.Computing
3. Az.Resources
4. Az.Storage

To check that these modules are present, click the Modules under *Shared Resources* section of Azure Account in the Azure portal. You will see that unfortunately and surprisingly, none of the Az modules are added by default to the Azure Automation Account (this is confirmed to be true until mid-March 2020 when I wrote this article). To install these modules, click the Browse Gallery link at the top and then search for the required module, e.g. Az.Accounts.

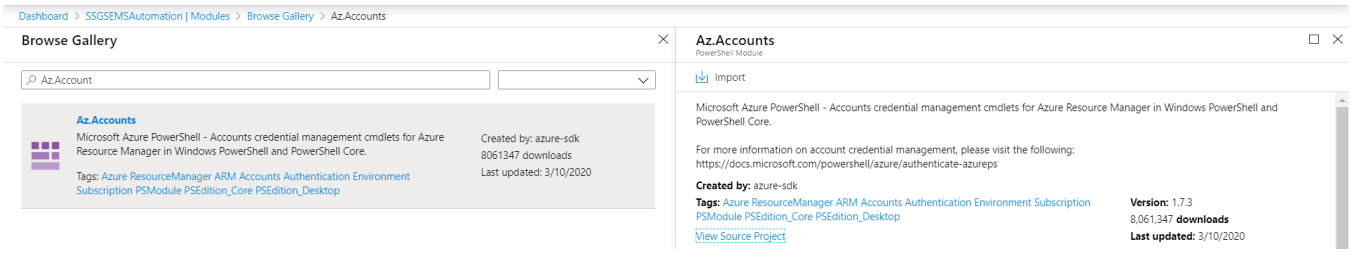


Figure: Add selected Az module

Ensure that you are importing the stable version of the module and not a Preview version. Preview versions of these modules do not have features that we require. Click the Import button, agree to update dependent modules and then confirm by clicking the OK button. Import all the modules that are required (mentioned above).

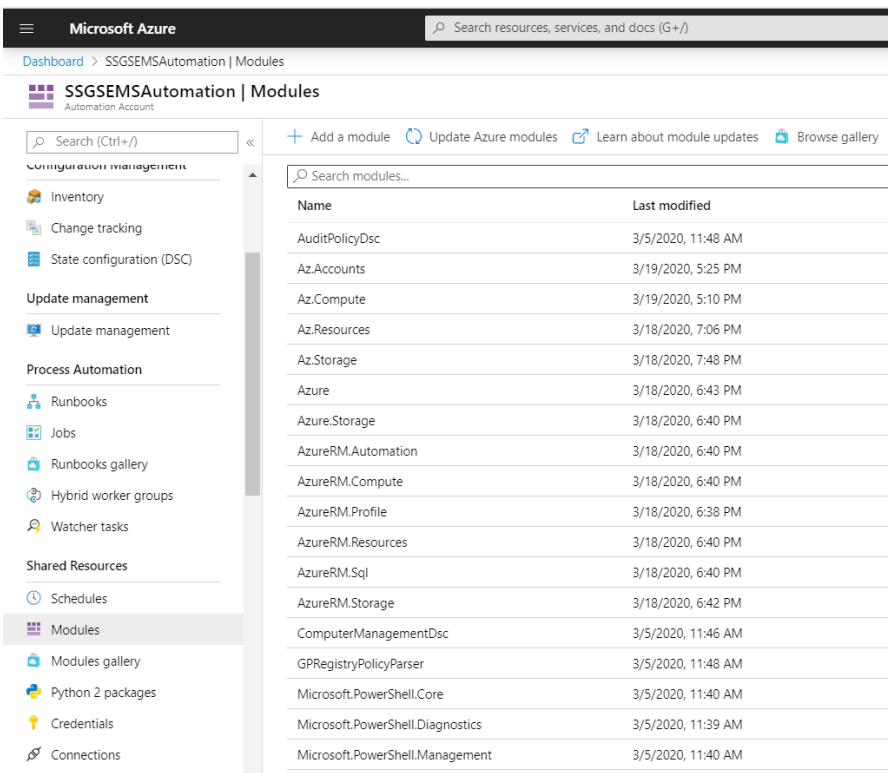


Figure: Modules installed in azure automation

Another configuration change to do is to set the access policy. We need to access various Azure resources like storage, VMs etc. in the PowerShell runbooks that we will be creating. To give access of those resources to the identity of the PowerShell runbook, create a Service Principal. In Azure Automation, it is called "Run as Account". Click the "Run as Account" under the Account Settings section and then click "+ Azure Run As Account".

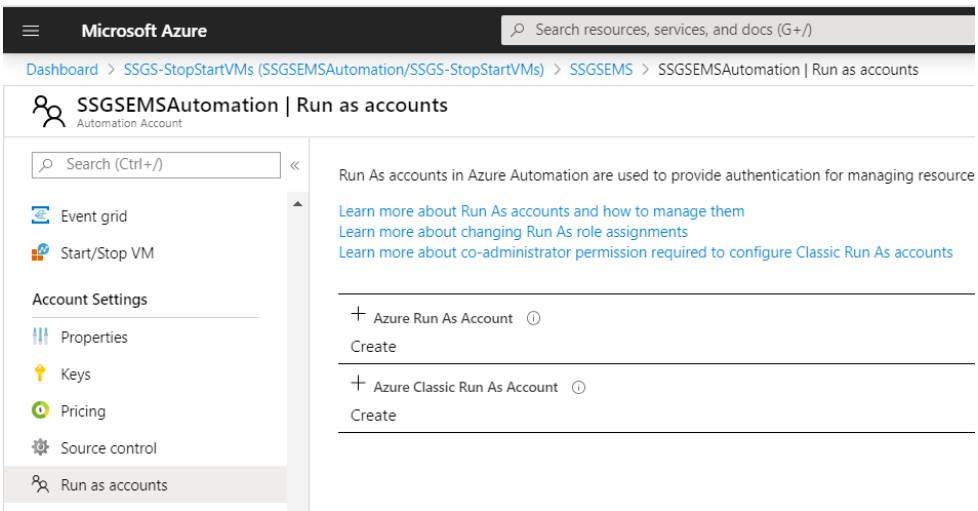


Figure: Create Run as Account

This creates the Service Principal and sets it in the "Contributor" role of the RBAC (Role Based Access Control) of the subscription. By doing so, we are allowing access to all Azure resources in our subscription.

We now have all the necessary components setup and configured. These components are:

1. Azure Monitor – Log Analytics
2. Azure Monitor – Application Insights
3. Azure Automation Account

Let's now create the tests in Application Insights and a query in Log Analytics that will be used for monitoring the application and the hardware hosting that application.

Create the Monitoring Conditions

Create the Availability Test

In this case study, I have taken the example of availability of all the pages of application as the indicator of quality of the application. Availability of pages of application are possible to be monitored using Availability Tests that are created under Application Insights.

I have already written a guide to create such Availability Tests in my earlier tutorial on Application Insights. Please refer to it and create at the least one such availability test for the application that we have hosted on our VM.

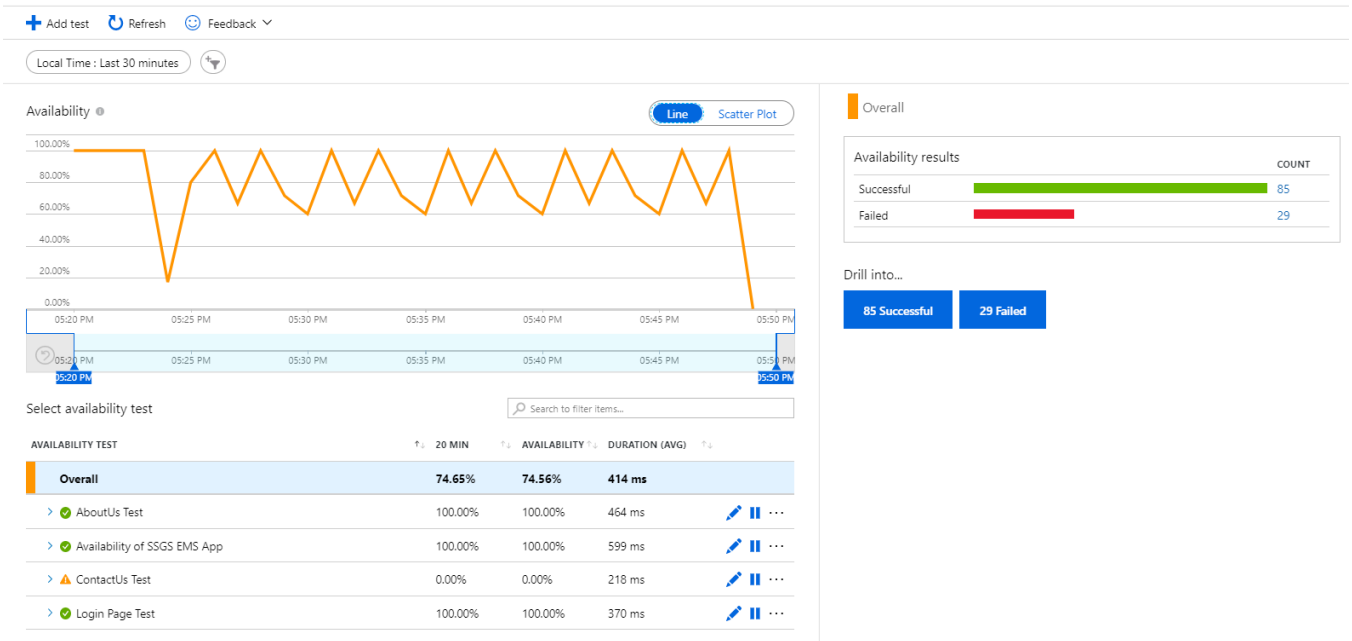


Figure: Application Insights Availability Test

If the availability test failure goes above 10%, we want to take an action. This action is to reset IIS so that if the application change has not got initiated, then it will be. Resetting IIS is an action that is subtle compared to restarting the whole VM. It is also more complex because we need to go inside the VM, find the IIS process and restart it using the tool iisreset.exe. We will need to write an Azure Automation PowerShell Runbook to do that.

Create the Log Analytics Query

The queries of Log Analytics are written in a language that is known as **Log Analytics Query Language**. In its earlier form, it was known as Kusto.

Log Analytics Query Language targets the tables of data that store the collected logs. Some of these tables are event, operation, perf (Performance Counters), syslog etc.

Note: *Learn more about Log Analytics Query Language.*

Providing just the table name, say Perf, will return all the records (or top 10000, whichever is less) of that table.

We can filter the logs using "where" clauses in the query. To set a filter on the results, we will use the | (pipe) operator and give a *where* clause. If we want that only records of "% Processor Time" performance counter should be returned, then we will add the where clause as –

```
| where CounterName == "% Processor Time"
```

We can further narrow the results for getting records where value of the "% Processor Time" is more than 90%. To do so, add another *where* clause as –

```
| where CounterValue > 90
```

Now the entire query will look like this –

```
Perf
| where CounterName == "% Processor Time"
| where CounterValue > 90
```

We can also set the time limit for the records to be considered.

When we run this query, it will return only those records that fulfill the filter criteria. *I have modified the query for a threshold of 30% to get some results quickly*. The query and the results are shown here:

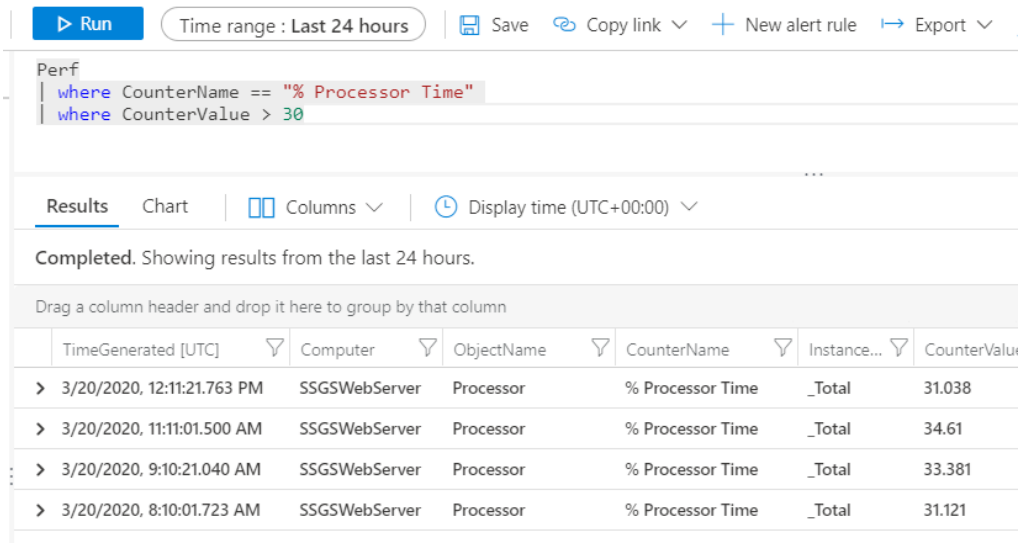


Figure: Log Analytics query with results

We can also sort the results and take only a few records from it by using "sort" and "top" clauses. Aggregation of the results is done with the "Summarize" clause and grouping is done with the "by" clause.

So far, we have used Log Analytics to create a query that filters the logs to give us only the records where "% Processor Time" is beyond 90%. Let us save that query with a name "Pct Proc Load GT 90".

We can now create an Alert based upon that query. Alert under Azure Monitor needs a condition of a signal logic to be true, and the action to be taken, if it becomes true. The action we need to take is to restart the VM. This action can be configured in Azure Automation as a PowerShell Runbook.

Create Actions against Notifications

Create Azure Automation Runbooks to Take Action When Notified

Once the Azure Automation Account is configured in this way, we can start creating the runbooks. The first runbook that we will create is to restart the VM.

On the Azure Automation account, click the Runbooks under the Process Automation section. It shows a list of three tutorial runbooks. Click on the "Create Runbook" button. Give the name RestartVM to this runbook and select PowerShell as the type of runbook.

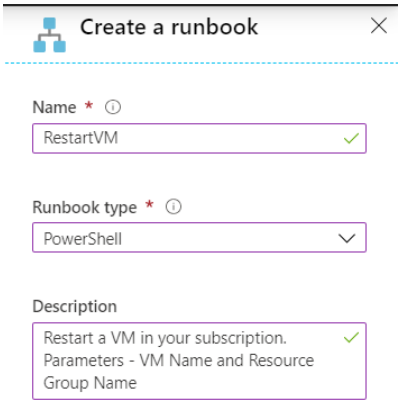


Figure: Create a PowerShell Runbook

In the runbook that is created, add the following code:

```
param (
  [Parameter(Mandatory=$true)]
  [String] $VMName,
  [Parameter(Mandatory=$true)]
  [String] $RGName,
)
## Authentication
try
{
  # Ensures you do not inherit an AzContext in your runbook
  $null = Disable-AzContextAutosave -Scope Process
  $connectionName = "AzureRunAsConnection"
  $Conn = Get-AutomationConnection -Name $connectionName
  Connect-AzAccount `
-ServicePrincipal `
-Tenant $Conn.TenantID `
-ApplicationId $Conn.ApplicationID `
-CertificateThumbprint $Conn.CertificateThumbprint
    Write-Output "Successfully logged into Azure."
}
catch
{
  if (!$Conn)
  {
    $ErrorMessage = "Service principal not found."
    throw $ErrorMessage
  }
  else
  {
    Write-Error -Message $_.Exception
    throw $_.Exception
  }
}

## End of authentication
Write-Output "Trying to restart virtual machines ..."
try
{
  Restart-AzVM -ResourceGroupName $RGName -Name $VMName
}
catch
{
  Write-Error -Message $_.Exception
```

```
  throw $_.Exception
}
```

This script first authenticates to Azure and then runs the Restart-AzVM cmdlet to restart the VM that is identified by the Resource Group Name and the VM Name. We will save this runbook and publish it so that it is available from outside, before creating the second runbook.

The next runbook that we are going to create is for execution of iisreset on the VM. We need to run a PowerShell script on the VM to do that.

Initially we will keep that PowerShell script in an Azure Storage as a blob. This PowerShell script will only have the command of "iisreset", and nothing else.

Create the PowerShell script in your favorite editor and save it locally as a file "ResetIIS.ps1". Then in Azure Portal, create an Azure Storage account and a blob container in it named "psscripts". Set the Container level access permission. Upload the ResetIIS.ps1 file in this container.

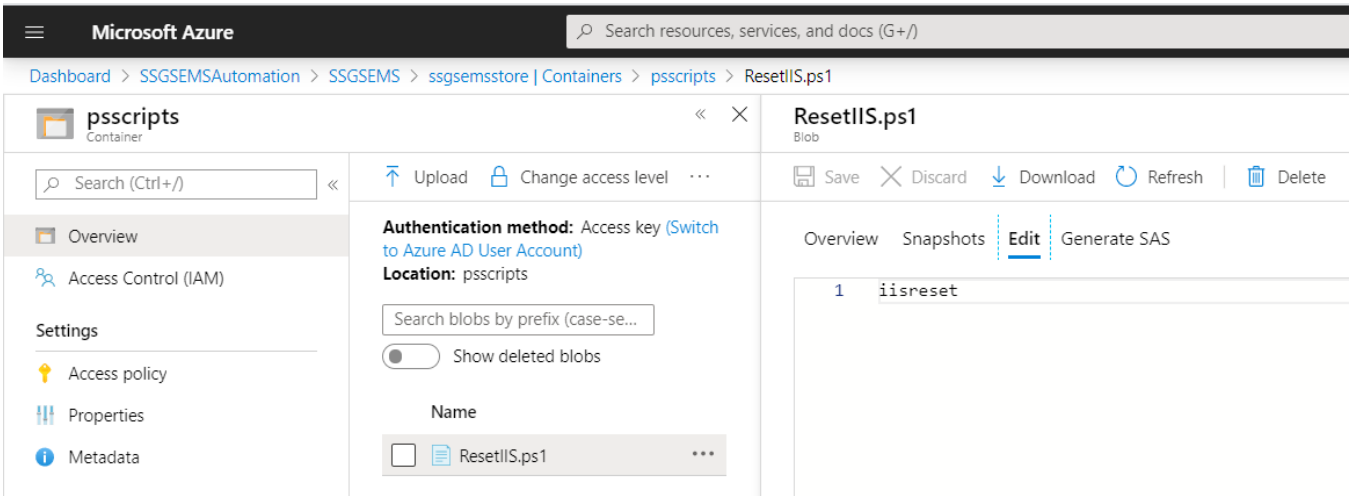


Figure: Reset IIS PowerShell script

Once it is saved, get the URL of that file and store it somewhere to use it later in another script.

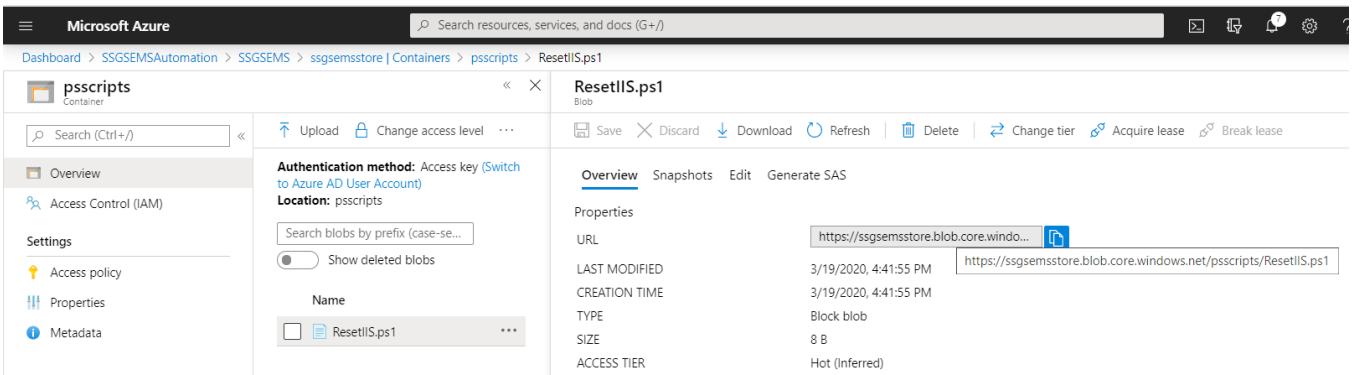


Figure: URL of PowerShell script blob

Now we will create a second runbook under the Azure Automation Account created earlier. This runbook will also be a PowerShell type runbook. Let's call it as "ResetIISOnVM".

The code in this runbook is very similar to the code in our previous one. The only change in it is that instead of *Restart* command, there will be two different commands. First one is to download the ResetIIS.ps1 PowerShell script from the Azure Storage using *wget* tool. The second one is the Invoke- AzVMRunCommand

cmdlet to call ResetIIS.ps1.

```
wget "https://ssgsemsstore.blob.core.windows.net/psscripts/ResetIIS.ps1" -outfile
((Get-Location).path + "\ResetIIS.ps1") -UseBasicParsing
    Invoke-AzVMRunCommand -ResourceGroupName $RGName -VMName $VMName -ScriptPath
((Get-Location).path + "\ResetIIS.ps1") -CommandId 'RunPowerShellScript'
```

Parameters of Resource Group Name and VM Name will be provided at the time of creating the alert that calls this runbook. We will now save this runbook.

Now the Log Analytics Queries representing the condition for sending alerts are ready. The Azure Automation Scripts that represent the action are ready too. We only need to connect each script to the appropriate query. This is done with the help of **Azure Monitor Alerts**.

Creating Azure Monitor Alerts

Alerts is a mechanism in Azure Monitor by which notification of some condition is passed to the receptors. This notification can be as simple as sending an email or a SMS, or it can be complex as a call to an Azure Function or a Logic App. We are going to create the alerts that invoke Azure Automation Runbooks.

Create an Alert for Hardware Overload

To create an alert, we need to specify the condition and the action. We will start the wizard to create a new Alert Rule. Under the Azure Monitor, select the Alerts and click the "+ New alert rule" button to start the alert rule creation wizard. Let us first select the Azure Subscription and Log Analytics Workspace to monitor as the target of monitoring.

To define the logic to raise an alert, click the *Add* button to select the condition. From the custom saved queries, select the query that we had created to check if "% Processor Time" goes beyond 30. The value 30 is for demonstration purpose only, in the real-life conditions you may want it to be somewhere beyond 80.

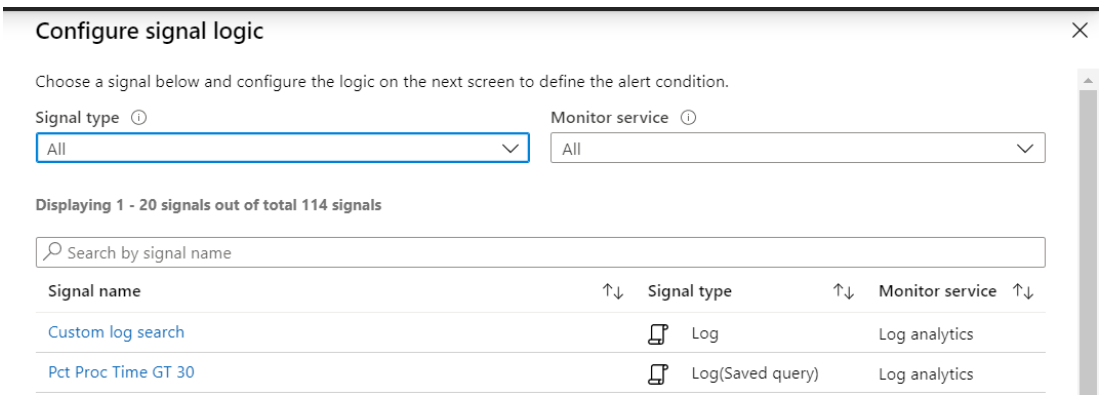


Figure: Select query as alert signal logic

The search query that we had created is already there. In addition to the query, we can also configure that alert should be raised if the alert logic threshold exceeds more than three times (3 is for example, we can change that value if desired). We can also change the frequency of evaluation if needed.

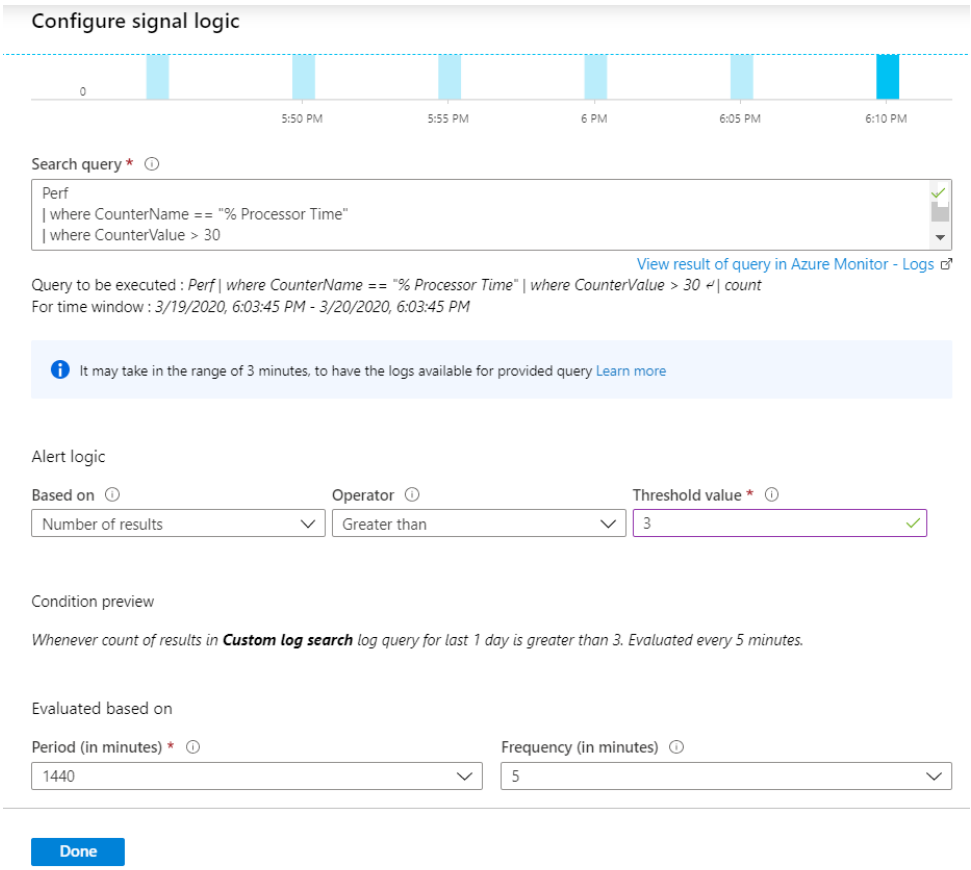


Figure: Alert signal logic

The next step is to set the action to be taken when this condition is met. In the section of Action Group, we will create a new action group and a new action under that. In this action, let's select action of the type Automation Runbook. In the details selection, we will select "User" defined runbook and then drill down to the runbook that we had created to restart the VM. Provide necessary parameters like VM Name and Resource Group Name in which the VM exists.

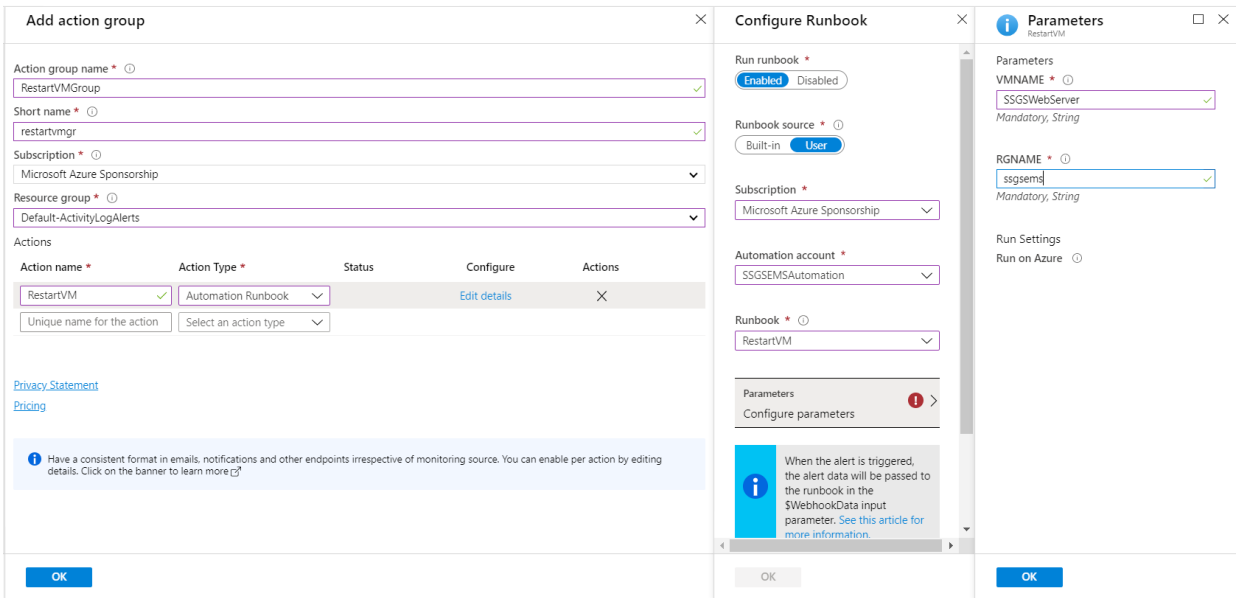


Figure: Add action to restart VM

Give a name to the Alert Rule and save it by clicking the button of "Create Alert Rule".

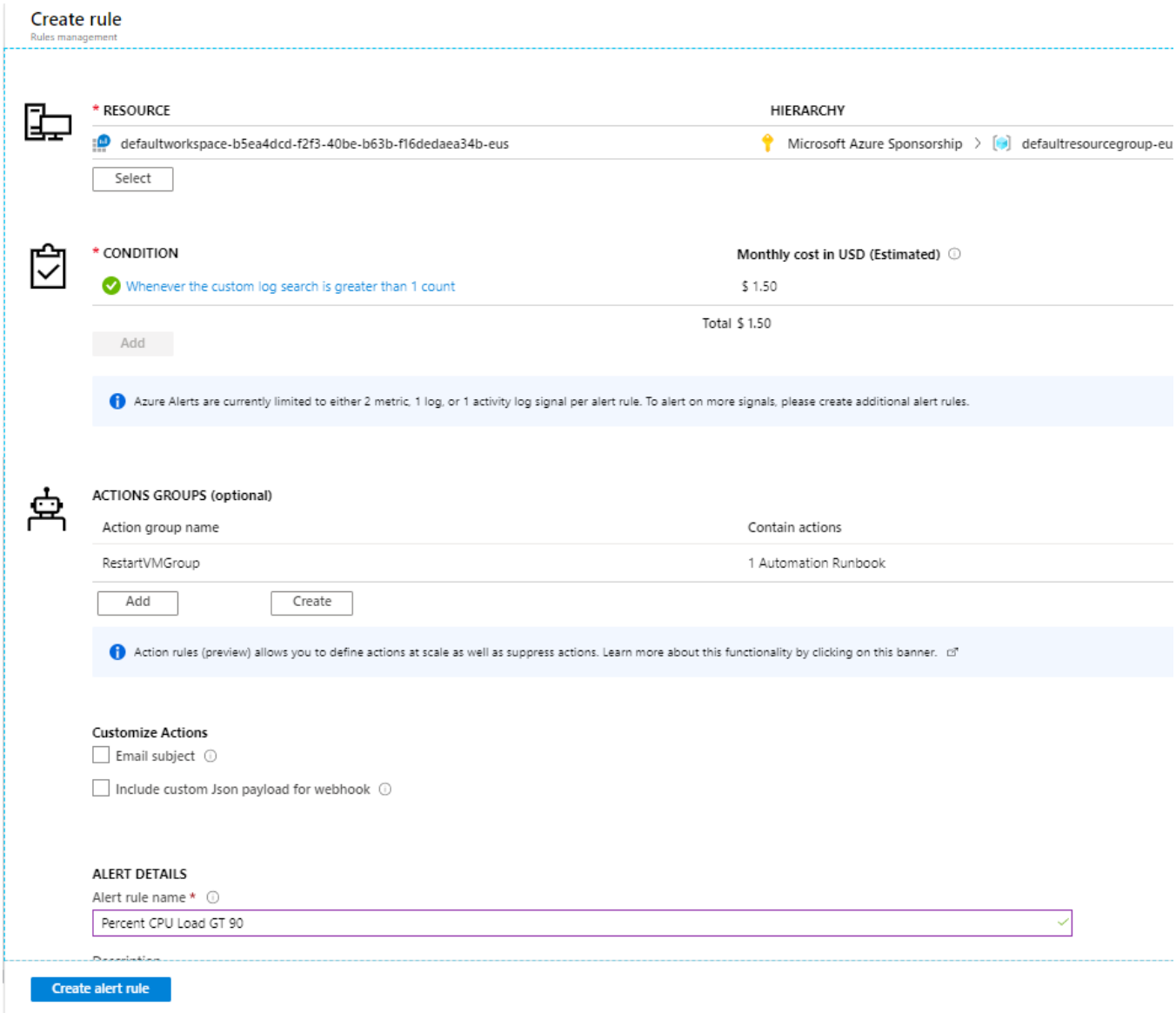


Figure: Add rule restart VM

This way, we have linked the Log Analytics – Azure Monitor Alert – Azure Automation Runbook, so that if the load on the processor exceeds beyond the set limit consistently, the VM will be restarted without any manual intervention.

Create an Alert for Test Failures

We will now create the combination of Application Insights – Azure Monitor Alert – Azure Automation Runbook, so that IIS on the VM will be reset if the availability of the application goes below set limit.

We will start the wizard to create a new alert. Let's select the Application Insights resource of the application as the resource to be monitored. In the Signal Logic (condition), select the availability test that was created. Set a logic that the Test passed at a value smaller than 95%.

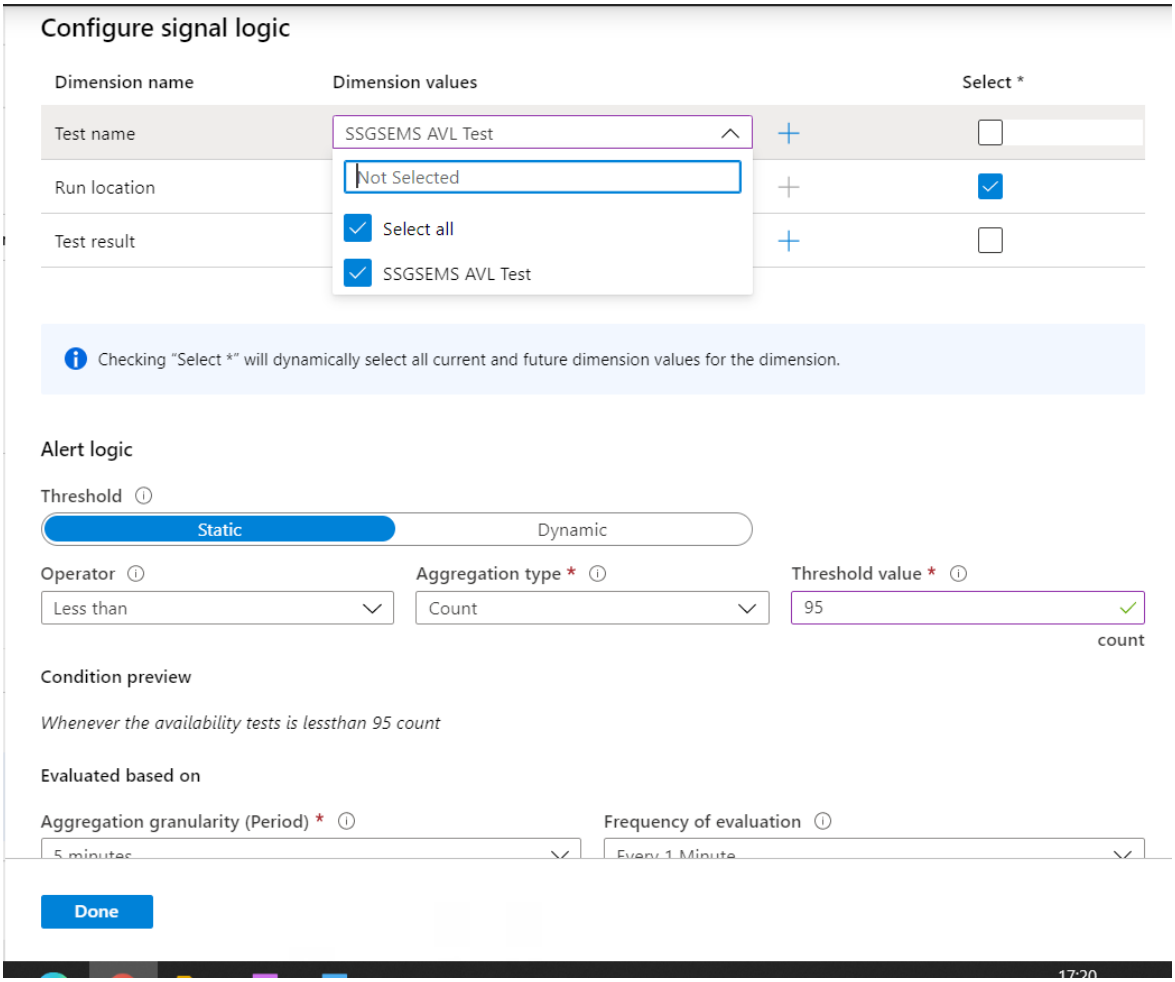


Figure: Signal logic for availability tests condition

In the action, select the Azure Automation Runbook to reset the IIS on VM.

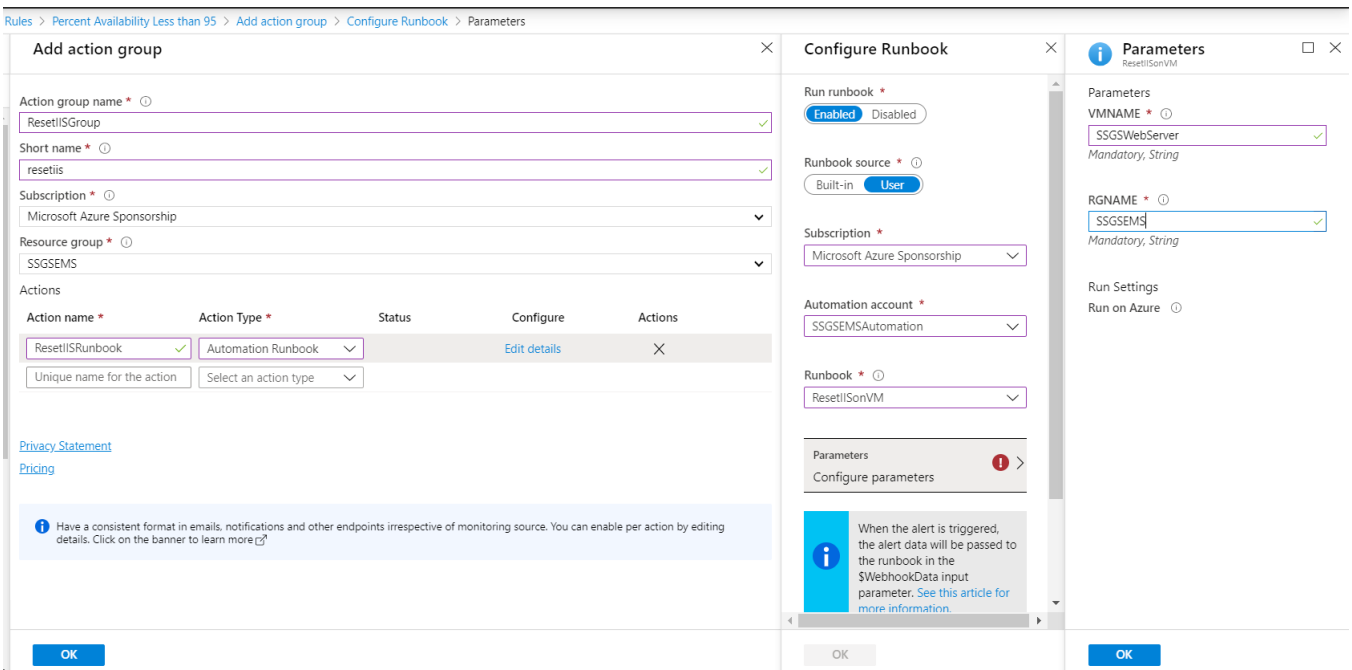


Figure: Add action to reset IIS

Save the alert by clicking the "Create new alert" button. This way, we have linked the Application Insights –

Azure Monitor Alert – Azure Automation so that if availability test fails more than a set limit, the IIS on the VM will be reset without any manual intervention.

Summary

In this tutorial, we have seen **how to automate some of the important parts of Operations, an integral part of DevOps.**

We have taken a case of an ASP.NET application hosted on IIS that is on a VM in Azure. We used services offered by Azure Monitor – Log Analytics, Application Insights and Alerts to monitor the application and the hardware hosting it. We used Azure Automation Runbooks to take the desired actions – restart the VM if the hardware is overloaded or to reset IIS, if the application has some problems.

These examples can be extended to monitor other parameters and automate many of the actions that are desired to be taken. Lesser dependency on manual intervention makes these operations more reliable.

• • • • • • •

Microsoft® MVP Most Valuable Professional

Subodh Sohoni

Author

Subodh is a consultant and corporate trainer. He has overall 28+ years of experience. His specialization is Application Lifecycle Management and Team Foundation Server. He is Microsoft MVP – VS ALM, MCSD – ALM and MCT. He has conducted more than 300 corporate trainings and consulting assignments. He is also a Professional SCRUM Master. He guides teams to become Agile and implement SCRUM. Subodh is authorized by Microsoft to do ALM Assessments on behalf of Microsoft.

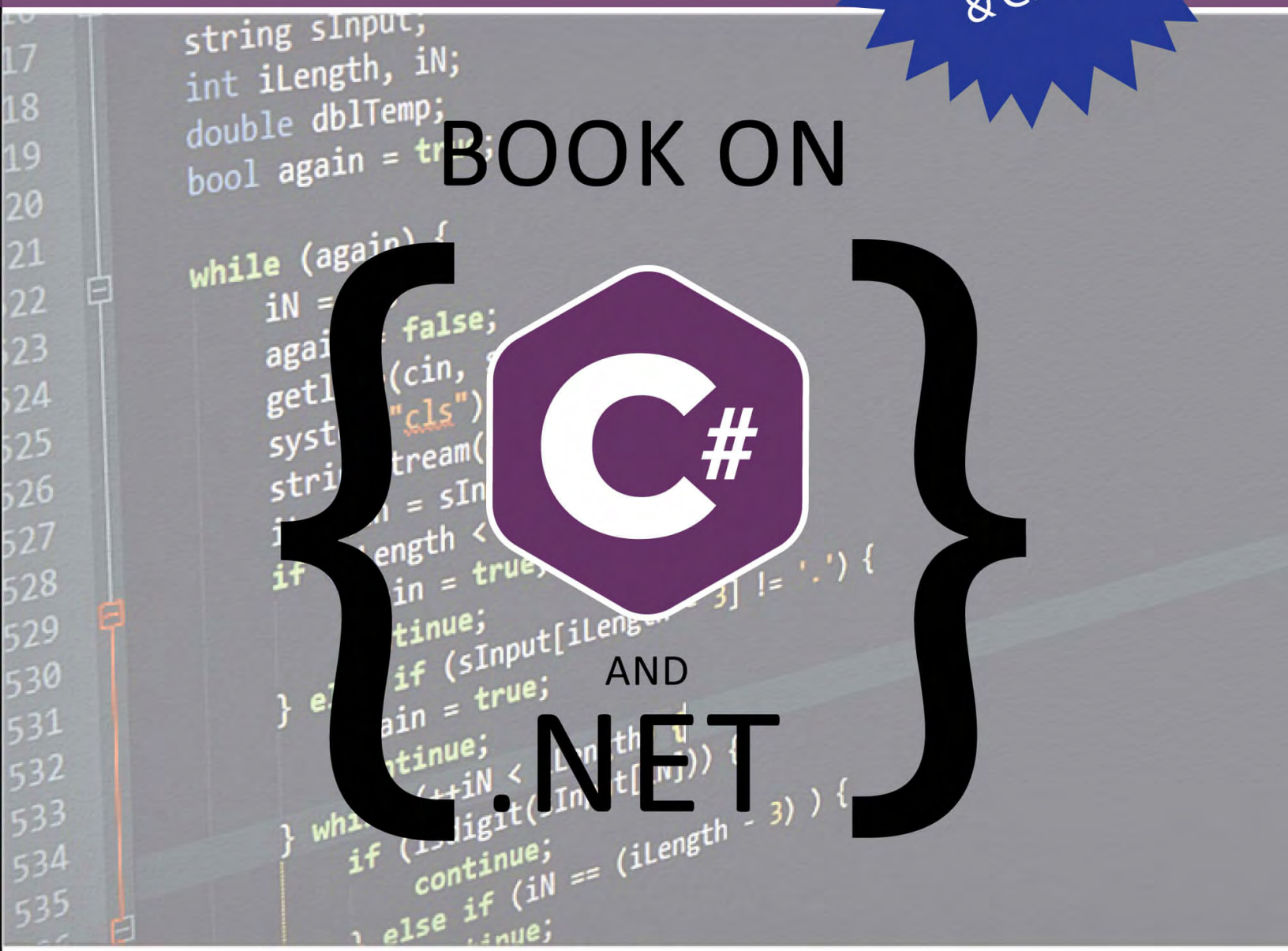
Follow him on twitter @subodhsohoni

*Thanks to **Gouri Sohoni** for reviewing this article.*

COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY AWESOME BOOK ON { C# AND .NET }

Includes .NET Core 3.0 & C# 8.0



DAMIR ARH

ORDER NOW

Damir Arh

DEVELOPING CLOUD APPLICATIONS IN .NET

This tutorial provides an overview of different cloud services available to .NET developers for publishing their applications on Microsoft Azure.

Cloud applications have many commonalities with ordinary web applications. The main difference is in the way they are hosted. Instead of being deployed to our on-premise servers, they take advantage of services offered by cloud providers.

In this tutorial, I'm going to provide an overview of the cloud-based hosting options available for .NET applications, and how these options affect the application code. To learn more about web development in .NET, you can read my previous article from this series: Developing Web Applications in .NET.

Which hosting option should I choose?

This decision strongly depends on how adaptive the application is to the cloud environment:

- The least cloud specific hosting option is **Infrastructure-as-a-Service (IaaS)**. This term is used for virtual machines hosted in the cloud which are completely under our own control as if they were located on our premises.

- The next step is **Platform-as-a-Service (PaaS)**. In this model, the cloud provider keeps control of the operating system and only allows configuration of the web server software. For the application to work in such an environment, it must not depend on the local file system or any other software running or being installed locally.

- To avoid this restriction, applications can be deployed as Docker containers which declaratively specify all the local dependencies and configuration they require. At deployment time, the hosting environment will compose an isolated container instance based on these specifications. Such a hosting model is probably the most representative part of the so-called **cloud-native applications**.

- The latest hosting model introduced by cloud providers is **serverless computing**. Instead of hosting the web application as a single unit, this model requires it to be decomposed into separate independent functions. These are deployed, started and invoked as separate units. That's the reason why this model is also named **Function-as-a-Service (FaaS)**.

When deciding whether to move from on-premise hosting to the cloud, it's important to keep in mind all the advantages of cloud services, especially if they seem to be a more expensive alternative at the first glance:

- Cloud allows us to **dynamically scale resources** to match current requirements. There's no need to own or pay for more resources all the time if they are only needed for a specific period (e.g. during traffic spikes or additional periodic processing of data).

- Provisioning new resources in the cloud for new projects is **much cheaper and faster** than buying new hardware.

- The cloud providers employ dedicated staff **focusing on all aspects of security**: updating the software with latest securing patches, managing network configuration and ensuring physical security of servers. It's very difficult to match the level of security they can provide.

You can read more about the things you need to consider before moving to the cloud in Vikram Pendse's article Microsoft Azure Cloud Roadmap.

In the remainder of the article, I'm going to explain each of the hosting options in more detail from the perspective of a .NET developer. I'm going to be using the services available in Microsoft Azure. However, most other cloud providers also have their own offerings for each of the listed hosting options.

Infrastructure as a Service (IaaS)

The Infrastructure-as-a-Service approach to hosting in the cloud is based on regular virtual machines with unrestricted administrator access including remote desktop for Windows-based machines and SSH for Linux-based machines.

This full flexibility in the way the machine can be configured, comes at a cost. Not only are we responsible for applying security fixes and OS updates, but we must also install and configure all the prerequisites for

our application to work. Let's see what this encompasses when deploying an ASP.NET or ASP.NET Core web application to such a virtual machine.

The Infrastructure-as-a-Service offering in Microsoft Azure is called Azure Virtual Machines. Through a simple user interface, it allows us to quickly provision a new virtual machine by providing only the most basic information such as the hardware specification, the operating system, the network configuration and the login information. If necessary, we can configure many additional aspects, both before and after the creation of the virtual machine.

Visual Studio 2019 has built-in support for publishing ASP.NET and ASP.NET Core based web applications to Azure Virtual Machines running Windows. The feature uses the Web Deploy tool to deploy the application to IIS (Internet Information Services) web server on the selected virtual machine. It's the same tool that we can use to deploy to our on-premise servers.

For this to work, the virtual machine must have the Web Deploy tool installed, as well as the .NET framework and IIS itself with ASP.NET support configured. All of this can be done manually via a remote desktop connection to the virtual machine. There are detailed instructions available along with a PowerShell script for automating the process. The documentation also includes a link to a template file which can be used to create a new virtual machine with all the required software already preinstalled.

To host an ASP.NET Core web application in IIS, the .NET Core Hosting Bundle must be installed in addition to all of the prerequisites above. To deploy the application in framework-dependent mode, a compatible version of .NET Core must be installed on the server as well. Otherwise, the application will only work when deployed in self-contained mode. This isn't specifically listed in the linked documentation, nor does there seem to be a template available to take care of that during the virtual machine creation. It's a .NET Core requirement for being hosted in IIS, which is in no way specific to Azure Virtual Machines.

Once the virtual machine is correctly set up, the publishing process from Visual Studio 2019 is straightforward. From the *Publish* window for an ASP.NET or ASP.NET Core web application, a wizard can be launched which can connect to an Azure account and list the available virtual machines when publishing to an Azure Virtual Machine is selected. Additional settings (e.g. build configuration) can be configured afterwards.

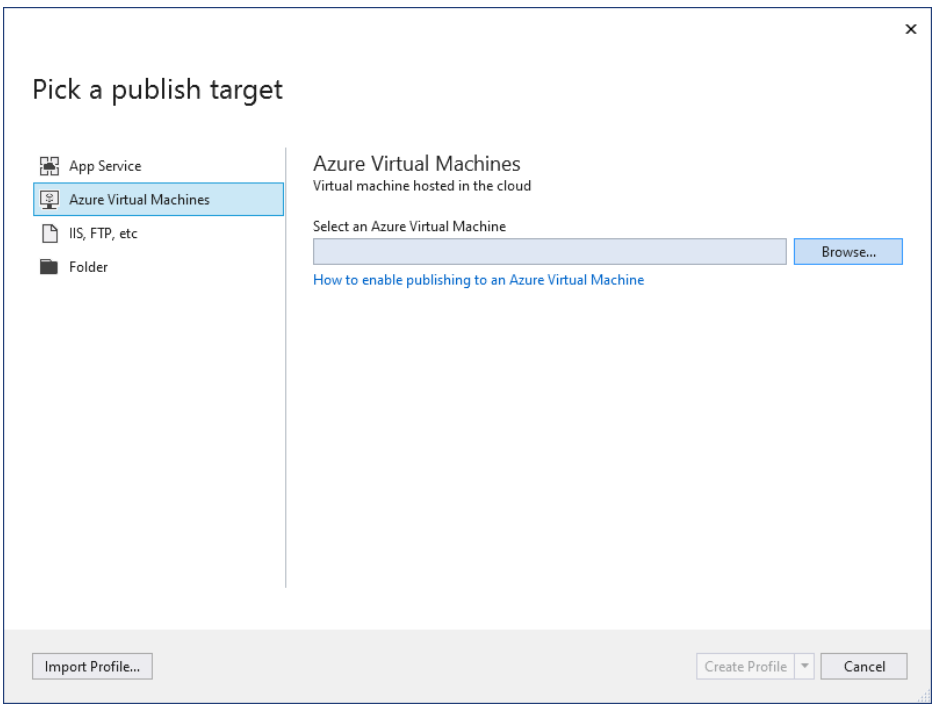


Figure 1: Publish to Azure Virtual Machines from Visual Studio

Of course, we don't want to deploy applications directly to production using Visual Studio. To have more control over the deployment, it should be done from our build server as part of the CI/CD (continuous integration and continuous deployment) process. Microsoft's build server in the cloud is named Azure Pipelines and it's a part of Azure DevOps.

To build and deploy a web application with Azure Pipelines, the source code must be published to a remote Git repository in Azure Repos, GitHub or elsewhere (TFVC – Team Foundation Version Control and SVN – Subversion repositories are also supported).

First, a build pipeline must be created to build the application and run the tests. Although its configuration is saved as a YAML file, most of it will be generated automatically based on the selected source code repository and the type of the application (ASP.NET or ASP.NET Core).

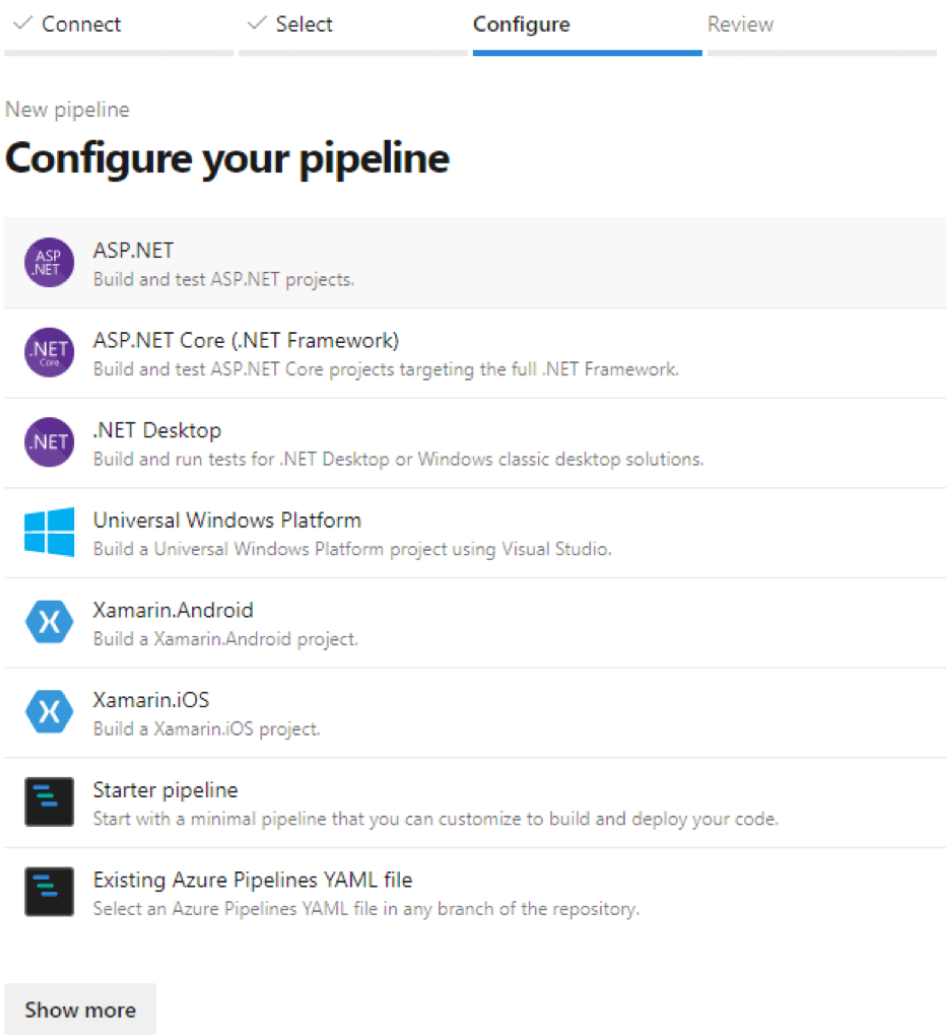


Figure 2: Selecting the application type for the build pipeline

Only one additional task must be added to the end of the file and there's even a graphical user interface for that. The *Publish build artifacts* task will make the result of the build available to the release pipeline which will deploy the application to the virtual machine.

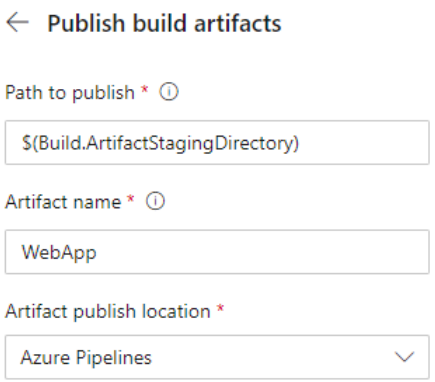


Figure 3: Configuring the Publish build artifacts task

The task wizard will insert the corresponding YAML snippet at the current cursor location in the file. This should result in the following final build pipeline configuration (in case of ASP.NET application):

```
# ASP.NET
# Build and test ASP.NET projects.
# Add steps that publish symbols, save build artifacts, deploy, and more:
# https://docs.microsoft.com/azure/devops/pipelines/apps/aspnet/build-aspnet-4

trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'Any CPU'
  buildConfiguration: 'Release'

steps:
- task: NuGetToolInstaller@1

- task: NuGetCommand@2
  inputs:
    restoreSolution: '$(solution)'

- task: VSBuild@1
  inputs:
    solution: '$(solution)'
    msbuildArgs: '/p:DeployOnBuild=true /p:WebPublishMethod=Package
/p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true
/p:PackageLocation="$(build.artifactStagingDirectory)"'
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'
```

```
- task: VSTest@2
  inputs:
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'

# Added by the Publish build artifact task wizard
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory) '
    ArtifactName: 'WebApp'
    publishLocation: 'Container'
```

The release pipeline will use Web Deploy for deployment just like Visual Studio does. There's detailed documentation available for configuring it. The process consists of the following steps:

- Creating an Azure Pipelines deployment group and adding the target server to it by installing an agent on the virtual machine using the provided PowerShell script.
- Installing a free extension for Azure Pipelines for deploying applications using Windows Remote Management (WinRM) and Web Deploy.
- Configuring the release pipeline to deploy the build artifact from the previously created build pipeline to the previously created deployment group.

With everything configured correctly, each new commit to the selected Git repository will trigger a new build. If it succeeds and all tests pass, the release pipeline will deploy the application to the virtual machine.

As you can see, the Platform-as-a-Service approach doesn't provide a lot of benefit in comparison to hosting applications on *on-premise* servers, except for the obvious fact that we don't need to worry about the hardware anymore. To benefit more by migrating to the cloud and have a simpler deployment processes, we will need to choose one of the other approaches.

Platform as a Service (PaaS)

The Platform-as-a-Service approach is a great compromise that doesn't require too much modification to our web applications or development processes, but still makes the hosting and the deployment process much simpler than the Infrastructure-as-a-Service approach. Instead of having to manage the virtual machine, we only interact with the web server software which is exposed as a cloud service.

In Microsoft Azure, the service is called Azure App Service Web Apps. When creating a new instance, we start by selecting the stack of our application. For .NET applications different versions of the .NET framework and .NET Core are available. For .NET Core applications, we can also choose between the Windows and Linux operating systems.

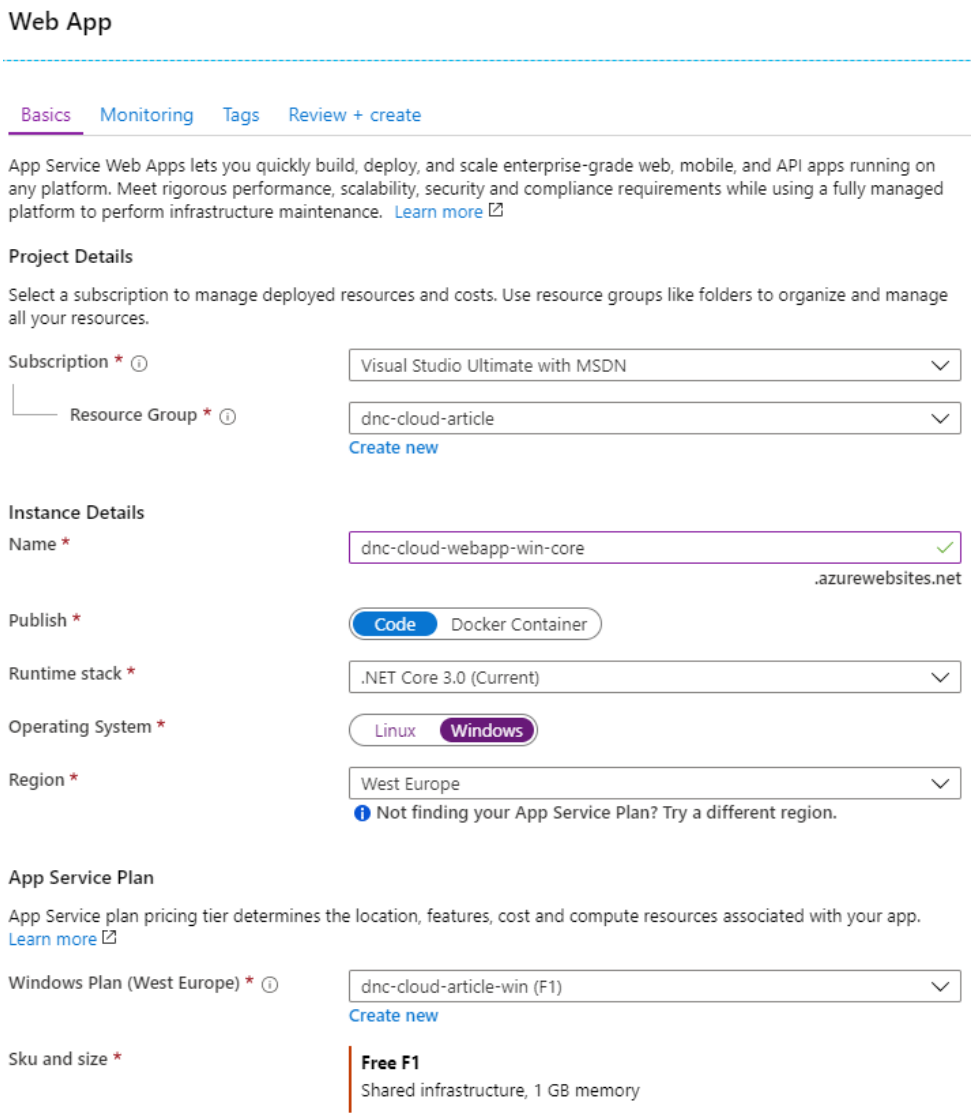


Figure 4: Configuring a new Azure App Service Web App

No matter the choice, an ASP.NET or ASP.NET Core web application can be easily deployed to the Azure App Service Web App instance using Visual Studio 2019.

From the *Publish* window, a wizard can be launched to select the target instance from those available in the selected Azure subscription. After clicking *Publish*, the application files will be copied to Azure and the deployed web application will be launched in the default web browser.

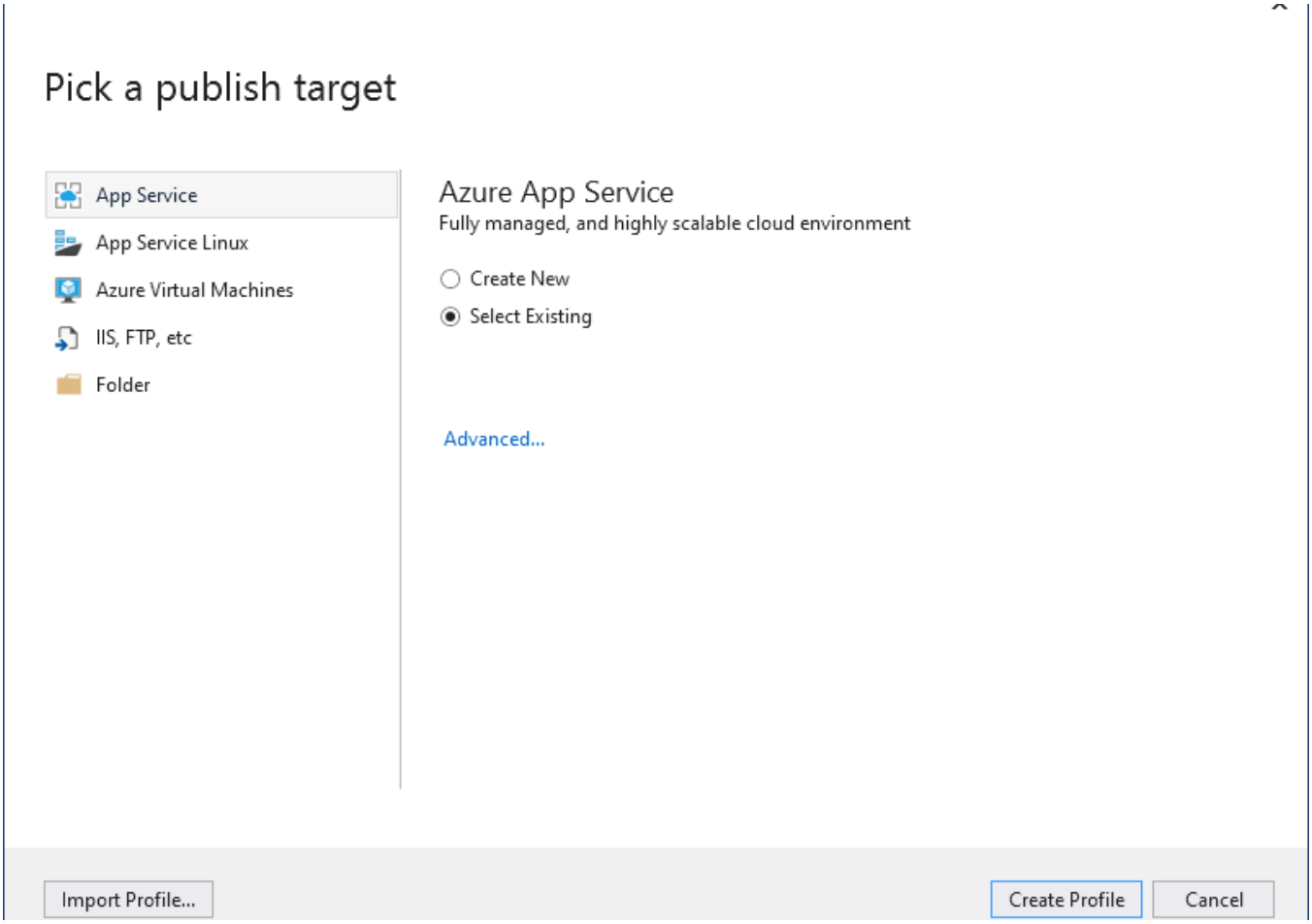


Figure 5: Publish to Azure App Service Web App

Of course, deployment to Azure App Service Web Apps is also supported from Azure DevOps. The build pipeline will remain configured the same as it was for deployment to Azure Virtual Machines. Only the release pipeline will be different.

When creating a new release pipeline for deploying to Azure App Service Web Apps, we can start with the Azure App Service deployment template which is mostly preconfigured for our needs.

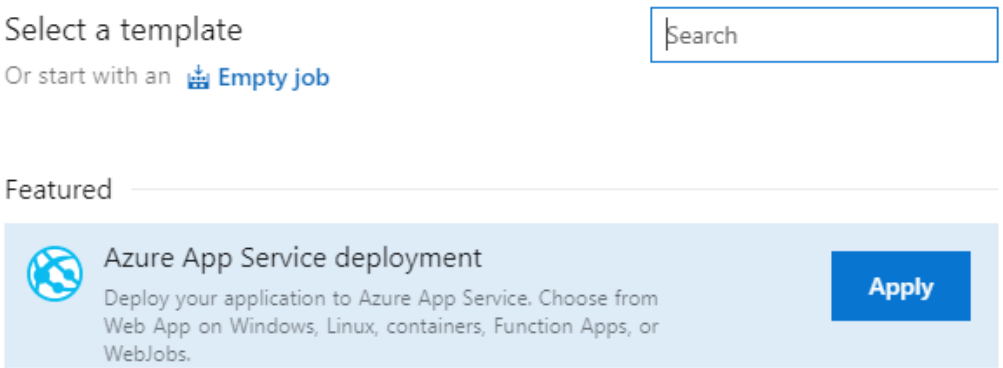


Figure 6: Azure App Service deployment release pipeline template

The most important part remaining is the configuration of the deployment task in the template. We must authorize a connection to the selected *Azure subscription* and then choose the *App Service type (Web App on Windows or Web App on Linux)* and the *App Service name* (from the list of services in our subscription).

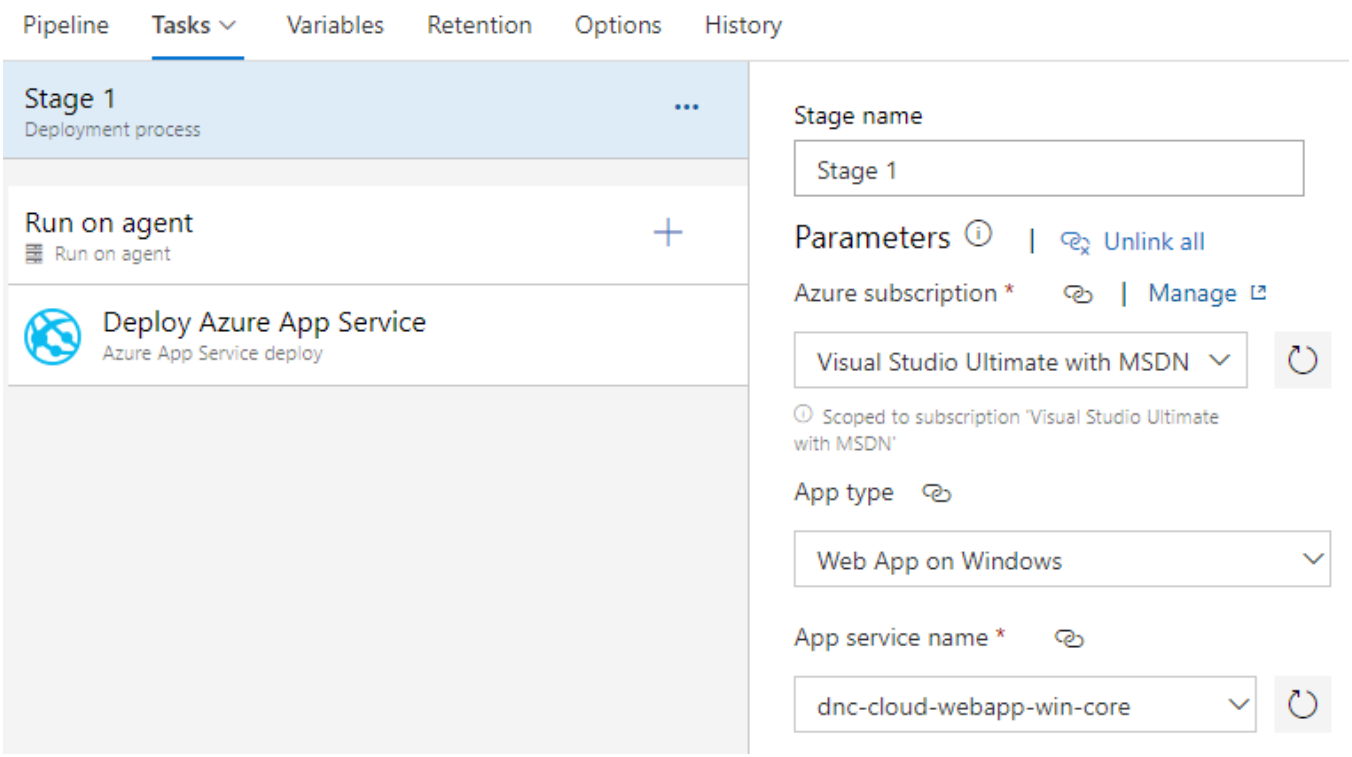


Figure 7: Configuring the deployment task

Just like in the case of deploying to Azure Virtual Machines, we need to select the artifact from the build pipeline we want to deploy. Once we set that up, the web application will be deployed to the selected Azure App Service Web App instance whenever a new commit is pushed to the corresponding Git repository.

As you can see, deploying to Azure App Service Web Apps is much simpler than to Azure Virtual Machines. However, since the server is preconfigured, the application is running in its sandbox and there's no way to install additional software components on the hosting server. **All required dependencies must be deployed as part of the application**.

Also, there's no guarantee that any files stored locally from the application will persist. The application might be migrated to a new server and at that time, only application files that were deployed to the service, will be preserved. So any application state must be stored elsewhere, e.g. in a separate Azure Storage instance for files and unstructured data, and in an Azure SQL Database or Azure Database for PostgreSQL instance for relational data.

If this is too restrictive, **deploying to one or more Docker containers might be a better fit**.

Cloud-Native Applications

Docker containers can be thought of as **lightweight virtual machines**. They still provide an isolated environment for the software running inside them but share the kernel of the operating system they are running on. Hence, they require less resources than virtual machines.

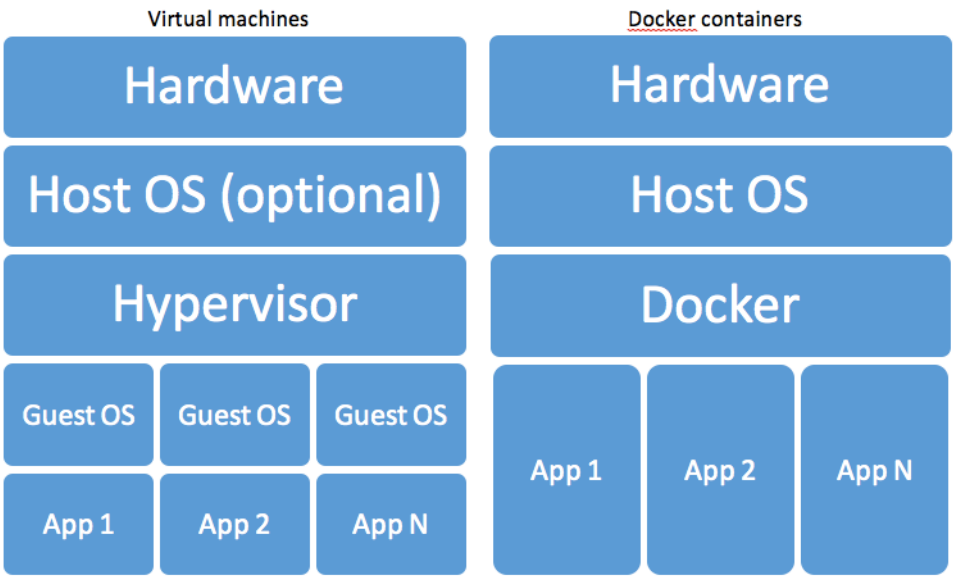


Image 8: Comparison between applications hosted in virtual machines (left) and Docker containers (right)

Azure App Service Web App for Containers is a variation of Azure App Service Web Apps which runs the web application from a Docker image instead from the files deployed to it. It requires the Docker image with that application to be published into a container registry. The one in Azure is named Azure Container Registry.

To publish an ASP.NET Core application to a container registry (and from there deploy to Azure App Service Web App for Containers), it must be configured correctly so that the build pipeline can create a Docker image with it. The simplest way to achieve that for an ASP.NET Core application is to *Enable Docker Support* when creating a new project from the Visual Studio 2019 template.

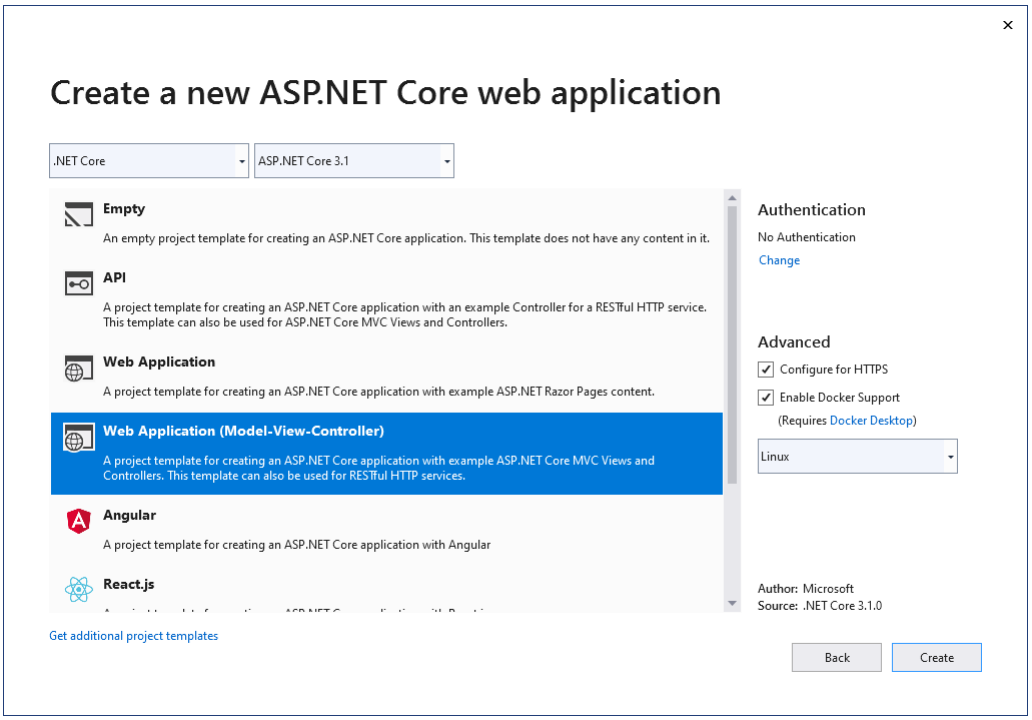


Figure 9: Enabling Docker support in a new web application

Visual Studio 2019 will transparently take care of running and debugging such an application from a Docker container on the development machine as long Docker Desktop for Windows is installed. However, there's no built-in functionality in Visual Studio 2019 to publish the resulting Docker image to a container registry.

On the other hand, Azure DevOps doesn't lack such support. In a typical setup, the build pipeline will be responsible for publishing the Docker image to the container registry. Although there is a Docker template for a new build pipeline, it unfortunately doesn't include all the necessary steps preconfigured. Most of the pipeline will need to be configured manually:

- To allow publishing an image to a container registry, a new service connection to it must first be created in the Azure DevOps *Project Settings*. If you want to use an Azure Container Registry, you need to create one beforehand.

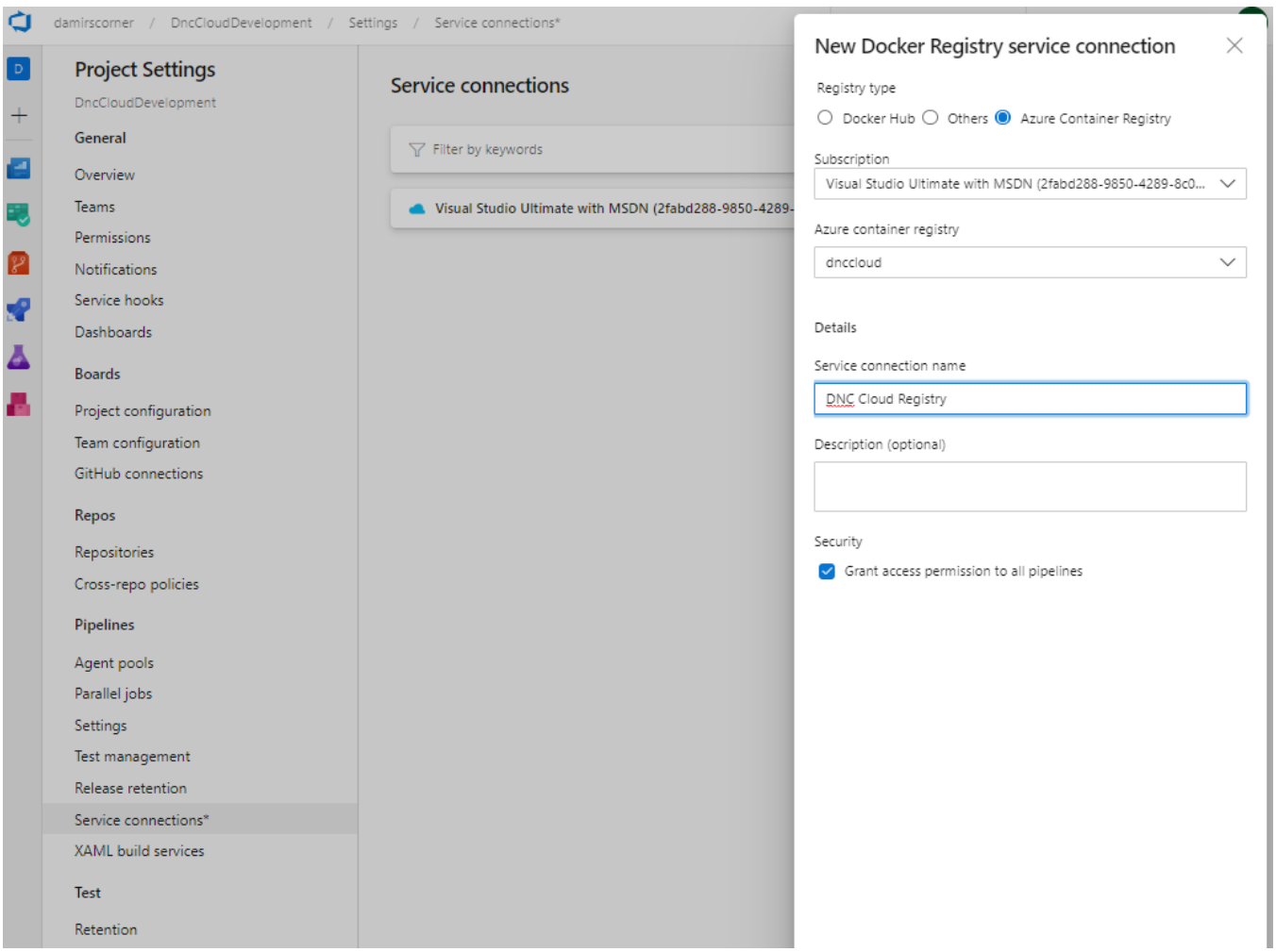


Figure 10: Creating a new service connection to a container registry

- With that in place, a new pipeline can be created; preferably from the *Starter pipeline* template because the *Docker* template in its current state, needs to be changed considerably to be helpful.

- The existing steps from the template must be replaced with the *Docker* task available in the assistant. Only the previously connected *Container registry* must be additionally configured for the task along with a unique Container repository name for the application. In Docker terminology, a **repository** is a collection of images with the same name but different tags. **Tags** correspond to versions of these images. A container **registry** will typically host multiple repositories.

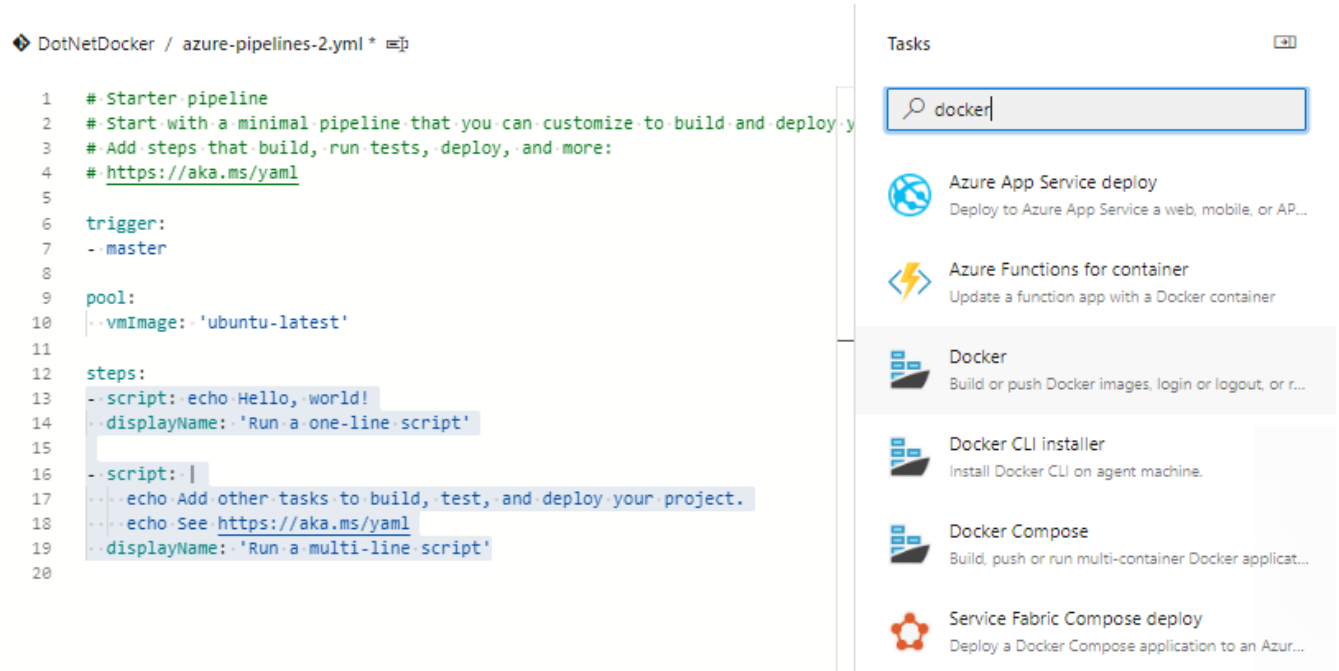


Figure 11: Adding a Docker task to the build pipeline

- A Docker image is built from a `Dockerfile`. The step created in the pipeline assumes that the file will be placed in the root folder of the code repository. Since it is placed in the project folder by the Visual Studio template, the working directory must be set to root using the `buildContext` input or the build will fail:

```
- task: Docker@2
  inputs:
    containerRegistry: 'DNC Cloud Registry'
    repository: 'dnc-cloud-article'
    command: 'buildAndPush'
    Dockerfile: '**/Dockerfile'
    buildContext: .
```

When the newly configured build pipeline is run, it will publish the image to the container registry so that it can be used by the Azure App Service Web App for Containers which requires the *Registry*, *Image* (corresponds to repository name) and *Tag* to be specified.

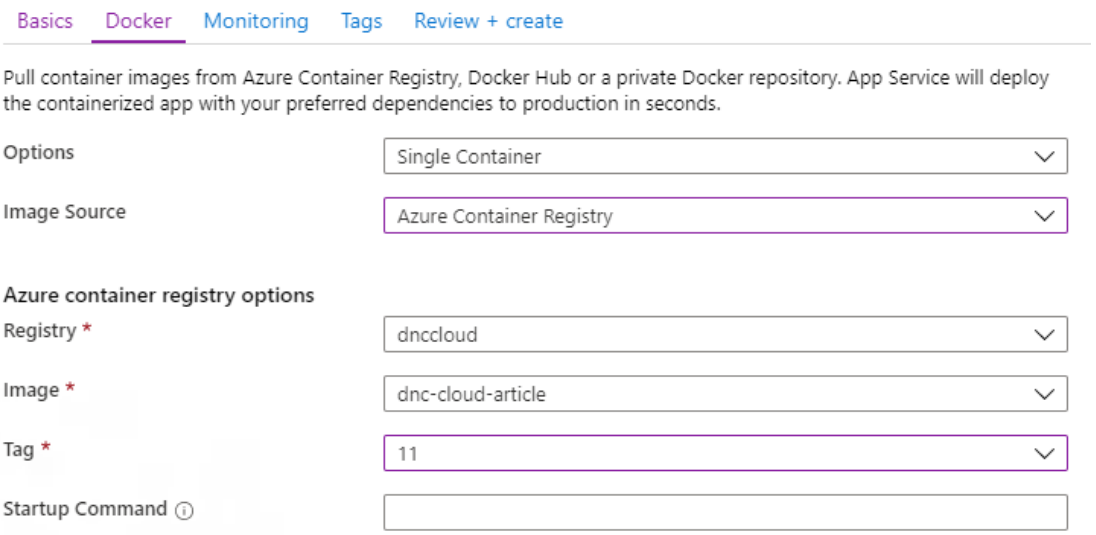


Figure 12: Configuring the image for the Web App for Containers

This is already enough to get the application running in Azure. The build pipeline publishes the container image with the application in the container registry. In the Azure App Service Web App for Containers configuration, we selected a previously published specific version (i.e. tag in Docker terminology) of the image to deploy. This version of the application can already be accessed by opening the URL of the created Web App in the browser.

For automatic deployment of new versions, an Azure DevOps release pipeline must be configured.

Again, the *Azure App Service deployment* template will be used, but this time *Web App for Containers (Linux)* must be selected as the *App type*. In addition to selecting the Azure subscription and the target *App service name*, the Docker image to deploy must be specified in the Registry or Namespace (*Login server* of the container registry found on its *Overview* page in Azure Portal) and *Repository* (i.e. repository name as set earlier) fields.

Editorial Note: *If you are new to Azure DevOps, I strongly recommend these* ***Azure DevOps tutorials****.*

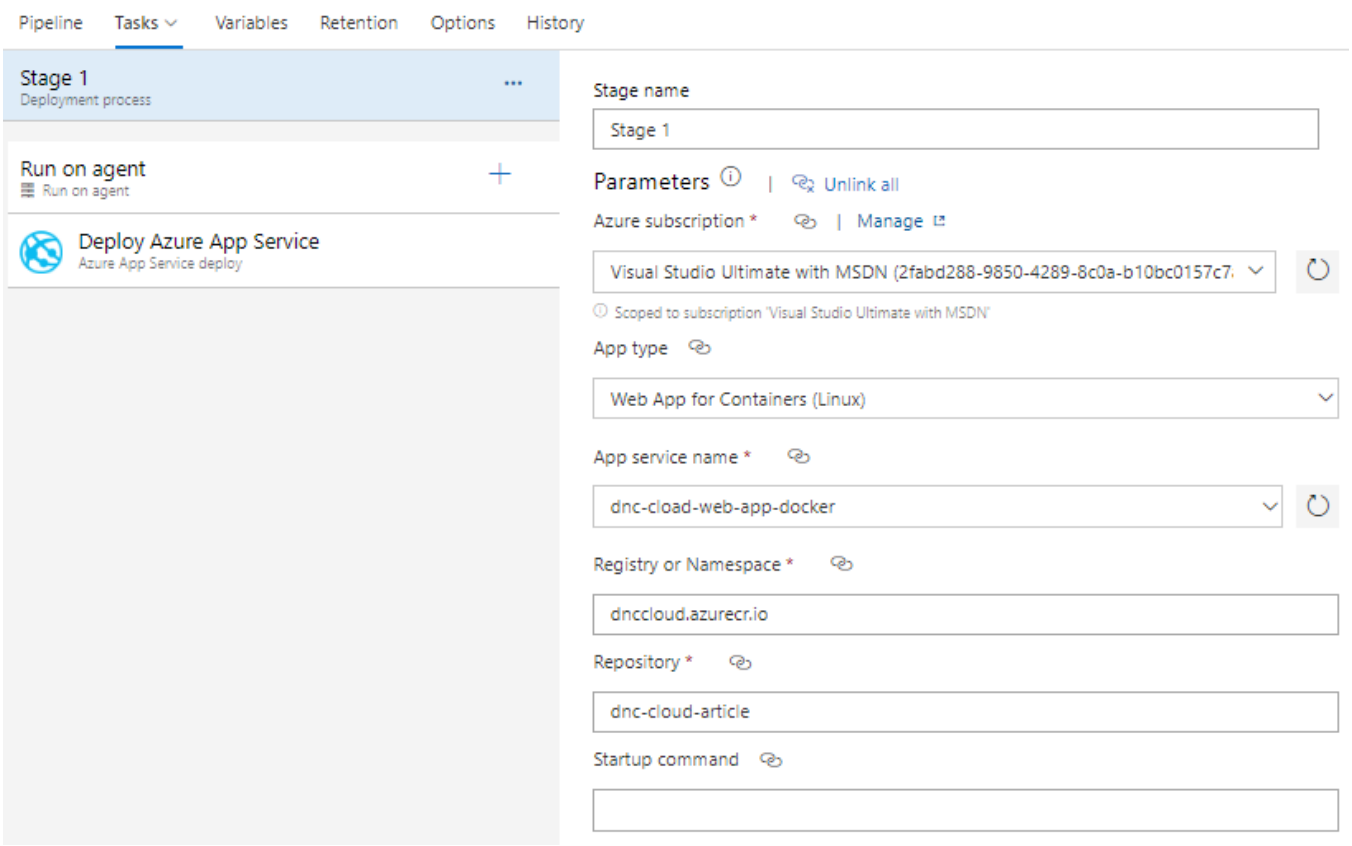


Figure 13: Release pipeline deployment step configuration

The *Azure Container Registry* should act as the artifact source for the release pipeline; configured with details pointing at the Docker image published by the corresponding build pipeline. This will ensure that the release pipeline is triggered every time the build of the Docker image succeeds.

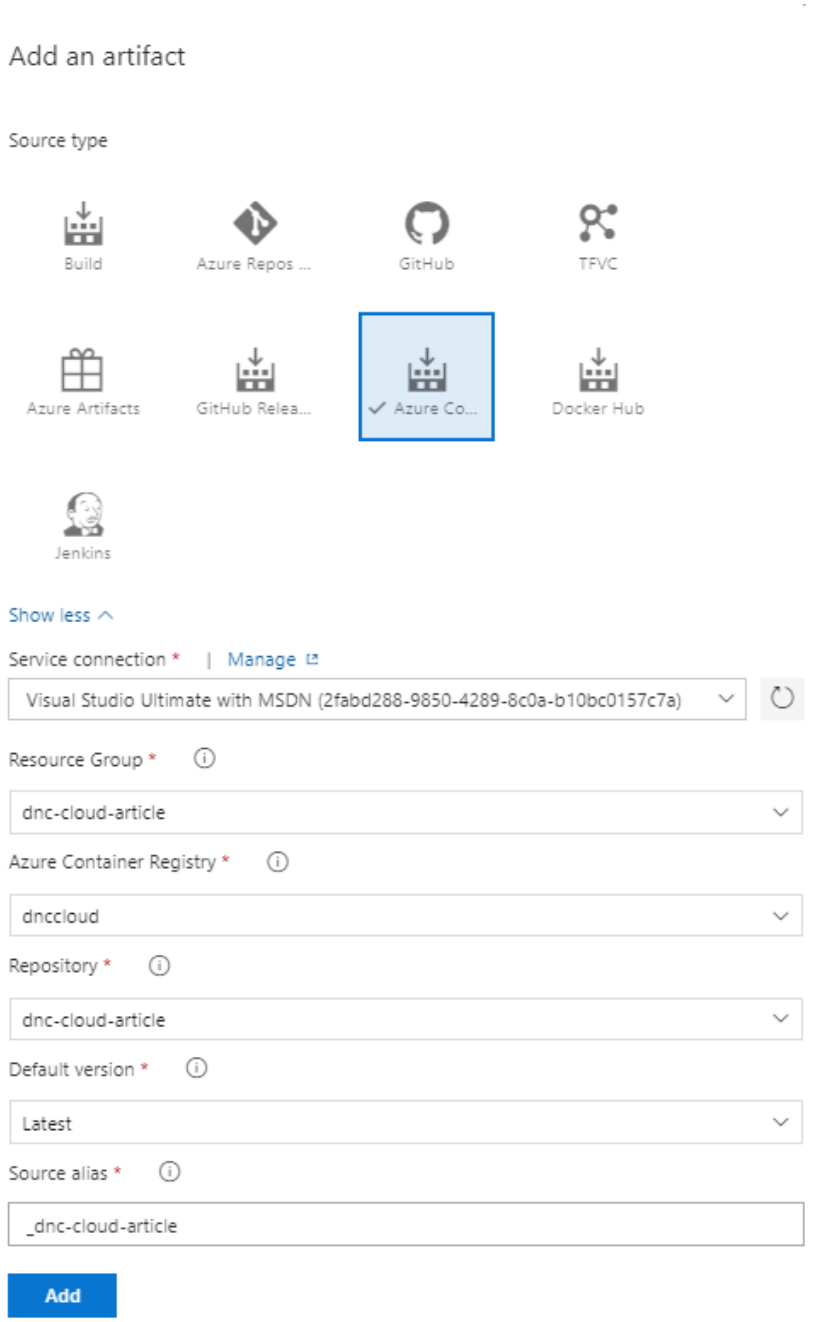


Figure 14: Release pipeline artifact configuration

Advantages of Docker Images

If you're not familiar with Docker images, you might wonder what are the advantages of this deployment approach over simply deploying the web application file to an Azure App Service Web Apps instance.

For one, Docker images are in no way specific to the service where they are hosted. Changing the hosting service or the provider in the future will only affect the release pipeline, keeping the application code and build pipeline unchanged.

More importantly, Docker images can include additional software dependencies apart from .NET Core runtime, although the default Docker image as configured in the `Dockerfile` created by the Visual Studio template, doesn't have any.

If these dependencies are standalone services and not components directly used by the application, they will typically be deployed as separate Docker images. The multiple images required by an application and their interaction are configured using a docker-compose.yml file. Azure App Service Web App for Containers already has preview support for using such a Docker compose file instead of a single Docker image.

However, as the number of required Docker images grows, a dedicated service for hosting Docker images becomes a better choice. In Azure, there are two available: Azure Kubernetes Service (a managed instance of a standard container orchestration service) and Azure Service Fabric (recommended choice when fully supported Microsoft's technology stack is required).

When this point is reached, it makes sense to use separate Docker images not only for third party dependencies, but also for different internally developed components of the application. To allow for this, each one of them needs to be developed as a standalone service. Such services are usually called **microservices**. They are a key part of the so-called **cloud-native applications**, i.e. applications which are built from the ground up for hosting in the cloud so that they can be easily horizontally scaled.

Serverless Computing

All deployment models described so far include server resources available and payed (paid) for full-time (either directly in Infrastructure-as-a-Service model or indirectly in Platform-as-a-Service and Cloud-Native Applications models).

Serverless computing is different from all of those in that server resources are only consumed and payed for, when they are used. The idea of not choosing specific server resources in advance and paying for those is also where the name serverless computing originates from.

In Azure, there are two serverless services available. Azure Container Instances is a serverless hosting model for Docker containers. Since it doesn't include any advanced orchestration capabilities, it's more like Azure App Service Web App for Containers than Azure Kubernetes Service (AKS) or Azure Service Fabric. The main difference is that the resources are paid for based on actual consumption not the selected performance specifications.

The second serverless service is Azure Functions. This one is quite similar to Azure App Service Web Apps. However, it doesn't support hosting of regular ASP.NET or ASP.NET Core web applications. Instead, a specific Azure Functions application model must be used.

As the name implies, Azure Functions applications consist of individual functions. These can be invoked using different triggers, e.g. a timer, an HTTP request, a message arriving in a queue etc. Such an application model is often also called **Function-as-a-Service**. Of course, C# is one of supported languages. Azure Functions runtime version 1.x requires development in the .NET framework, later runtimes (versions 2.x and 3.x) require development in .NET Core. All runtimes are still fully supported.

Visual Studio 2019 has built-in support for developing Azure Functions applications in C# for any runtime version which also includes running and debugging them locally and publishing them directly to Azure.

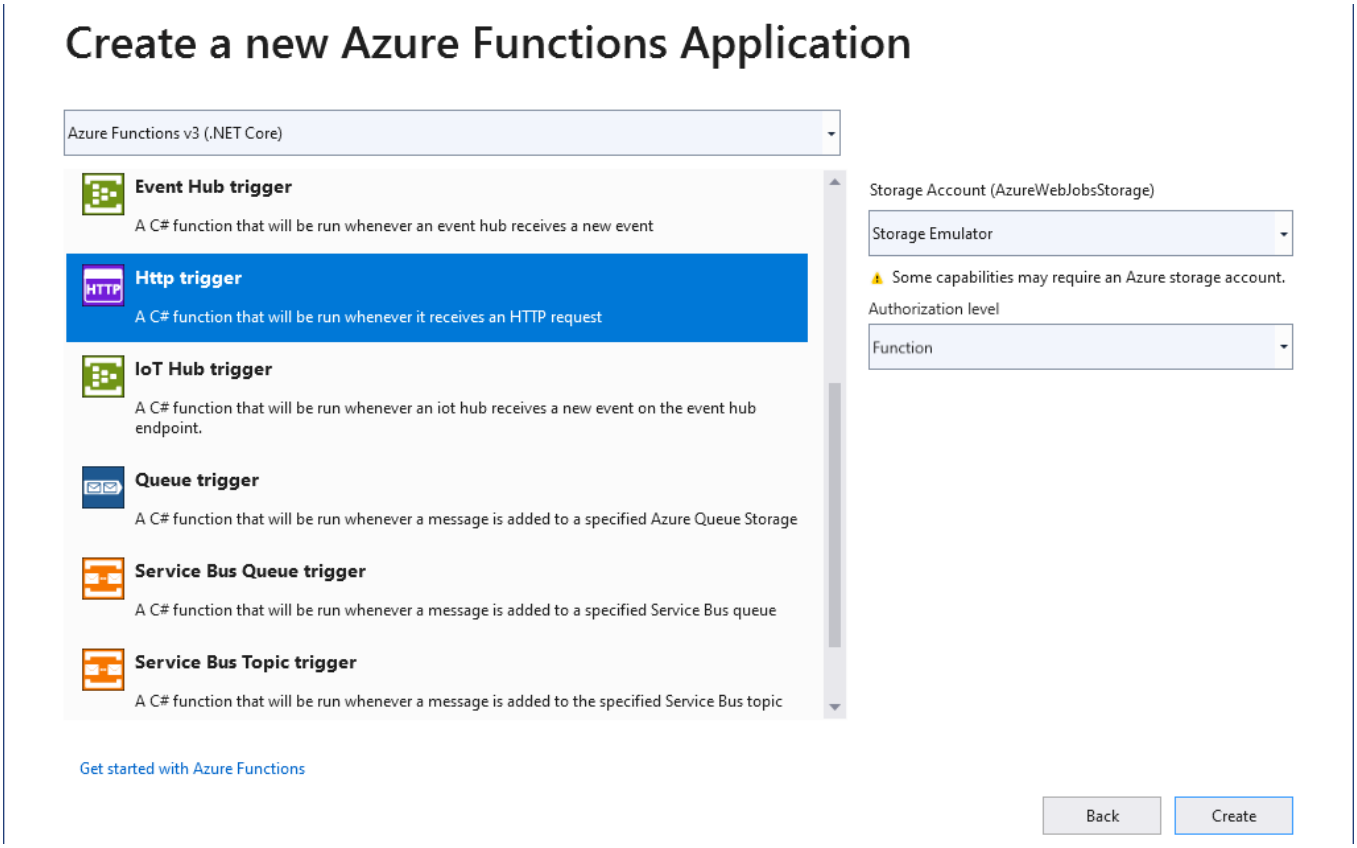


Figure 15: Selecting Azure Functions runtime version and function triggers

When creating a new application using the Azure Functions project template, the runtime version needs to be selected first. For every new function, the type of trigger to bind it to, must be selected. The selected trigger is then further configured using attributes as can be seen from the following code snippet:

```
[FunctionName("Function1")]
public static async Task<IActionResult> Run(
  [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
  HttpRequest req, ILogger log)
{
  log.LogInformation("C# HTTP trigger function processed a request.");

  string name = req.Query["name"];

  string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
  dynamic data = JsonConvert.DeserializeObject(requestBody);
  name = name ?? data?.name;

  return name != null
    ? (ActionResult)new OkObjectResult($"Hello, {name}")
    : new BadRequestObjectResult("Please pass a name on the query string or in the
    request body");
}
```

Once the application is ready, it can be deployed to Azure from the *Publish* window in Visual Studio just like a regular web application. In the wizard, an Azure Functions application previously created in Azure Portal can be selected, or a new one can be created in the selected Azure subscription directly from Visual Studio.

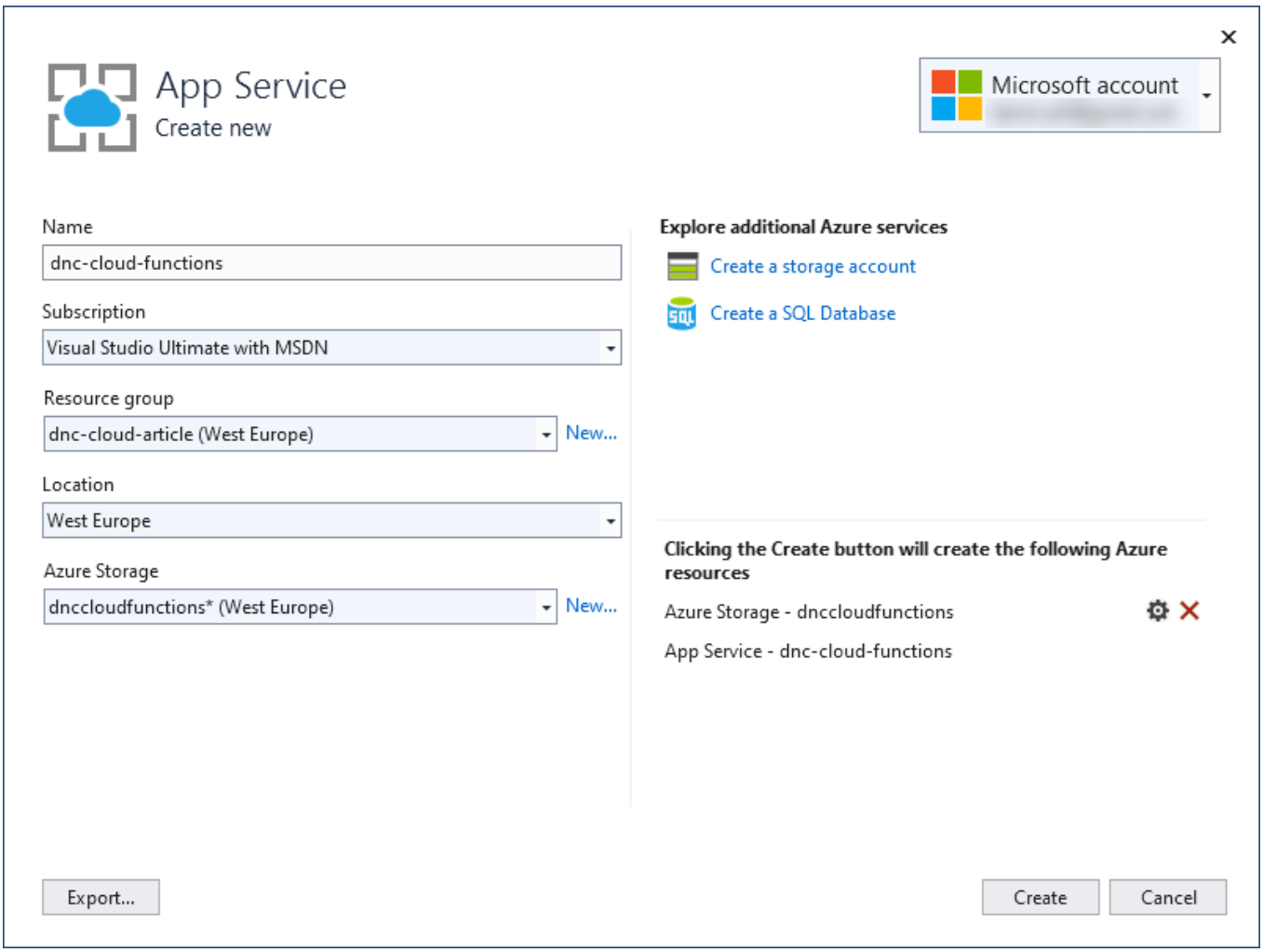


Figure 16: Creating a new Azure Functions application inside Visual Studio

Since a live application is usually not deployed directly to production from Visual Studio, there's support for building and deploying Azure Functions applications also in Azure DevOps. The build pipeline for that should be created from the *.NET Core Function App to Windows on Azure* pipeline template.

It creates a mostly preconfigured pipeline which only requires us to select the target Azure Functions application to deploy the application to. The reason for that is that unlike many other build pipeline templates, this one also includes a step to publish the application and not only to build it.

To trigger application deployment on every commit, this should be good enough. But to take advantage of additional functionalities provided by release pipelines such as manual approvals, a separate release pipeline needs to be created.

In this case, we first need to delete the second stage of the build pipeline which does the deployment and only keep the first stage which builds the application and publishes the artifact to be consumed from the release pipeline. We can create the release pipeline from the *Deploy a function app to Azure Functions* pipeline template and configure it by selecting the build artifact from the build pipeline as the trigger and an Azure Functions application as the deploy target.

Conclusion

It's easy to think of the cloud as just a deployment model for your application.

In a way, it is exactly that. However, cloud services offer much more than having virtual machines without your own hardware. If you're migrating an existing application to the cloud, moving the virtual machine from your on-premise hardware to the cloud, might be the only option.

But there are many other options which can be simpler and more cost-effective if they fit your requirements. Especially, if you're creating a brand-new application, it's a good idea to consider the services available and choose the most appropriate one before you start the development. This choice is likely to affect the way you need to develop your application even if you already know that you want to stay in the .NET ecosystem you're familiar with.

Microsoft® MVP Most Valuable Professional

Damir Arh

Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

*Thanks to **Daniel Jimenez Garcia** for reviewing this article.*

Mahesh Sabnis

USING AZURE COGNITIVE SEARCH APIS IN AN ANGULAR APPLICATION

What is Azure Cognitive Search?

Azure Cognitive Search is a **Search-as-a-Service** cloud solution in Microsoft Azure. This service provides tools and APIs for developers to add rich search experiences to their application.

The Azure Cognitive Search Service can build indexing on the Data Source so that it can be queried from application code using end user's inputs. As defined in the official documentation, *in Azure Cognitive Search, an index is a persistent store of documents and other constructs used for filtered and full text search.*

Azure Cognitive Search Service uses AI enrichment capability on indexing, which can be used to extract text from images, blobs and other unstructured data sources, and make the search richer. The text extraction is implemented through cognitive skills that is attached to an indexing. The cognitive skills use **Natural Language Processing (NLP)** and **Image Processing**.

NLP includes language detection, key phrase extraction, text manipulation, etc. The Image processing skills use Optical Character Recognition (OCR). Using AI and NLP is beyond the scope for this article, but those interested can read more about it from this link.

Azure Cognitive Search Service functionality is provided using **.NET SDK** and **REST API**. Since the Azure Cognitive Search Service runs in the cloud, we need not worry about the availability and infrastructure, as its managed by Microsoft.

Figure 1 gives an idea of the Azure Cognitive Search Service in the application.

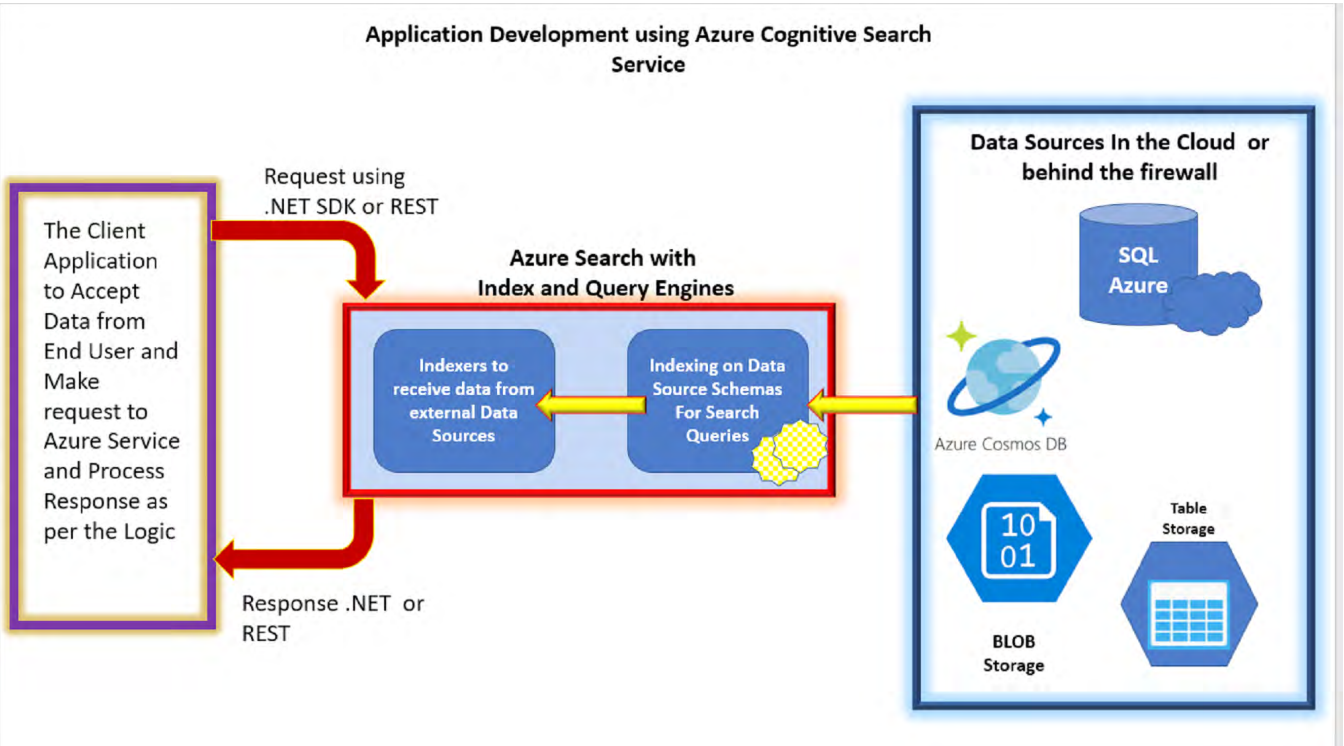


Figure 1: Application development approach using Azure Cognitive Search Service

Inspiration for this tutorial

An organization where I am working for as an Azure consultant, had a requirement to integrate enterprise

search features into their existing application. They wanted to provide a high-responsive search experience to the end-users.

The requirement was to consolidate data across various data stores on Microsoft Azure and provide search functionality on it based on end-user's input. The UX requirement expectation was a highly responsive UI with fast search. After doing a rigorous research and adopting complex querying mechanisms, as well as NoSQL solutions, we decided to use Azure Cognitive Search service. We soon found that it had the capability to address most of my client's requirements. We tried Azure Cognitive Search Service on various complex data structures and also on a high volume of data – with success!

Having delved into a relatively new piece of technology (I am no search expert) and getting a good experience of Azure Cognitive Search Service, I persuaded Suprotim to let me write on it for this month's magazine edition.

Scenarios of using Azure Cognitive Search Service

- While handling search operations on a high volume of data from different data sources (e.g. Azure SQL Database, SQL Server on Azure VMs, Cosmos DB, Azure BLOB Storage, Azure Table Storage, etc.), writing complex join queries increases complexity of the operations.

  In such cases, you can consolidate the data from various data sources in JSON documents formats and use Azure Cognitive Search Service Indexers to load the data in index to ease search operations. For example, let's assume you want to search 'Order details' across countries, regions, cities, customers, categories, etc, then consolidating data into a single store and searching this store using Azure Cognitive Search Service, will be helpful.

- If search is using Filters, AutoComplete, Suggestions, etc., then Azure Cognitive Search Service is a good option.

- Indexing unstructured text or extracting text from image. Please note that we need to use Azure Cognitive Search AI features over here.

Azure Cognitive Search Service provides the following features for search

- Free-form text search
  - Full-Text Search
  - Simple Query Syntax
  - Lucene Query Syntax

- Relevance
  - Simple Scoring

- Geo-Search

- Filters and facets
  - Faceted Navigation
  - Filters

- User Experience Features
    - Autocomplete
    - Search Suggestions
    - Sorting
    - Paging
    - Hit Highlighting
    - Synonyms

In this tutorial, we will implement the Azure Cognitive Search Feature using REST APIs along with Simple Query syntax and Lucene Query syntax. We will implement the application using the following steps:

1. Creating the Data Source using Azure CosmosDB
2. Creating Azure Cognitive Search Service
3. Creating Angular Client Application to access Azure Cognitive Search Service REST API

Creating the Data Source using Azure CosmosDB

As we have discussed in Figure 1, we need a Data Source for consolidating the data to perform search. I have used the simple Northwind database and the Orders Table in it. But since some of the columns from this table are reference columns, I have read data from the Order's table using the following query:

```
Select OrderID, Customers.ContactName as CustomerName, Employees.FirstName + ' ' +
Employees.LastName as EmployeeName,
OrderDate, RequiredDate, ShippedDate, Shippers.CompanyName as ShipperName,
Freight, ShipName, ShipAddress, ShipCity,ShipPostalCode, ShipCountry
from Orders, Customers, Employees, Shippers
where
Customers.CustomerID=Orders.CustomerID and Employees.EmployeeID =Orders.EmployeeID
and Shippers.ShipperID=Orders.ShipVia
```

Create a Cosmos DB Account. If you are new to CosmosDB, follow this link to read steps for creating Cosmos DB account.

Editorial Note: *For those who want to delve deeper into CosmosDB, I recommend reading* Azure Cosmos DB – Deep Dive *by Tim Sommer.*

Now create an **OrdersDb** database and two containers named as **AllOrders** and **SuppliersData**. To add data in Cosmos DB Containers, download Data Migration tool for CosmosDB.

If you are not aware about this tool, you can visit this tutorial to read more about it, and how to use it. Once you migrate data from Orders table to the AllOrders container, the JSON document data will be displayed in the AllOrders container. We will also read data from SuppliersData table (38000 rows) and migrate to the SuppliersData container. The query for reading data from the SuppliersData is as follows:

```
select SupplierID, CompanyName,ContactName,ContactTitle,Address,City,Country,Phone
from SuppliersData
```

Creating Azure Cognitive Search Service

Let's create an Azure Cognitive Search Service. Open the Azure portal using https://portal.azure.com. From the Azure portal, using **Create a Resource** we can create Azure Cognitive Search Service as shown in the following figure:

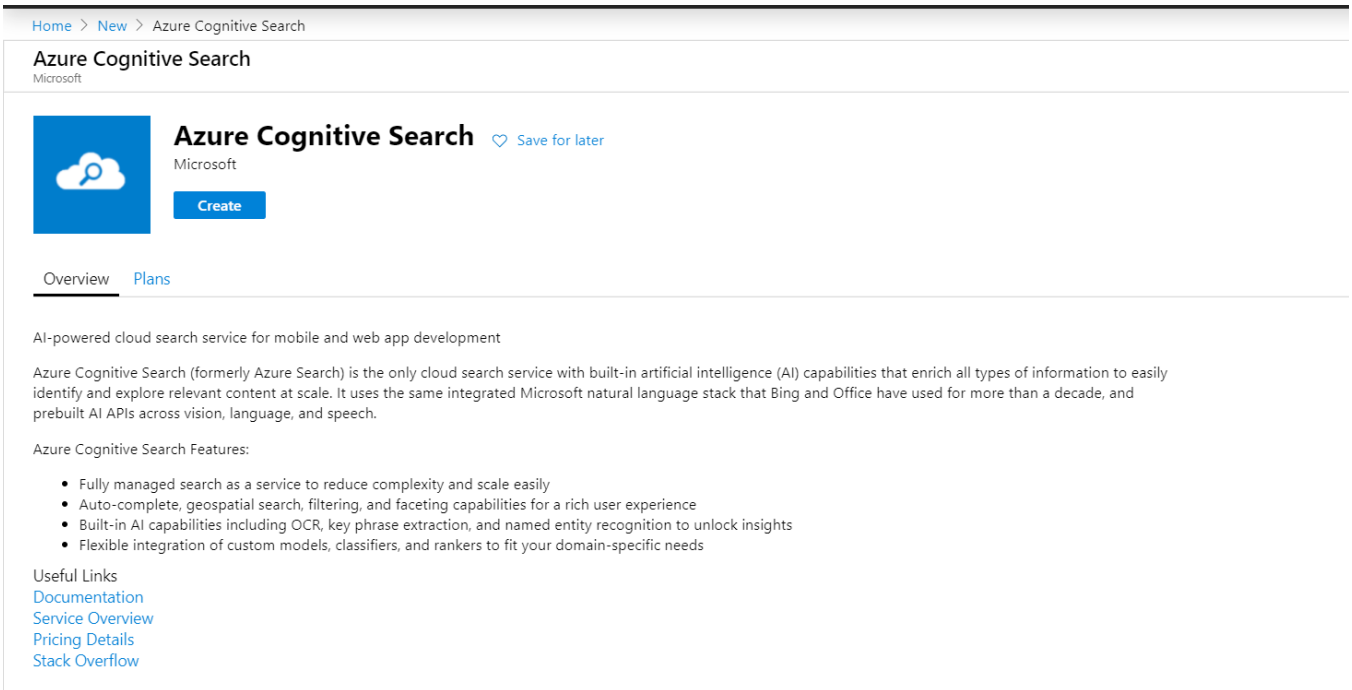


Figure 2: Creating a new Azure Cognitive Search Service

To create a service, provide the following information as shown in Figure 3.

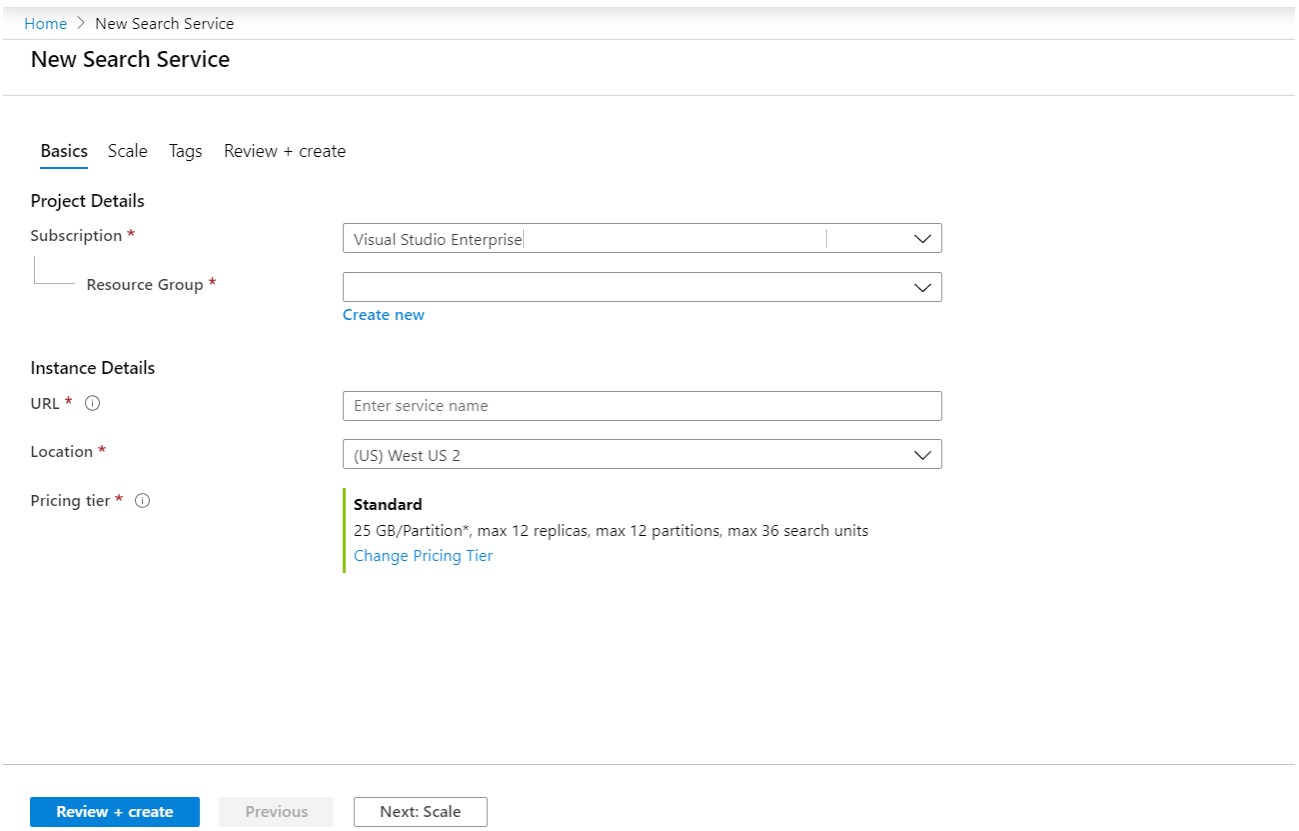


Figure 3: New Search Service

One important point to note is the **Pricing tier**. Azure Cognitive Search Service configuration is completely based on the pricing tSipliire. We can see the configuration details by clicking on the **Change Pricing Tier** link as shown in Figure 4.

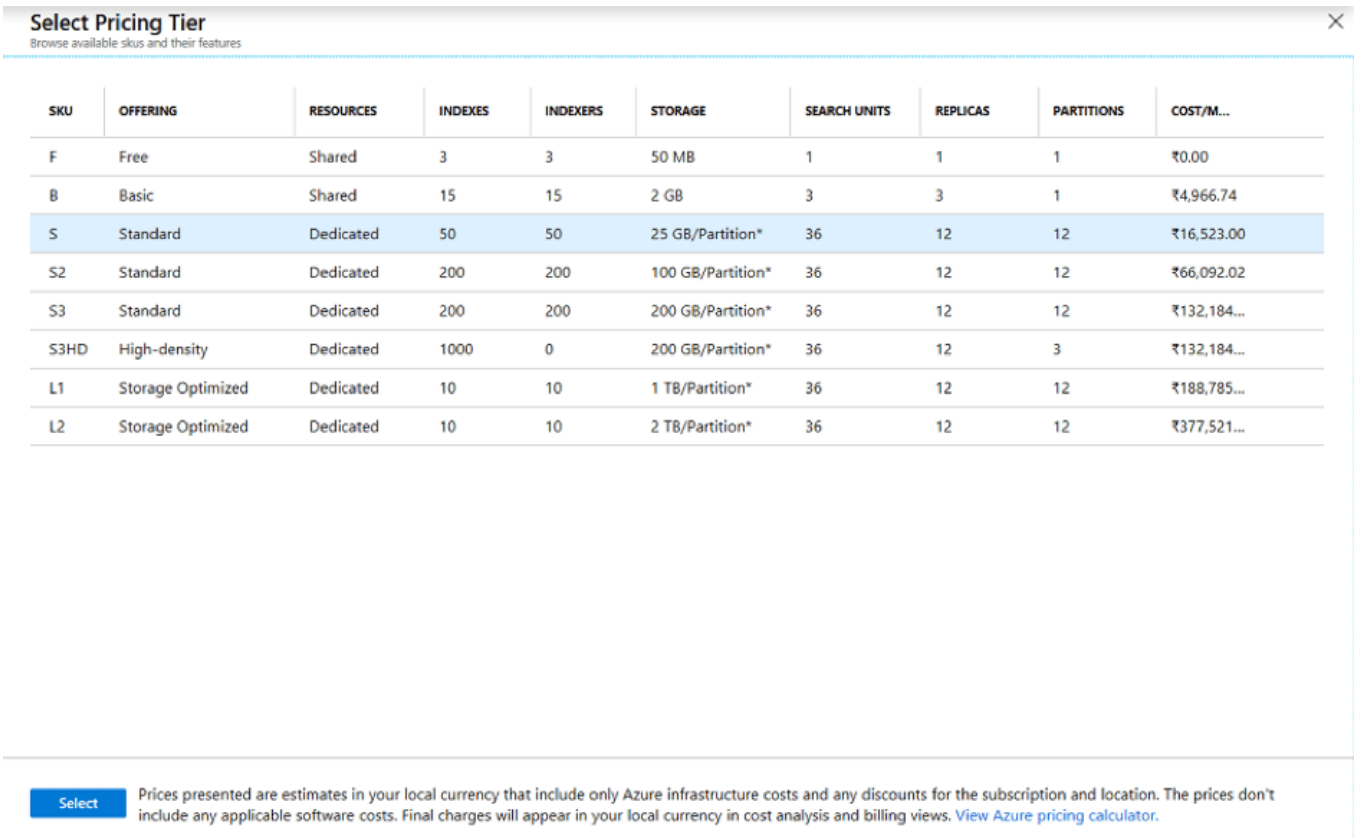


Figure 4: The Pricing tier showing the Azure Service Configure details

There are offerings for configuring Azure Cognitive Search Service. Service Indexes, Indexers count and Storage size are defined based on the offering.

Once the service is created, its details will be shown as seen in Figure 5.

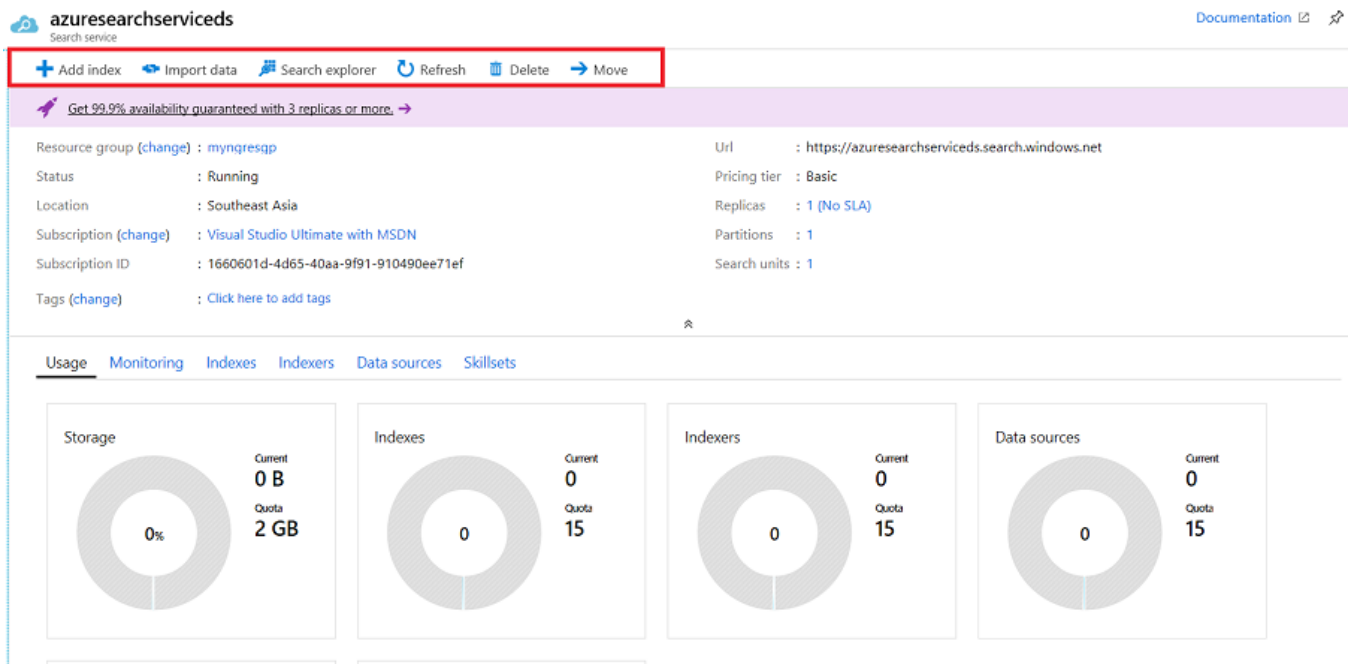


Figure 5: The Azure Service details

We need keys for accessing Search Service in the client application. These keys will be used to authorize the client application to access Azure Service. We can see these keys as shown in Figure 6.

Figure 6: Azure Cognitive Search Service Keys

To configure the data source for the search service, click on the **import data** link as shown in Figure 5. The **Connect to your data** tab option will be displayed, where you can select the data source as shown in Figure 7.

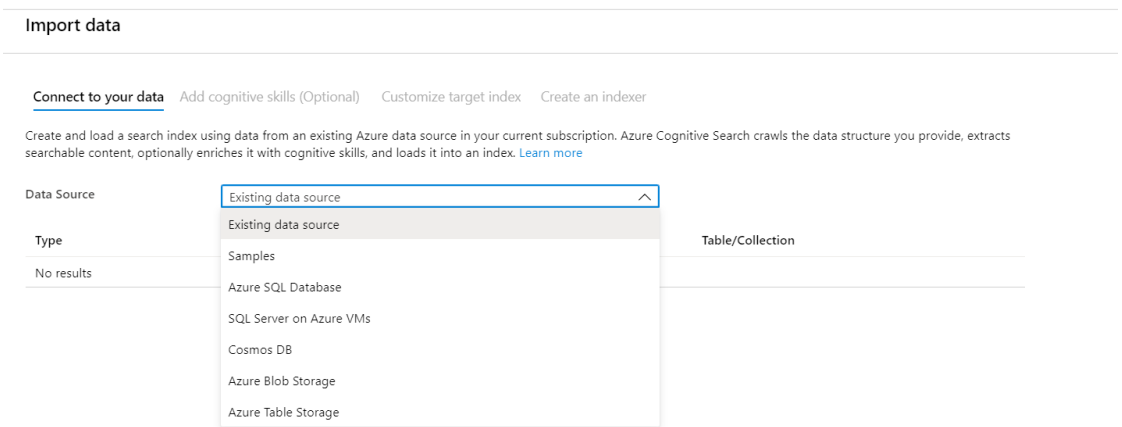


Figure 7: Connect to data source

Select CosmosDB from the dropdown. The **Connect to your data** tab will bring up a UI where we can enter data source information like Cosmos DB Account Key, Database name and Container name from the Cosmos DB as shown in Figure 8.

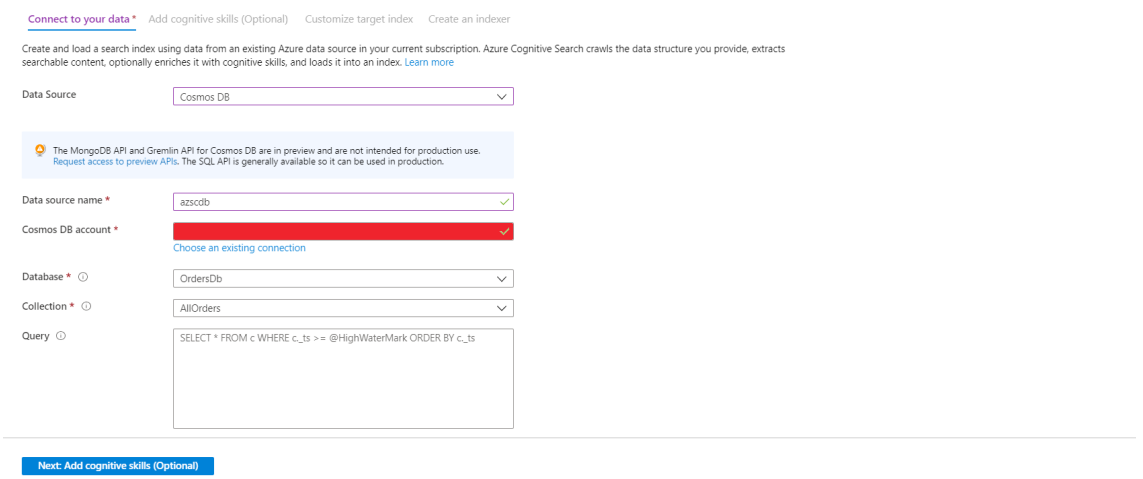


Figure 8: Data Source Configuration

Leave the Query TextBox as empty, we will be selecting all the columns from the source collection. Click on the **Next** button, here we can attach Cognitive Service with the Azure Cognitive Search Service. This is an optional step and we will keep it as-is without any changes. See Figure 9.

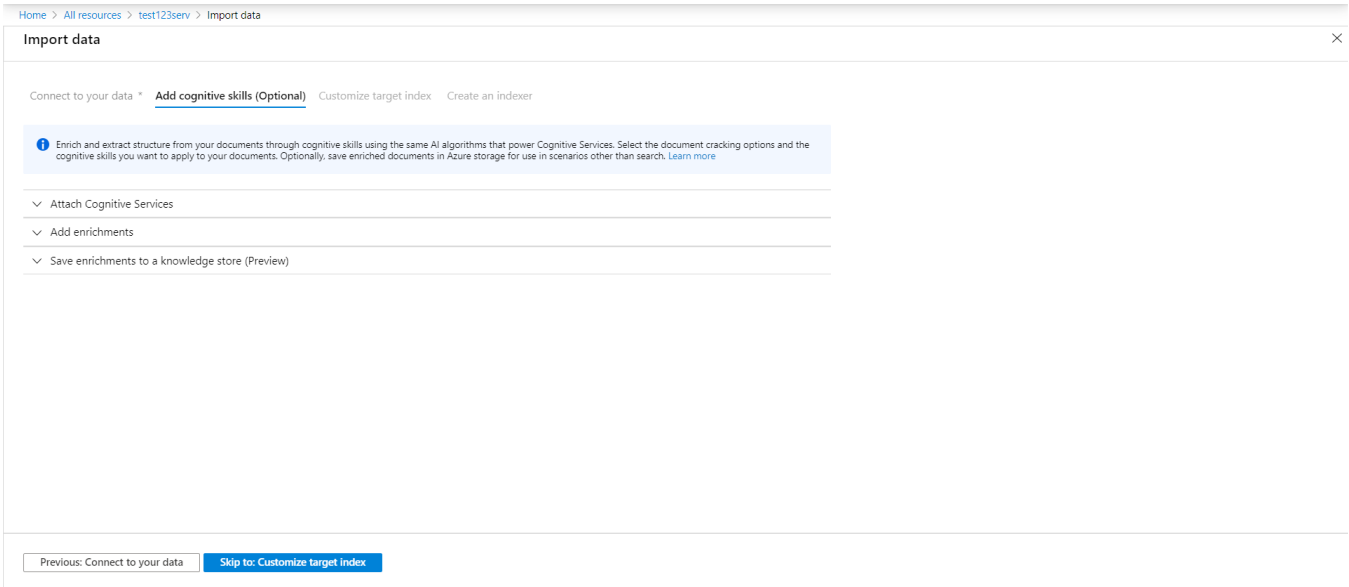


Figure 9: Cognitive Service Configuration

Click on the **Skip to: Customized target index button**.

Settings index configuration is the most important step for the Azure Cognitive Search Service. The **Index** is like a database table. This holds data from the source so that queries on the data can be accepted. We can configure the index by setting following configuration values:

1. **Index name** - is the unique index name. The index name can contain lower case characters, alphanumeric characters, digits.
2. **Key** - the index can have only one key field. This must be a string. This key represents the unique identifier for each document for every document stored in the index.
3. **Suggester name** - is required for Auto-Complete suggestions.
4. **Search mode** - is used for strategy used to search
5. We can select fields from the index so that we can perform following types of queries on it

    a. Retrievable
    b. Filterable
    c. Sortable
    d. Facetable
    e. Searchable
    f. Suggester

Set values for the Index as shown in Figure 10:

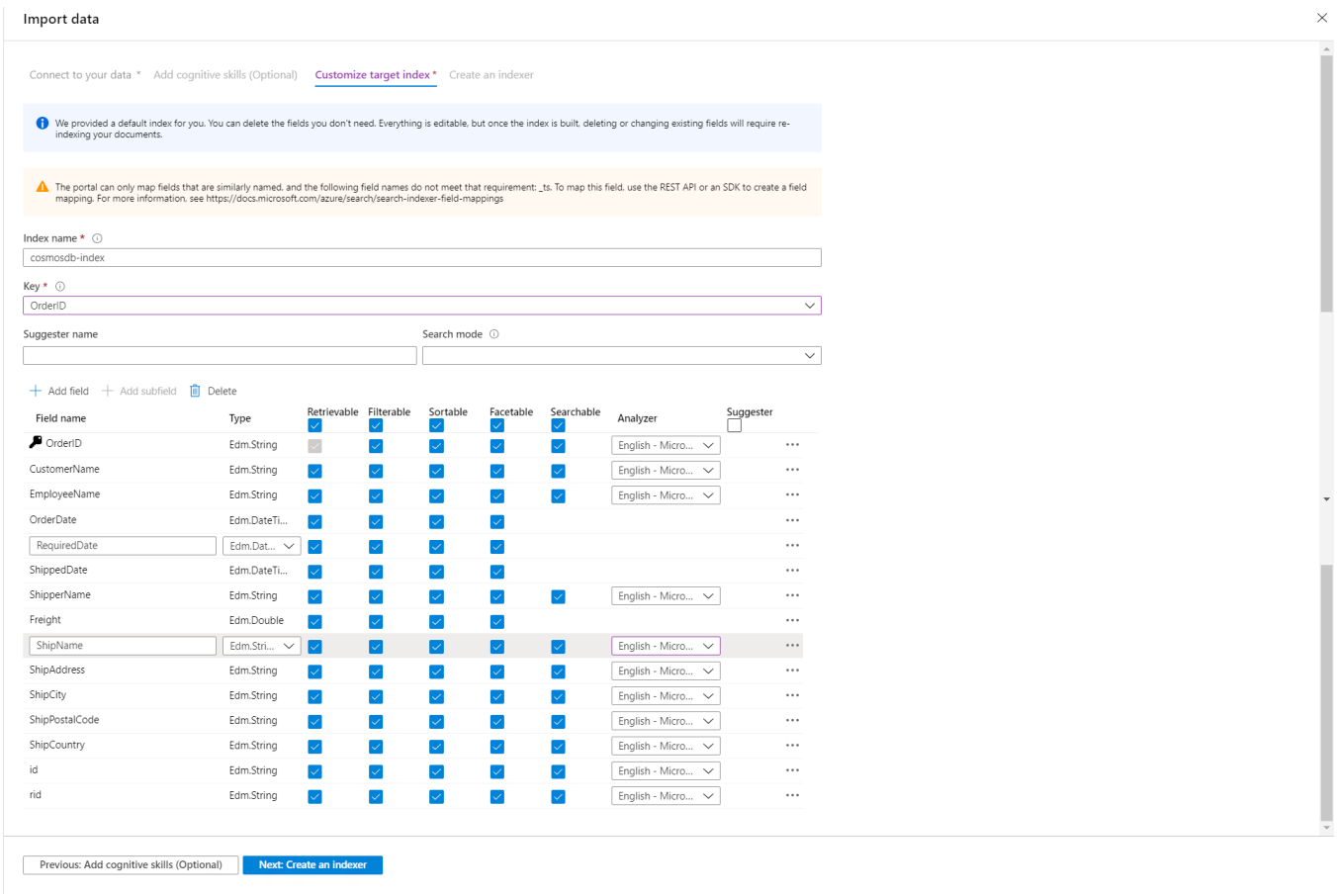


Figure 10: Index Configuration

Note that although we are making all fields as Retrievable, Filterable, Sortable, Facetable, Search can only be possible on string fields. These will be used for text-based search.

The Analyzer column provides a drop down for the query analysis syntax. Here we will select **English -Microsoft**. We are doing this for plain text-based search. There are several other Analyzer values which can be set as per your business needs.

Click on the **Next: Create an Indexer** button to create an index as shown in Figure 11.

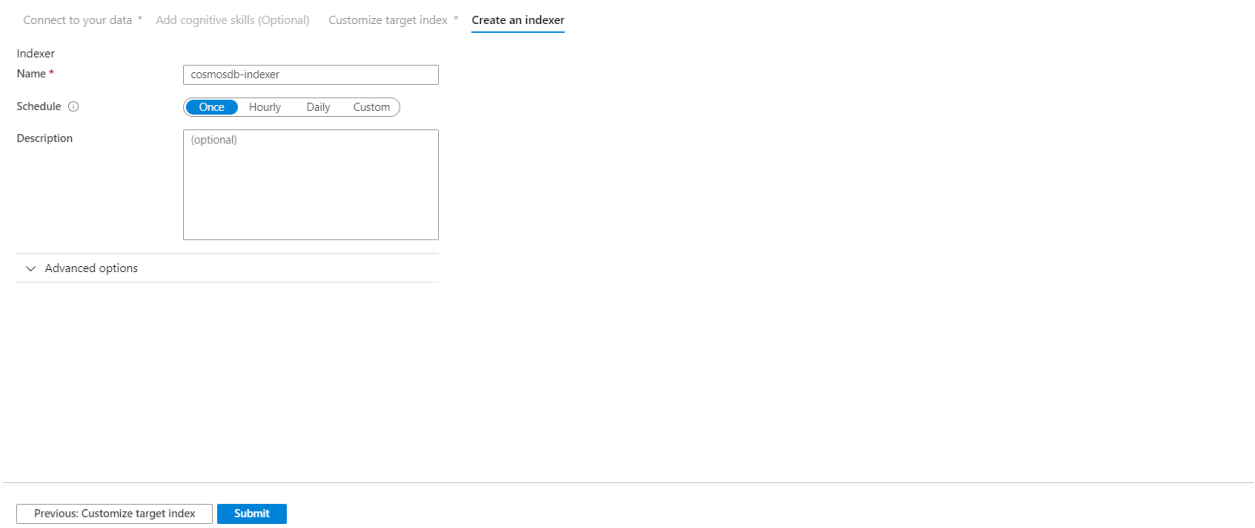


Figure 11: Create an Indexer

Here we are keeping the defaults as-is. The indexer will streamline and automate data indexes for all data source connection, data read and serialization operations.

Indexers are available for Azure CosmosDB, Azure SQL Database, Azure BLOB Storage and SQL Server on Azure VM. Click on the **Submit** button to complete the service configuration.

Once the Azure Cognitive Search Service is configured, we can view the details as shown in Figure 12.

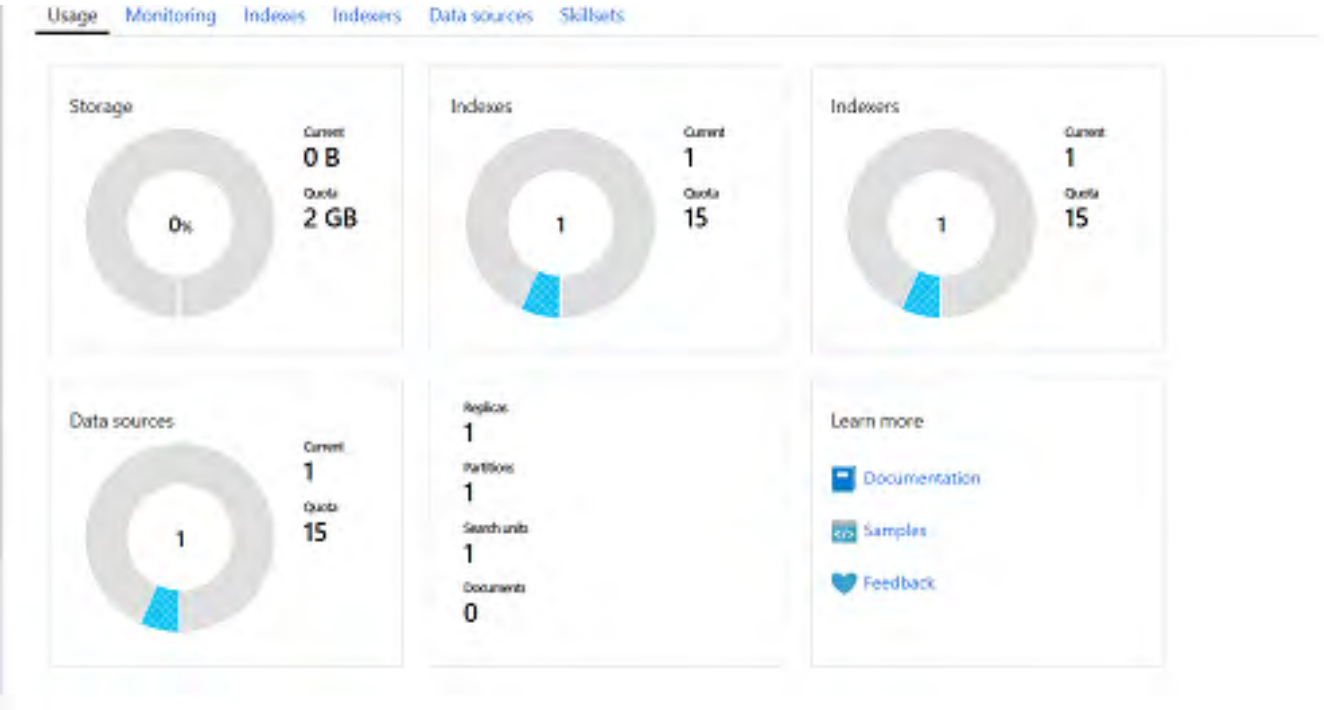


Figure 12: The Azure Service Configuration success details

As shown in Figure 12, we can see all the indexer, indexes, storage, data source, etc. for the Azure Cognitive Search service.

Click on the **Indexes** tab and then click on the **Refresh** tab, the page will display the fetched documents from the data source. See Figure 13.

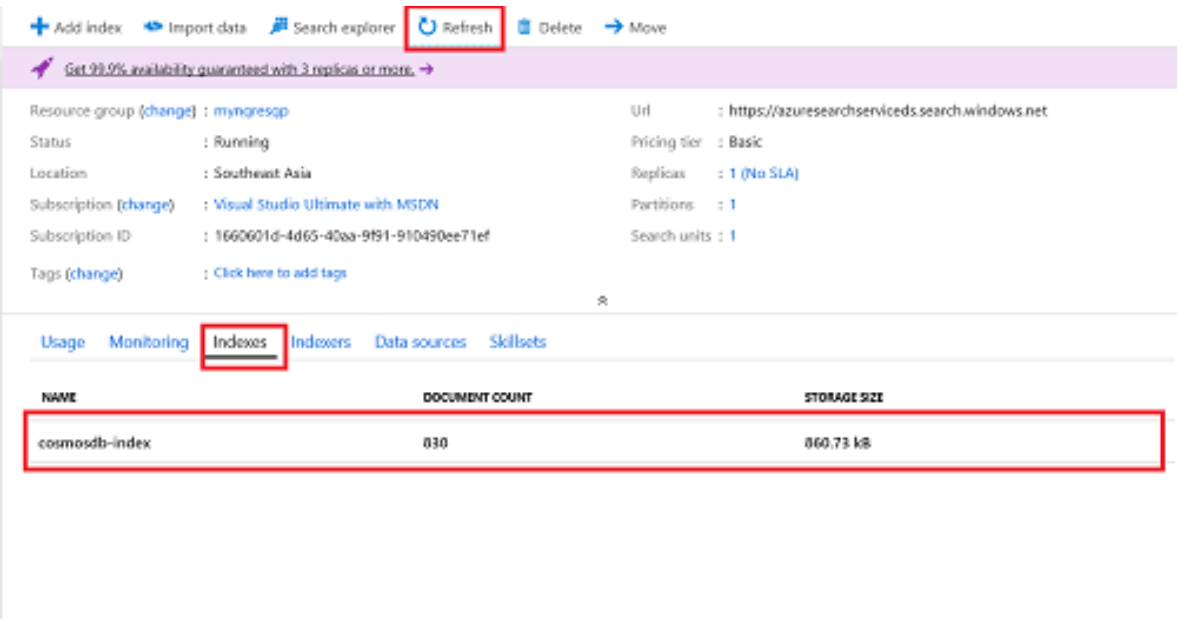


Figure 13: The Indexes with data loaded

It shows all the 830 documents (in this case) received from the data source. Now click on the **Indexers** tab and see the status. The status shows the Indexers has run successfully to load the data as seen in Figure 14.

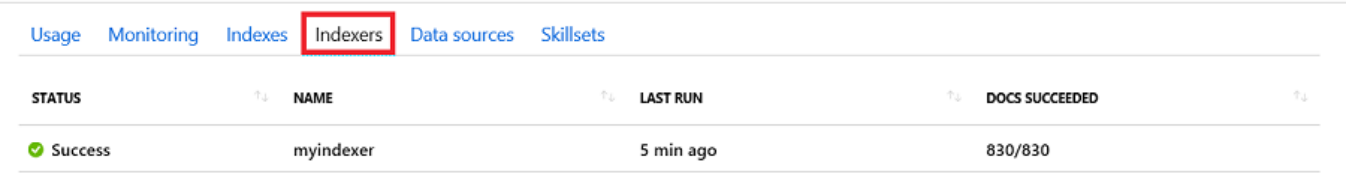


Figure 14: Indexer success

The other important details of the service e.g. URL will be displayed as shown in Figure 15.

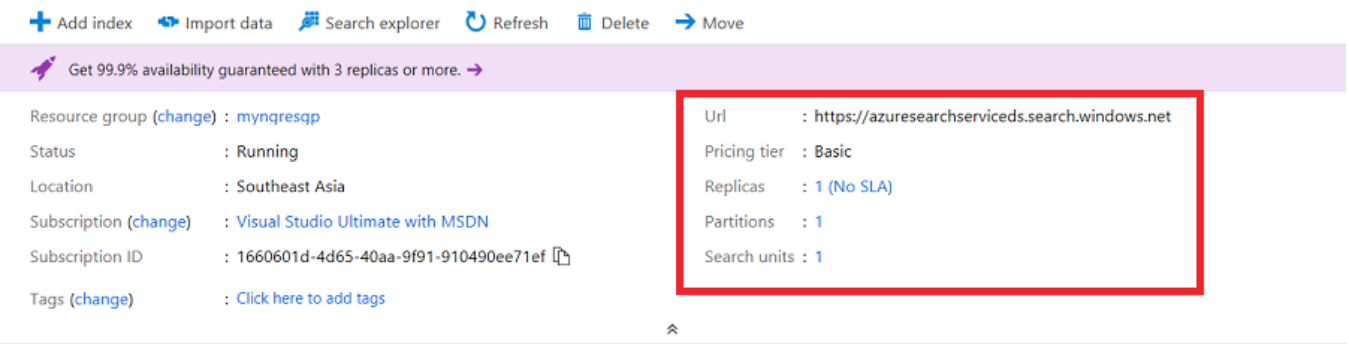


Figure 15: Azure Cognitive Search Service other details

We can query using the **Search Explorer** link. This link will navigate to the search explorer as seen in Figure 16.

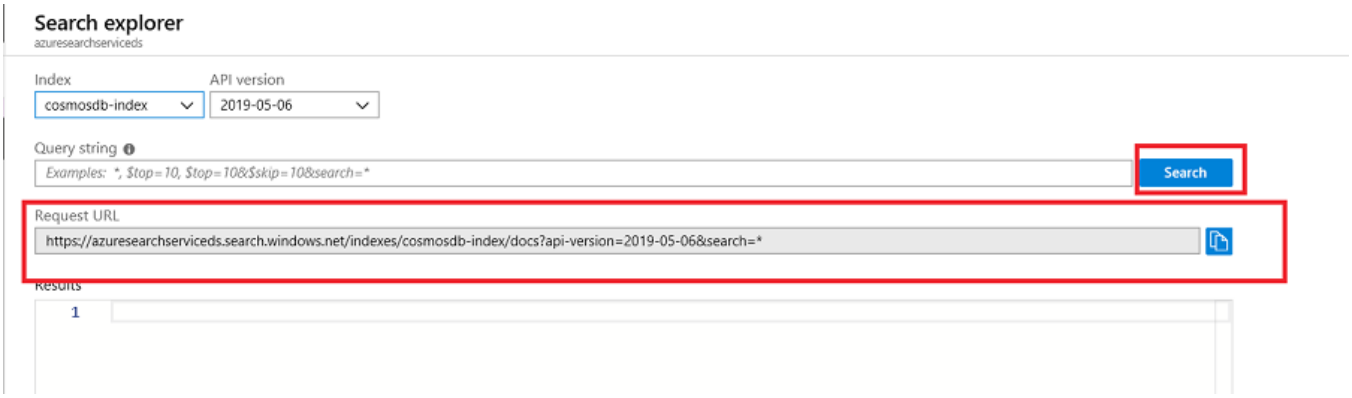


Figure 16: Search Explorer

Figure 16 shows the Index name, the API Version, Query String and Request URL. After clicking the Search button, by default, 50 records will be returned. The Request URL will contain the Query string values entered into the Query String box. In the Query String box enter '`Roland Mendel`' and click on the Search button to get all records with name `Roland Mendel`.

Azure Cognitive Search Service Limits

While creating Azure Cognitive Search Service, be aware of the maximum limits for storage, number of indexes, workloads, documents etc. These limits are helpful to provide an access of Azure Cognitive Search Service to the client application. These limits vary based on the Provisioning of Azure Cognitive Search E.g. Free, Basic, Standard, etc. More information about these maximum limits can be read from this link.

As seen in Figure 4, we have selected the Standard Provisioning for the Search Service. This means that we can create a total of 50 Indexes. To create a new container in Cosmos DB, follow all the steps discussed in the Create Data Source Sections.

From the Northwind Database and its **SuppliersData** table, execute the following query"

```
select SupplierID, CompanyName,ContactName,ContactTitle,Address,City,Country,Phone
from SuppliersData
```

As already explained in the steps for creating a data source, migrate the data from this table to the new Cosmos DB container. Import this data in the Azure Cognitive Search Service and create a new Index for reading the Suppliers Data. So now we have two indexes in Azure Cognitive Search Service.

Creating Angular Client Application to access Azure Cognitive Search Service REST API

Azure Cognitive Service provides a feature for generating a React.js application with HTML views. This can be done by clicking on "Indexes" from Search Dashboard and then clicking on "Create Search App (Preview)) as shown in Figure 17.

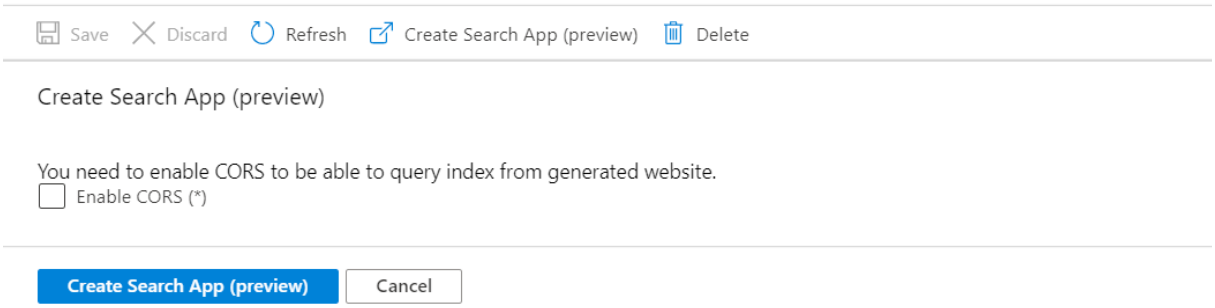


Figure 17: Sample Preview app

Since the Create Search Sapp is in preview, in this step, we will create a new Angular client application and use Azure Cognitive Search REST API to perform search operations.

We will implement a Simple Query Search in the client application. We require the Search Service Admin keys as explained in Figure 6. The admin key is used to authorize the client application against the Search Service.

Create the Angular Application using angular CLI and name this application as **azuresearch-client**.

Step 1: In the **app** sub-folder of the **src** folder, add a new file and name this file as servicedetails.ts. Add the following code to it:

```
export class ServerDetails {
  // 1. the Service name
  public static searchServiceName = 'allorderssearch';
  // 2. The Admin Key
  public static searchServiceAdminApiKey = '<MY ADMIN KEY>';
  // 3. The Index Name
  public static searchIndexName = 'cosmosdb-index-allordersdata';
  public static searchIndexNameLucene = 'cosmosdb-index-suppliersdata';
```

```
  // 4. The API Version
  public static apiVersion = '2019-05-06';
  // 5. The Search URLS
  // tslint:disable-next-line: max-line-length
  public static searchUri = `https://${ServerDetails.searchServiceName}.
search.windows.net/indexes/${ServerDetails.searchIndexName}/docs/search?api-
version=${ServerDetails.apiVersion}`;

// tslint:disable-next-line: max-line-length
public static searchUriLucene = `https://${ServerDetails.searchServiceName}.
search.windows.net/indexes/${ServerDetails.searchIndexNameLucene}/docs/search?api-
version=${ServerDetails.apiVersion}`;

}
```

Listing 1: The Azure Cognitive Search Service Details

You can see Azure Cognitive Search service details in the code. The important thing here is the URIs, as these URIs are used for performing query-based search operations.

Step 2: Add the following two code files in the **app** folder. These files will be used for *Orders* class and *SuppliersData* classes

```
export class Orders {
  constructor(public OrderID: string,
    public CustomerName: string,
    public EmployeeName: string,
    public OrderDate: string,
    public RequiredDate: string,
    public ShippedDate: string,
    public ShipperName: string,
    public Freight: number,
    public ShipName: string,
    public ShipAddress: string,
    public ShipCity: string,
    public ShipPostalCode: string,
    public ShipCountry: string,
    public id: string) {
  }
}
```

Listing 2: The Orders class

```
export class SuppliersData {
  constructor(
    public SupplierID: string,
    public CompanyName: string,
    public ContactName: string,
    public ContactTitle: string,
    public Address: string,
    public City: string,
    public Country: string,
    public Phone: string,
    public id: string

  ) {
  }
}
```

Listing 3: The SuppliersData class

Step 3: In the **app** folder, add a new file and name it as **azuresearchservice.ts**. This file contains an Angular service. This service will use the **HttpClient** class to make a call to Azure Cognitive Search Service to perform Search operations. Add the following code in the file:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { ServerDetails } from './serverdetails';

@Injectable({
  providedIn: 'root'
})
export class AzureSerchService {
  constructor(private http: HttpClient) {
  }

  searchData(query: string, pageSize: number): Observable {

    let result: Observable = null;
    const options = {
      headers: new HttpHeaders({
        'api-key': ServerDetails.searchServiceAdminApiKey,
        'Content-Type': 'application/json'
      })
    };
    result = this.http.post(ServerDetails.searchUri, JSON.stringify({
      search: query,
      top: pageSize,
    }), options);
    R1eturn result;
  }

  searchSuppliersLuceneData(query: string): Observable {

    let result: Observable = null;
    const options = {
      headers: new HttpHeaders({
        'api-key': ServerDetails.searchServiceAdminApiKey,
        'Content-Type': 'application/json'
      })
    };
    result = this.http.post(ServerDetails.searchUriLucene, JSON.stringify({
      search: query
    }), options);
    return result;
  }
}
```

Listing 4: The Service class

In the Service class, we have **searchData()** and **searchSuppliersLuceneData()** methods. These methods will be used to perform simple search and Lucene Text search respectively.

In the searchData() method, we are making a POST request by passing **api-key** and **Content-Type** in header. The api-key will authenticate the call against the Azure Cognitive Search Service. We are sending the **search** and **top** parameters in the request body. The search parameter will post the query for search operations and the top parameter will define how many results can we expect in response. The default response records are 50.

Similarly in the searchSuppliersLuceneData() method we are passing search query to perform **search** operations. You can read more about Simple Query Syntax from this link and the Lucene Query Syntax from this link.

Step 4: In the *app* folder, add a new file and name it as app.english.search.component.ts. Add the following code in it:

```
import { Component, OnInit } from '@angular/core';
import { AzureSerchService } from './azuresearchservice';
import { Orders } from './orders';

@Component({
  selector: 'app-search-component',
  templateUrl: './app.component.view.html'
})
export class AppEnglishSearchComponent implements OnInit {
  query: string;
  orders: Array;
  headers: Array;
  private order: Orders;
  recordCount: number;
  pageSizeArray: Array;
  pageSize: number;

  constructor(private serv: AzureSerchService) {
    this.query = '';
    this.orders = new Array();
    this.headers = new Array();
    this.recordCount = 0;
    // the page size array
    this.pageSizeArray = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000];
    // the defaule page size
    this.pageSize = 100;

    this.order = new Orders('', '', '', '', '', '', '',  0, '', '', '', '', '','');
  }



  ngOnInit(): void {
    // tslint:disable-next-line: forin
    for (const p in this.order) {
      this.headers.push(p);
    }
    this.onChangeQuery();

  }
  // method for search query
  onChangeQuery(): void {
    this.serv.searchData(this.query, this.pageSize).subscribe(resp => {
      const response: any = resp.value;
      this.getOrders(response);
    });
  }
  // receive the search data
  private getOrders(data: []): void {
```

```
    this.orders = new Array();
    for (const ord of data) {
      this.orders.push(ord);
    }
    this.recordCount = this.orders.length;
  }
}
```

Listing 5: The Simple Query Component class

The `Simple Query Component` class defines page size array so that the end-user can select the page size to define expected records in the response against the search query. The method `onChangeQuery()` will call the `searchData()` method from the Angular service by passing the search query and the page size parameters to it. The `getOrders()` method will update the orders array by pushing the search records in it.

Step 5: In the *app* folder, add a new html file and name it as app.component.view.html with following markup in it:

```
<div>
    <h2>The Standard Search Using 'English- Microsoft'</h2>

    <div class="form-group">
        <label>Enter Search Value</label>
        <input type="text" class="form-control" placeholder="Enter your search
here" (keyup)="onChangeQuery()" [(ngModel)]="query">
        <label>Select Page Size</label>
        <select class="form-control" (change)=" onChangeQuery()"
[(ngModel)]="pageSize">
      <option *ngFor="let size of pageSizeArray" value="{{size}}">{{size}}</option>
    </select>
    </div>
    <hr>
    <label>Total Match Found: {{recordCount}}</label>
    <hr>
    <h2>Orders Details</h2>
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td *ngFor="let h of headers">{{h}}</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let ord of orders">
                <td *ngFor="let h of headers">{{ord[h]}}</td>
            </tr>
        </tbody>
    </table>
</div>
```

Listing 6: The Html markup for the English search

The html contains the required angular binding for invoking `onChangeQuery()` method on `keyup` event

of an input element. `apgeSizeArray` is bound with select element to display page size in the option element of the select element. The table element will display received orders after search service returns a response.

Step 6: Similar to Simple Query Search, the following code shows the logic for performing Lucene Query Search in the app.suppliers.component.lucene.ts file in the *app* folder:

```
import { Component, OnInit } from '@angular/core';
import { AzureSerchService } from './azuresearchservice';
import { SuppliersData } from './suppliersdata';


@Component({
  selector: 'app-suppliers-search-lucene-component',
  templateUrl: './app.suppliers.component.lucene.view.html'
})
export class AppSuppliersSearchLuceneComponent implements OnInit {
  query: string;
  suppliers: Array;
  headers: Array;
  private supplier: SuppliersData;
  recordCount: number;


  constructor(private serv: AzureSerchService) {
    this.query = '';
    this.suppliers = new Array();
    this.headers = new Array();
    this.recordCount = 0;
    this.supplier = new SuppliersData('', '', '', '', '', '', '', '', '');
  }

  ngOnInit(): void {
    // tslint:disable-next-line: forin
    for (const p in this.supplier) {
      this.headers.push(p);
    }
    this.onChangeQuery();

  }

  onChangeQuery(): void {
    this.serv.searchSuppliersLuceneData(this.query).subscribe(resp => {
      const response: any = resp.value;
      this.getSuppliers(response);
    });
  }

  private getSuppliers(data: []): void {
    this.suppliers = new Array();
    for (const sup of data) {
```

```
      this.suppliers.push(sup);
    }
    this.recordCount = this.suppliers.length;
  }
}
```

Listing 7: app.suppliers.component.lucene.ts for accessing the the searchSuppliersLuceneData method from Angular Service

We will also have the app.suppliers.component.lucene.view.html file for Search UI for Lucene Search as shown in Listing 8.

```
<div>
    <h2>The Standard Search Using 'English- Lucene'</h2>
    <div class="form-row">
        Note that you can query for Fields as FieldName:Value <br/> You can using
AND/OR Conditions e.g FieldName1:Value1 AND FieldName2:Value2
    </div>
    <div class="form-group">
        <label>Enter Search Value</label>
        <input type="text" class="form-control" placeholder="Enter your search
here" (keyup)="onChangeQuery()" [(ngModel)]="query">

    </div>
    <hr>
    <label>Total Match Found: {{recordCount}}</label>
    <hr>
    <h2>Suppliers Details</h2>
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td *ngFor="let h of headers">{{h}}</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let sup of suppliers">
                <td *ngFor="let h of headers">{{sup[h]}}</td>
            </tr>
        </tbody>
    </table>
```

Listing 8: The app.suppliers.component.lucene.view.html file

Listing 8 contains binding with the properties and methods from the AppSuppliersSearchLuceneComponent for performing search operations based on the Lucene Query Syntax.

Step 7: In the app folder, add a new file and name it as app.main.component.ts. This file will contain routing configuration across components added in Step 5 and 6. Add the following code in the file:

```
import { Component, OnInit } from '@angular/core';
```

```
@Component({
  selector: 'app-main-component',
  template: `
   <table class="table table-bordered table-striped">
       <tr>
          <td>
            <a [routerLink]="['']">English Search</a>
          </td>
          <td>
          <a [routerLink]="['lucene']">Lucene Search</a>
        </td>
       </tr>
    </table>
    <br/>
    <router-outlet></router-outlet>
  `
})
export class MainComponent implements OnInit {
  constructor() { }

  ngOnInit(): void { }
}
```

Listing 9: The MainComponent with the Routing

Step 8: Modify app.module.ts file in the project with the following code. This code defines routing for the components we added in Step 5 and Step 6.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, Component } from '@angular/core';
import { AppEnglishSearchComponent } from './app.english.search.component';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';
import { AppSuppliersSearchLuceneComponent } from './app.suppliers.component.
lucene';
import { RouterModule } from '@angular/router';
import { MainComponent } from './app.main.component';

@NgModule({
  declarations: [
    AppEnglishSearchComponent,
     AppSuppliersSearchLuceneComponent,
     MainComponent
  ],
  imports: [
    BrowserModule, HttpClientModule, FormsModule,
    RouterModule.forRoot([
      { path: '', component: AppEnglishSearchComponent },
      { path: 'lucene', component: AppSuppliersSearchLuceneComponent }
    ])
  ],
  providers: [],
  bootstrap: [MainComponent]
})
export class AppModule { }
```

Listing 10: The AppModule

Run the application from the Command Prompt using **npm run start** command - this will run the Angular

Application. Open the browser and enter http://localhost:4200 url in the address bar. This will render a view in the browser as shown in the Figure 18.

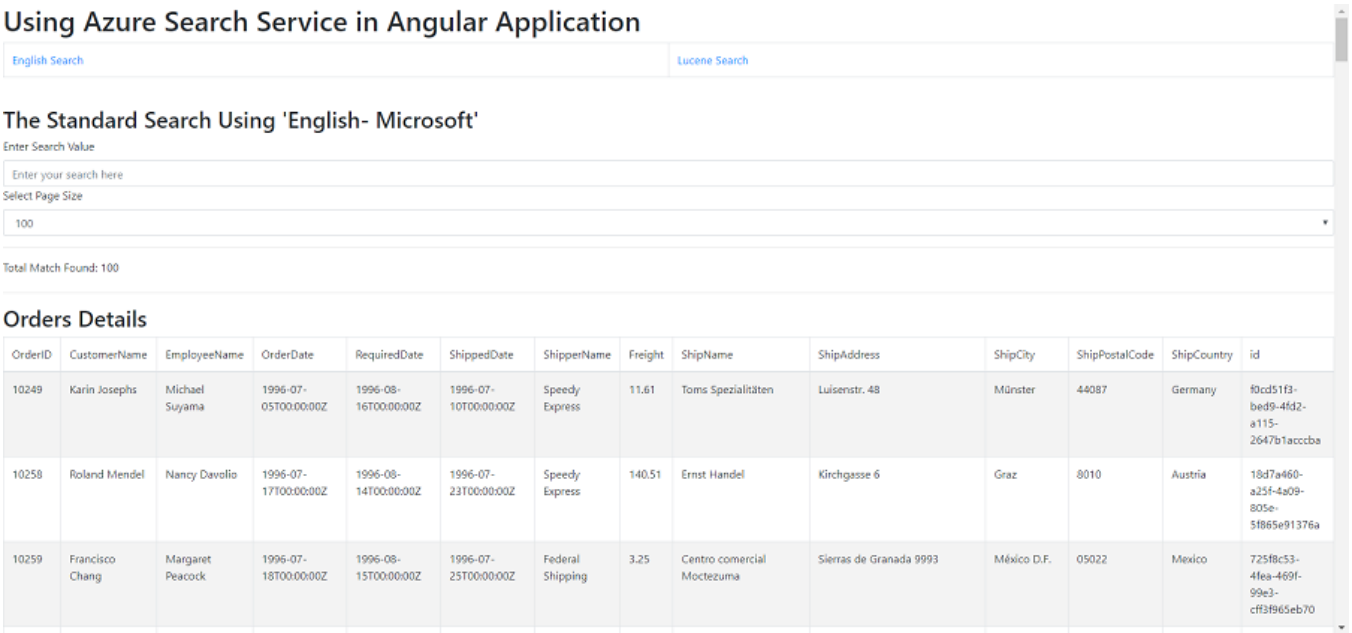


Figure 18: The Simple Query Syntax

This will load the first 100 records because the record size is 100. In the Enter Search Value, enter the value as Hari, it will immediately perform search operations. The search result will be shown as in the following figure:

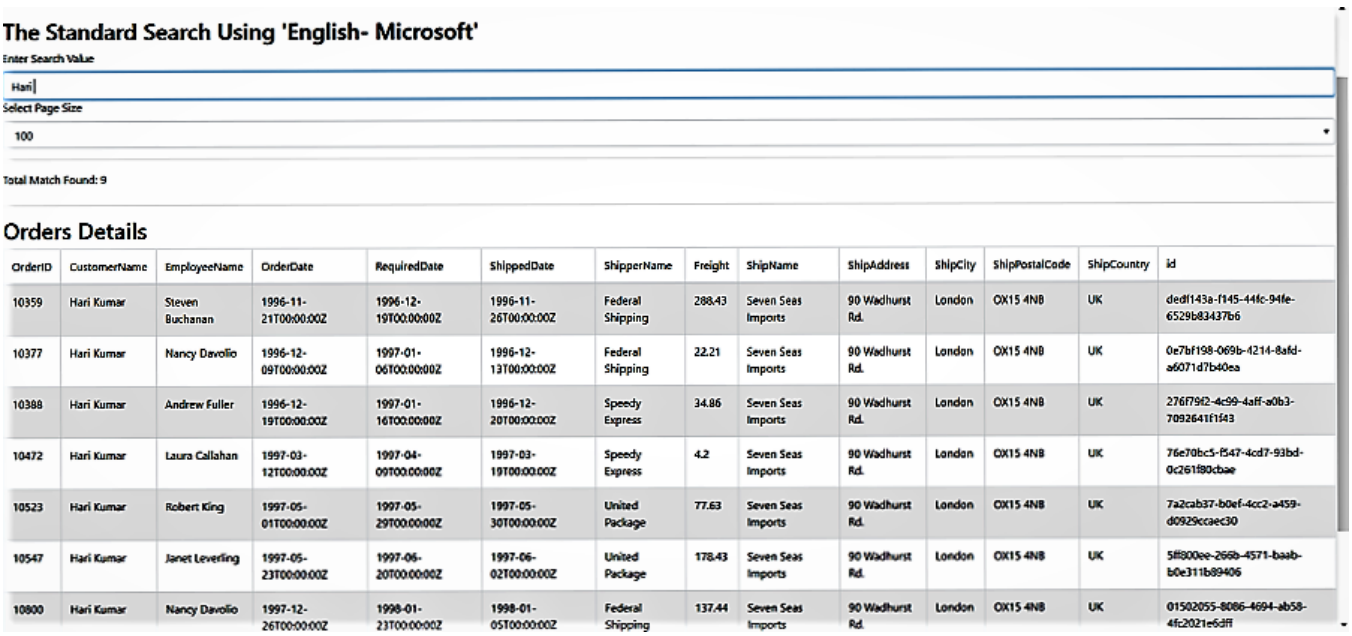
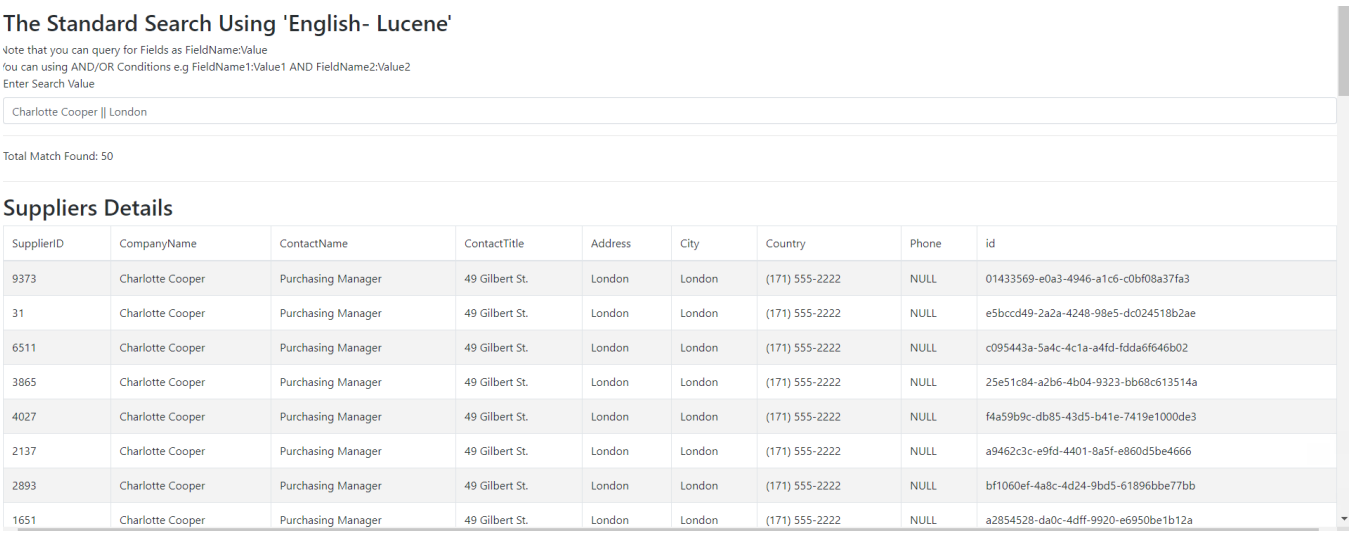


Figure 19: Simple Search based on query as Hari

We can use AND Operator for search using + operator e.g. enter “Margaret Peacock” + “Sierras de Granada 9993” in the search textbox (without the double quotes), and the search results will be displayed accordingly.

On the page, click on the Lucene Search. This page will show the Suppliers data with the first 50 records. In the Lucene Query Syntax, we can use Boolean operators, Regular Expressions, WildCards, Fuzze search,

etc. For example, in the Enter Search Value text box, enter *Charlotte Cooper || London*. A the search will take place on these values and result will be displayed as shown in Figure 20.



SupplierID	CompanyName	ContactName	ContactTitle	Address	City	Country	Phone	id
9373	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	01433569-e0a3-4946-a1c6-c0bf08a37fa3
31	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	e5bccd49-2a2a-4248-98e5-dc024518b2ae
6511	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	c095443a-5a4c-4c1a-a4fd-fdda6f646b02
3865	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	25e51c84-a2b6-4b04-9323-bb68c613514a
4027	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	f4a59b9c-db85-43d5-b41e-7419e1000de3
2137	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	a9462c3c-e9fd-4401-8a5f-e860d5be4666
2893	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	bf1060ef-4a8c-4d24-9bd5-61896bbe77bb
1651	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London	London	(171) 555-2222	NULL	a2854528-da0c-4dff-9920-e6950be1b12a

Figure 20: The Lucene Query syntax

In the Search Text box, enter Admin*. A search occurs for the word starting with Admin and it will search Admin, Administrator, etc. This makes it easy to perform search.

Conclusion

Azure Cognitive Search is a search-as-a-service solution with optimized and high-performance search capabilities. It allows developers to incorporate great search experiences into applications that needs enterprise search across various data sources.

• • • • • • •

Download the entire source code from GitHub at
bit.ly/dncm46-azure-search

Mahesh Sabnis

Author

Mahesh Sabnis is a DotNetCurry author and ex-Microsoft MVP having over 19 years of experience in IT education and development. He is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET, Cloud and JavaScript Technologies (all versions). Follow him on twitter @maheshdotnet

*Thanks to **Vikram Pendse** for reviewing this article.*

Yacoub Massad

CODING PRACTICES:

THE MOST IMPORTANT ONES – PART 1

In this tutorial, I will talk about the most important coding practices based on my experience.

INTRODUCTION

In this tutorial, I will talk about the coding practices that I found to be the most beneficial in my experience.

The most important practice is **Automated Testing**. More specifically, the practice of making sure that simple tests are written that allow you to verify that you don't break anything when you modify your code.

In this part, I will talk about this.

Note: *In this article, I give you advice based on my 9+ years of experience working with applications and sharing my knowledge. Although I have worked with many kinds of applications, there are probably kinds that I did not work with. Software development is more of an art than a science. Use the advice I give you if it makes sense in your case.*

Practice: Having tests that pin program behavior

Make sure that you have tests that pin the behavior of your programs. Here are the properties of the tests I am talking about:

1. **Require low maintenance:** A good test will not need to be modified often. A very good test will almost never be modified once it is written.

Before talking about writing low maintenance tests, let's talk about visibility layers.

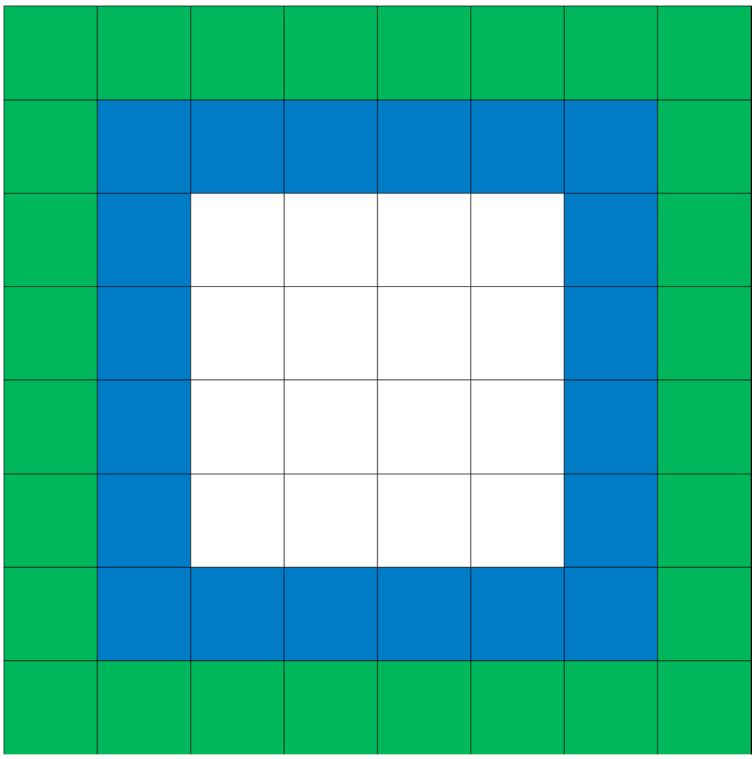


Figure 1: Visibility layers in some program

Figure 1 illustrates the visibility layers of some program. I use the word "layer" here in a very general sense.

I am not trying to talk about a specific architecture of a program. But in general, all programs have some sort of visibility layers. That is, there are parts of the program that are more visible to the outside world, and others that are internal and less visible from the outside.

The green squares in Figure 1 represent the outer visibility layer of the program – i.e. the most visible parts of the program. This may include the user interface, the database, the public API of the program (e.g. when the program is a web service), etc. The blue squares represent the code at a more internal layer. The white squares represent code at an even more internal level.

For example, the blue squares might contain code that is immediately below the UI, e.g. View Models in an MVVM application, or code that usually sits underneath the UI. These squares could also contain data access code that is just above the database itself, e.g. an ORM.

When writing tests, we can choose to use the green blocks as the points of interaction between the tests, and the system under test.

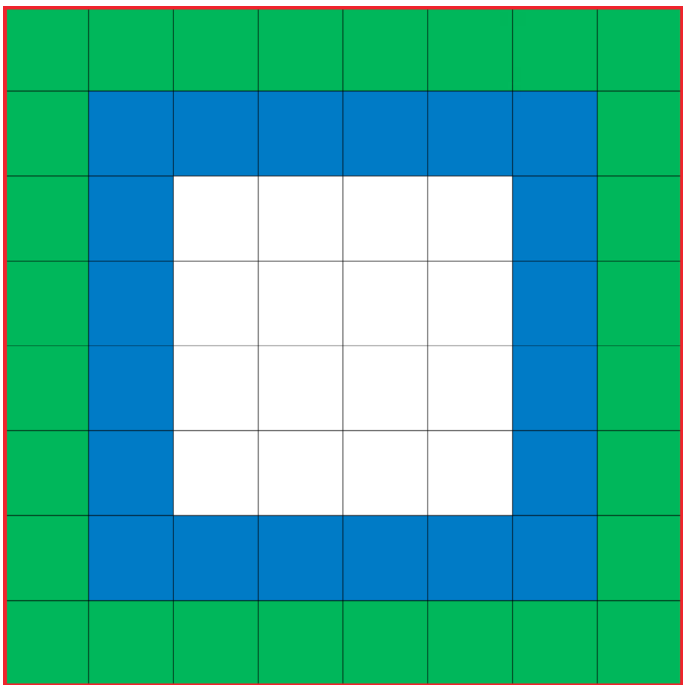


Figure 2: Using the outer visibility layer for testing

For example, we can write a test that interacts directly with the user interface (UI tests), and then checks that certain data has been put inside the database. As a more concrete example, we can use a UI testing library such as the Microsoft's Coded UI component to interact directly with the buttons and other input fields of some UI.

Also, in the assertion phase of our test, we can directly access data in a Microsoft SQL Server table to see that data was inserted as expected by the test. Alternatively, the assertion phase can also be done against the UI. For example, we can navigate to some reporting section of the UI and verify that the data that was given as input to the UI, is now part of some report that is visible in the UI.

We can also choose a more internal *visibility layer* to do that. See Figure 3 with an area surrounded in red.

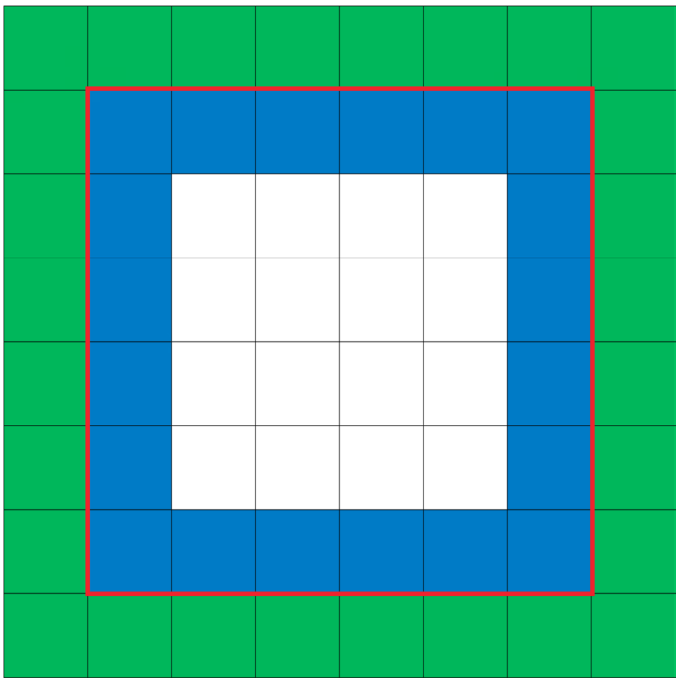


Figure 3: Using the second most visible layer for testing

For example, instead of writing a test that interacts with the user interface, we can write a test that interacts with the View Models in some MVVM application. Also, instead of checking data in the database server at the assertion phase, we can use an in-memory database in the test. Or, if we are using the repository pattern, we can use a fake repository object for testing.

We can also test internal code. We can test a single unit of behavior, or a group of units.

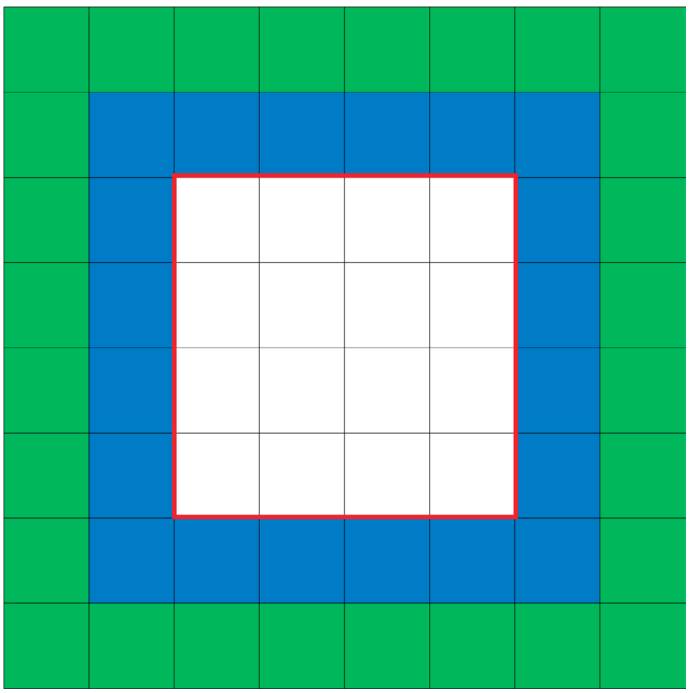


Figure 4: Testing a group of internal units of behavior

An example of such a test is a test that tests a few classes together. Figure 4 depicts this.

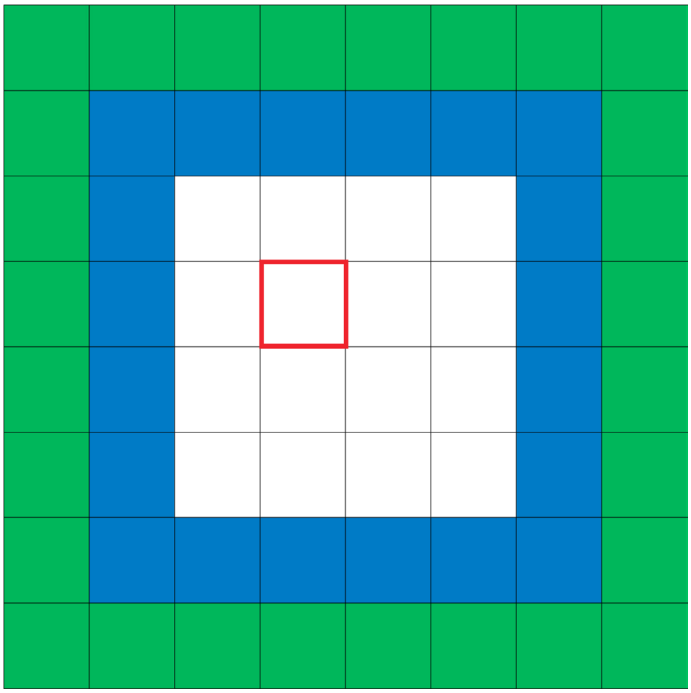


Figure 5: Testing a single unit of internal behavior

Figure 5 depicts testing a single unit of internal behavior, e.g., testing a single class. The boundary of the tests can be anything really.

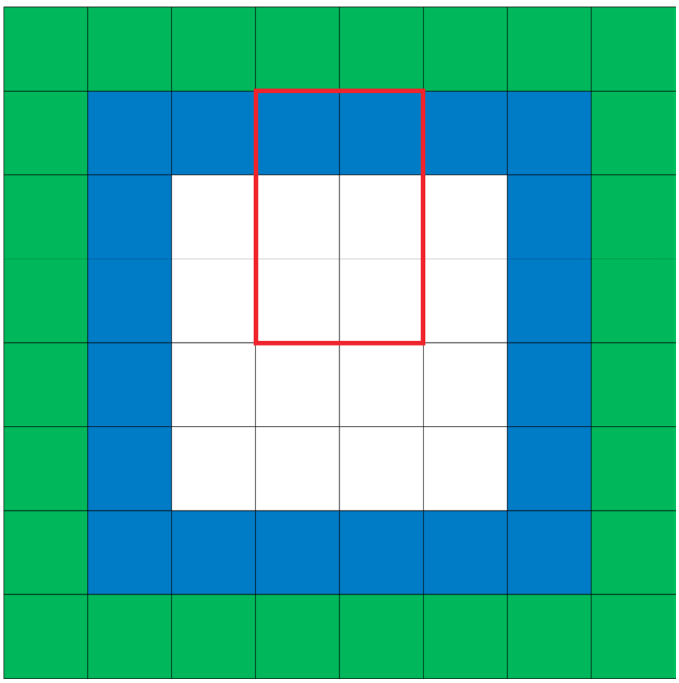


Figure 6: A test that spans different visibility layers

For example, we can invoke the program using the View Models of some MVVM application, and then assert against the internal state of some internal class in the program. **I am not saying that you should do this**, I am just exploring the possibilities.

Different tests have different advantages and disadvantages.

Although many kinds of tests can be important, here I only want to talk about the most important ones. That is, I want to talk about the ones that I consider doing them as the **number one best practice**.

For tests to require low maintenance, they should be written using points of interaction that are less likely to change. The following can change easily:

- **Internal code:** The internal blocks of our programs are likely to change. One day we may decide to implement a feature in a certain way. The next day we decide to do it another way for whatever reason. We might delete existing functions/classes and add new ones. Or we might change the signatures of these functions/classes. Such changes immediately break tests that are done against such functions/classes.

- **Volatile outer layer code:** The outer visibility layers of a program are closer to the user requirements. For example, the UI is what the user sees. The user cares more about the user interface and the functionality it provides than the internal implementation details behind it. The public API of a web service is what the consumer of the service sees. The consumer of the service cares about this API and doesn't really care about the internal implementation details.

Although such outer visibility layers are closer to user requirements, some kinds of components in such layers are still volatile and can change. The user interface is one example. Although the core features of the UI are relatively stable, the UI itself is volatile.

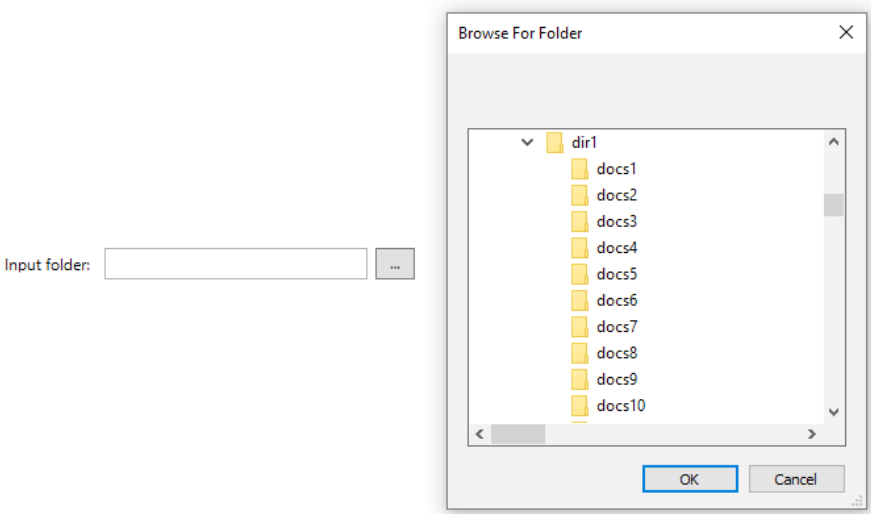
To explain this further, consider a program that allows users to translate some documents in a folder. The user is expected to give the following inputs to the program:

i. The input folder path
ii. Authentication settings for the translation server
iii. Output document format

The core UI feature here is to input these three pieces of data. However, you can implement this via different kinds of UI features, e.g.:

i. You can have all these fields in a single form, or you can have a wizard where each page of the wizard asks for only one of these inputs.

ii. You can use a single text box and a pick button that opens a dialog that allows you to pick the folder from the folder tree of the file system like Figure 7, or you can have the file system folder tree available directly on the form like Figure 8.



<= Figure 7: Picking the input folder by clicking a pick button first

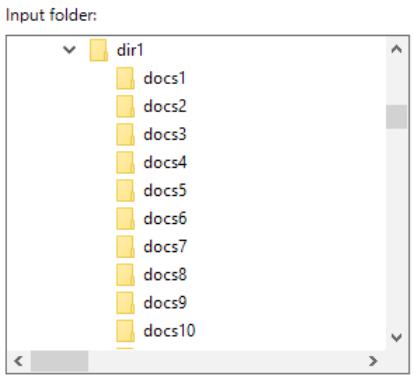


Figure 8 Picking the input folder directly

iii. You can use a drop-down list (e.g. via a ComboBox) to let the users select the output document format or you can use radio buttons.

As you can see, although the core UI functionality is to input three pieces of information, the UI itself can be anything, and the exact way we implement it can easily change.

There are other components that are volatile. For example, if your program communicates with an external web service to calculate exchange rates, the web service might be down, and you cannot control the data returned by such web service.

So, to make tests require low maintenance, we should write them against non-volatile components in the most possible outer visibility layer. For programs with a UI, a subcutaneous test might be what you need. For web services, the service API is usually the best places to use for tests.

2. **Be deterministic.** A failing test should always mean there is a problem. The test shouldn't fail because an external web service is down. It shouldn't fail because the currency exchange rates (which are returned by the external service) have changed. Fakes can usually be used to fix such issues. Note that some dependencies might be acceptable.

   For example, a local database (a special testing database controlled by the tests) might be acceptable. The price to pay in terms of indeterminism (that the local database might be down) in this case may (or may not) be relatively low compared to the value of testing against a real database.

3. **Be close to user requirements.** This is very much related to Pratice #1. Because user requirements change slower than implementation details and because they are mostly incremental in nature, having tests at the level of user requirements makes them less likely to change.

4. **Run relatively quickly:** The tests I am talking about here, run quickly.

   How quickly?

   Enough to be able to run the possibly hundreds or even more tests in a few minutes. Although it is best for a test to take less than a few milliseconds, it might be fine if a test takes a second or a few seconds. There is a lot of advice out there that tests should run even quicker than this. My problem with super-fast tests is that they usually test small internal blocks of code that are implementation details which would then make the tests require high maintenance.

   If you can have super-fast end-to-end tests, that would be great. If not, making the tests low-maintenance is more desirable. Of course, there might be cases were end-to-end tests are super-slow. In

that case, using a lower visibility layer might be a good option. The important thing is to realize the cost and value of testing at each layer and then deciding on the best option.

> *One advantage of super-fast tests is that you can run them frequently, even as you are typing code. Although this has value, the value of having low-maintenance end-to-end tests is higher in many cases. I usually run tests a few times every day, and that is enough for me.*

Another thing that might be good to do when creating a test suite is to add an additional layer for testing. That is, the tests themselves will not invoke the system under test directly, instead they will invoke it indirectly.

For example, if you are testing a program which is a translation web service, you can have a method in your test class called `InvokeSUT` that the tests call. `InvokeSUT` would in turn invoke the translation web service that is under test. This way, if there are minor changes to the public interface of the web service under test, only the `InvokeSUT` method will need to be changed instead of all the test methods.

Conclusion:

This article is about the coding practices that I found to be the most beneficial during my work in software development. In this part, Part 1, I talked about the most important practice: **having low-maintenance** relatively-quick-to-run end-to-end tests that pin the program behavior.

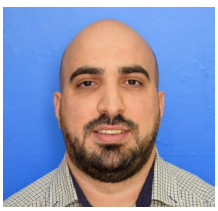
I will talk about some more practices in the upcoming articles.

• • • • • • •

Yacoub Massad

Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com.



*Thanks to **Damir Arh** for reviewing this article.*

Thank You

for the 46th Edition

@dani_djg

benjamij

@subodh_sohoni

@sravi_kiran

@yacoubmassad

@maheshdotnet

@damirarh

@suprotimagarwal

@vikrampendse

@gouri_sohoni

@saffronstroke

Write for us - mailto: *suprotimagarwal@dotnetcurry.com*