Building a Cloud Enabled Search Appliance

Jeffrey T. Fritz
7 Aug 2019 CPOL

Azure functions are an extremely versatile platform to operate on. The dtSearch tools allow us to build and manage a search index anywhere that we need intelligent search capabilities.

When you think of all of the documents that you have on your PC, your Mac or your phone you know that there are a lot of documents laying around that you either need to keep meticulously organized or be able to search for quickly in order to find the documents you need quickly and easily. I know that my OneDrive and iCloud drives are littered with photos, PDFs and Microsoft Office documents containing all sorts of materials and if I don't remember the hierarchy that I stored them in, I'll never find anything.

What if there was an easier way to store my documents in the cloud for backup purposes and get the same great searching capabilities as when I have those documents stored locally? That's where dtSearch and Azure functions can together provide a neat experience for managing your documents and their contents in the cloud.

Why Azure Functions and Storage

Azure Storage is a great facility that keeps multiple backups of your documents and data across regions. You may choose to store your data safely in the US East coast data center, but it's also in two other data centers in locations like Northern Europe and Brazil.

Azure Functions allow us to write a little bit of code that can be managed and run when appropriate to interact with those storage locations and perform services for us. This keeps us, as application authors and maintainers, out of the business of managing systems and run-times of our applications. We simply expect our functions to run when they are triggered to run. Nothing more, nothing less.

In the case of dtSearch and our data, this is a perfect match. We can structure our storage location so that it properly houses the archive of documents we want to process with our search engine, and we can also trigger the regeneration of that search index appropriately when a new document is added to the collection. The Azure Function architecture also allows us to make querying that managed search index a trivial operation with an HTTP endpoint, accessible from anywhere in the world.

Getting Started with Azure Functions

Azure Functions are a simple, low profile, consumption-based way to interact with our data. Their 'serverless' architecture means that we can focus on just writing an effective function to process requests and data. For this search appliance, I started by creating an Azure Functions application v2.0 in Visual Studio 2019. These projects are written using .NET Core and can also reference .NET Standard libraries. I chose to start with a Blob trigger configuration when walking through the initial configuration screens in Visual Studio.

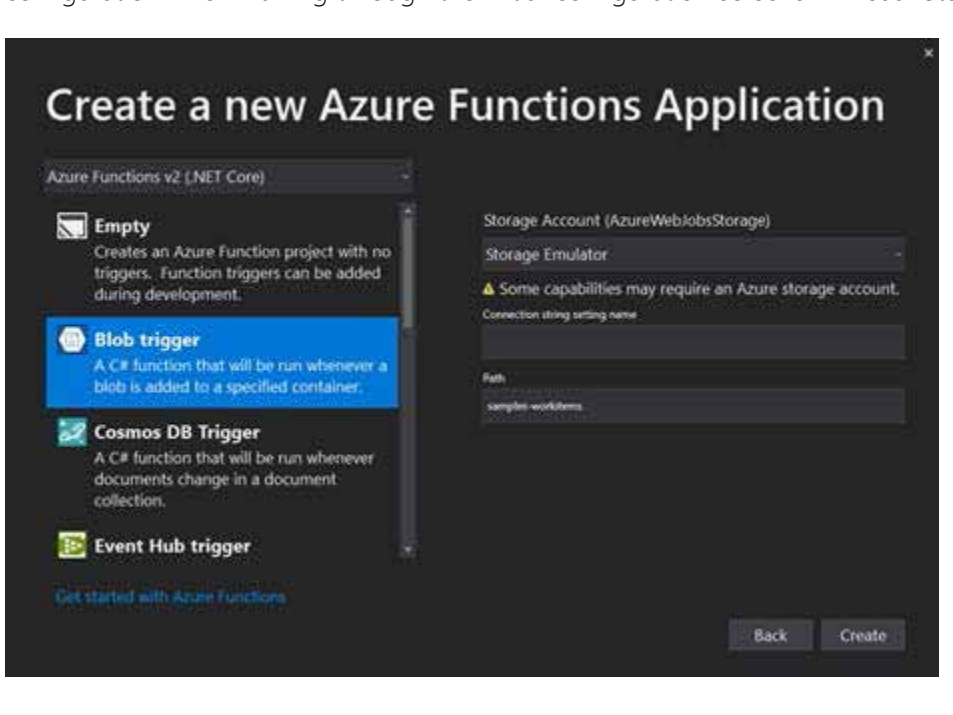

Figure 1 Configuring a New Azure Functions Project with a Blob Trigger

I left the other settings on the right side with the defaults, and created the project. By selecting 'Blob trigger', we can activate a method when a new file is loaded into Azure Blob storage. See where this is going? Let's write a simple method to take new documents from blob storage and prepare them to be added to the dtSearch Engine.

Hide Shrink ▲ Copy Code

```
[FunctionName("AddDoc")]
public static void AddDocument(
        [BlobTrigger("docstoindex/{name}", Connection = "AzureWebJobsStorage")]Stream
myBlob,
        string name, ILogger log)
{

        var fileNameParts = name.Split('.');
        var newFileName = string.Concat(fileNameParts[0], "_",
DateTime.UtcNow.ToString("yyyyMMddHHmmss"), ".", fileNameParts[1]);
        var targetStream = new FileStream(Path.Combine(DocsFolder, newFileName),
FileMode.Create);
        using (var sw = new BinaryWriter(targetStream))
        {
                var sr = new BinaryReader(myBlob);
                while (myBlob.Position != myBlob.Length)
                {
                        sw.Write(sr.ReadByte());
                }
                sw.Flush();
        }
        targetStream.Close();
        targetStream.Dispose();

        log.LogInformation($"C# Blob trigger function Processed blob\n Name:{name} \n
Size: {myBlob.Length} Bytes");

}
```

Let's break this down. The `FunctionName` attribute on this method defines a name that Azure will use to reference this method. The BlobTrigger hint inside the parameter list indicates that the arrival of a blob at the folder location "docstoindex" defined in the connection named '`AzureWebJobsStorage`' will start this method and the contents of that blob are made available in the Stream called '`myBlob`'. The filename is provided in the input parameter called 'name' and a logger for Azure functions is provided as well.

This method will identify the filename and add a datestamp before the file extension. Then, it will copy the file to an Azure File storage location identified by the `DocsFolder` property. The `DocsFolder` property is identified and managed with this syntax:

Hide Copy Code

```
private static string GetFolder(string folderName)
{

        var baseFolder =
Environment.ExpandEnvironmentVariables(@"%HOME%\data\dtSearch");
        var fullPath = Path.Combine(baseFolder, folderName);
        if (!Directory.Exists(fullPath)) Directory.CreateDirectory(fullPath);

        return fullPath;

}

private static string DocsFolder { get {

        if (string.IsNullOrEmpty(_DocsFolder)) _DocsFolder = GetFolder("Docs");

        return _DocsFolder;
} }
```

When you deploy an Azure Function application, an Azure Storage account is mounted and you can work with the contents in the ***%HOME%\data*** folder. In this case, we've created a folder for dtSearch and provisioned a Docs folder underneath that to hold the documents we will index.

Building the Index, and Running Native Code on Azure

With documents uploaded, how do we build a dtSearch index? The dtSearch Engine is written in native code, and has a .NET Standard library available. We can configure Azure to run our functions on a Windows service, and we can reference the dtSearch libraries appropriate for the platform with some hand-edited entries in our csproj file. For our project, we've copied the dtSearch Engine SDK folders for .NET Standard and placed their contents in the lib folder next to our projects.

Hide Copy Code

```
<ItemGroup>
<Reference Include="dtSearchNetStdApi">
<HintPath>..\lib\engine\netstd\dtSearchNetStdApi.dll</HintPath>
</Reference>
</ItemGroup>
<ItemGroup Condition="'$(OS)' == 'Windows_NT'">
        <Content Include="..\lib\engine\win\x86\dtSearchEngine.dll">
                <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
        </Content>
</ItemGroup>
```

At the time of this writing, the Azure Functions runtime on Windows servers would only load and work with 32-bit native DLLs. The Azure team doesn't guarantee support for native libraries at all, but in our case (spoiler alert), the ***dtSearchEngine.dll*** file loads and runs properly. These two ItemGroup elements allow our .NET Core code to reference the .NET Standard wrapper library for dtSearch that in-turn references the native DLL that is conditionally included in the second ItemGroup. If we were also deploying to Mac or Linux, there would be additional elements with conditional references to native libraries for those operating systems as well.

With the search engine properly referenced, we can add a function to our Azure Functions project that will analyze the contents of the `DocsFolder` referenced in our previous code sample, and build an `IndexFolder` that we can serve search results from.

Hide Copy Code

```
[FunctionName("BuildIndex")]
public static async Task BuildIndex(
[QueueTrigger("buildindex", Connection = "AzureWebJobsStorage" )]string queueItem,
ILogger log)
{

        var indexPath = GetFolder("Index");
        log.LogInformation("Building index at " + indexPath);
        log.LogInformation("Building index with documents from " + DocsFolder);

        using (var job = new IndexJob())
        {
                job.IndexPath = indexPath;
                job.ActionCreate = true;
                job.ActionAdd = true;
                job.FoldersToIndex.Add(DocsFolder);
                job.Execute();
        }

        log.LogInformation("Completed Indexing");
}
```

This time, we're going to trigger this "BuildIndex" function based on the arrival of an entry in the queue called "buildindex". Once an entry arrives, we ignore the content of the message from the queue and start a dtSearch IndexJob that writes the contents into the index folder. The dtSearch Engine supports building an index and searching against the index concurrently, so we don't need to worry about locks or managing state of the index folder. Separating the upload of documents from the construction of the index means that we can upload many documents and build the index once the final document is uploaded.

Now that we have an index, how do we search it? Azure Functions can also be triggered as an HTTP endpoint. You can browse to the function and trigger its execution, or you can use an HttpClient to fetch data from the function's location. We'll build the search function as follows:

Hide Shrink ▲ Copy Code

```
[FunctionName("Search")]
public static IActionResult Search([HttpTrigger] HttpRequest request, ILogger log)
{

        if (request.Query["t"].Count == 0) return new NotFoundResult();

        var queryTerm = request.Query["t"].ToString();

        var results = new SearchResults();
        using (var job = new SearchJob())
        {

                job.Request = queryTerm;
                job.MaxFilesToRetrieve = 10;
                job.IndexesToSearch.Add(IndexFolder);
                job.SearchFlags = SearchFlags.dtsSearchDelayDocInfo;
                job.Execute(results);

        }

        log.LogInformation($"Searching for '{queryTerm}' and total hits found: " +
results.TotalHitCount);
        if (results.TotalHitCount == 0) return new NotFoundResult();

        return new ContentResult { Content = results.SerializeAsXml(), ContentType =
@"text/xml", StatusCode = 200 };

}
```

This `Search` method is triggered with the HttpTrigger hint, specifically looking for a querystring argument "t". We can use requests formatted like this "https://dtsearchsample.azurewebsites.net/api/SearchIndex?t=OCR" to search for the term OCR in our index. We can also add "&code=<security code>" to the end of that URL if we secure the function with an Azure Function Key.

A standard dtSearch `SearchJob` is executed, looking for a maximum of 10 files to return, and searching in the `IndexFolder` we just defined in the previous function. The job is executed, and the results are returned in XML format.

```
-<dtSearchResults>
   <Request>ocr</Request>
   <BooleanConditions/>
   <FileConditions/>
   <TotalFiles>1</TotalFiles>
   <TotalHits>16</TotalHits>
   <SearchFlags>4096</SearchFlags>
   <SearchDate>2019-07-22 21:49:08Z</SearchDate>
   <dtSearchEngineVersion>8596</dtSearchEngineVersion>
 -<Item>
     <Location>D:\home\data\dtSearch\Docs</Location>
   -<Title>
       Updating 1980's Technology with the Help of 2017's Imaging Tools I live in an ol
     </Title>
     <IndexRetrievedFrom>D:\home\data\dtSearch\Index_1</IndexRetrievedFrom>
     <HitCount>16</HitCount>
     <OriginalHitCount>16</OriginalHitCount>
     <Score>16</Score>
     <ScorePercent>100</ScorePercent>
   -<Hits>
       2156 2008 1917 1867 1736 1678 1561 1490 1394 1375 1276 1263 592 491 486 378
     </Hits>
     <Size>220593</Size>
     <TypeID>268</TypeID>
     <ParserID>0</ParserID>
     <Filetype>Word 2007</Filetype>
     <WordCount>2375</WordCount>
     <Date>2019-07-17 22:09:23Z</Date>
     <CreateDate>2019-07-17 22:09:23Z</CreateDate>
     <DocId>1</DocId>
     <IIndex>0</IIndex>
     <IndexedBy>8596</IndexedBy>
     <Ordinal>0</Ordinal>
     <FieldFlags>0</FieldFlags>
     <UserFields/>
   </Item>
</dtSearchResults>
```

Figure 2 Search Results from the dtSearch engine in XML format

Configuring and Deploying to Azure

Deploying our function to Azure is a multi-step process that can be easily accomplished through the Azure portal. First, search for and create a "Function App". We have filled out the required configuration with the following details:

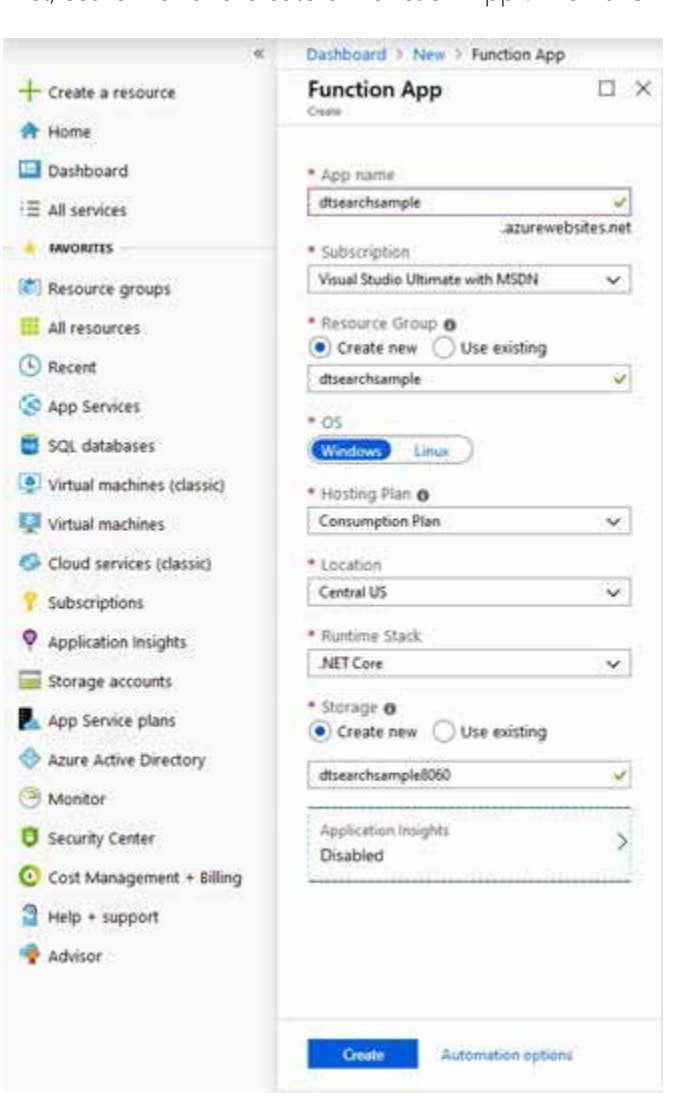

Figure 3 Create an Azure Function App

For this project, we need to be sure that we create a Windows OS hosted function app. If we wanted to deploy to Linux, we would need to include the Linux native dtSearchEngine.so library. You'll also notice the new storage plan created to host the files used by the Azure functions. This is where our search index and our indexed files will reside.

Click 'Create' at the bottom and in a few short minutes the Azure function application will be created. Once created, we need to double check the bitness of the application in the 'Configuration' settings. Click through to the 'General Settings' view and verify the platform is set to 32 Bit.

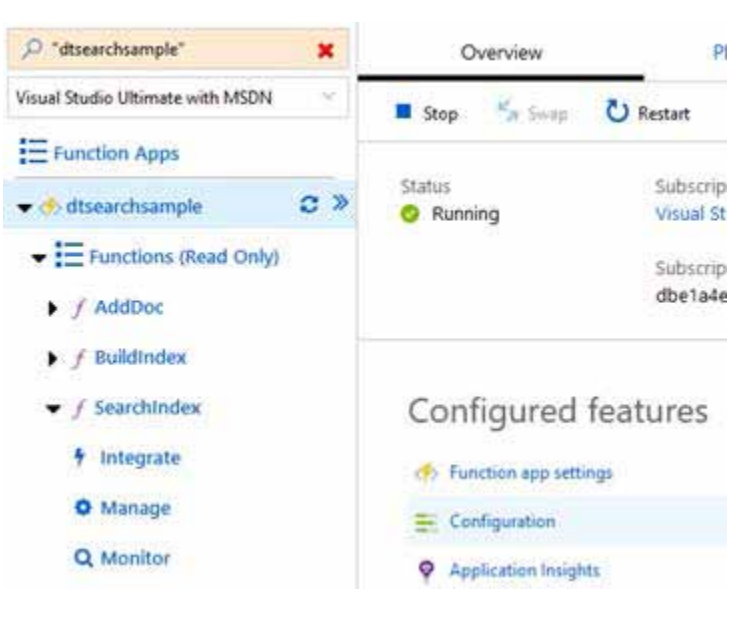

Figure 4 Azure Function Configuration

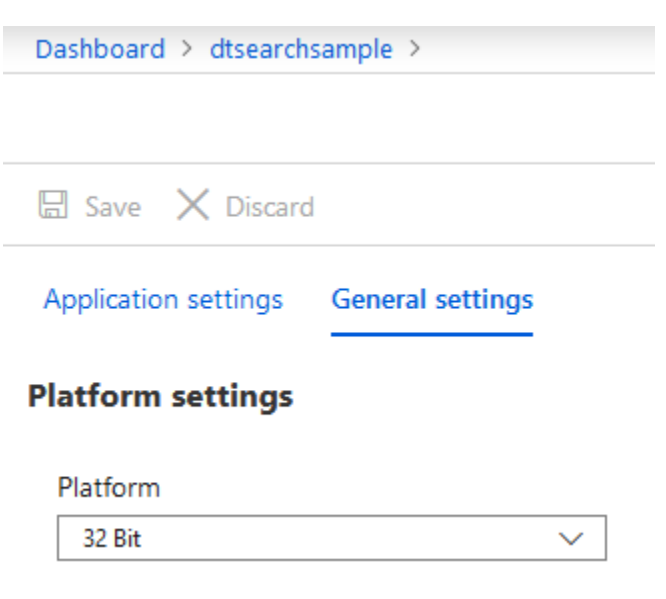

Figure 5 Platform Bitness Setting

Next, click the 'Get Publish Profile' link on the main view of the Function app in the Azure portal. We can then start publishing the application from Visual Studio by clicking the 'Build-Publish…' main menu item.

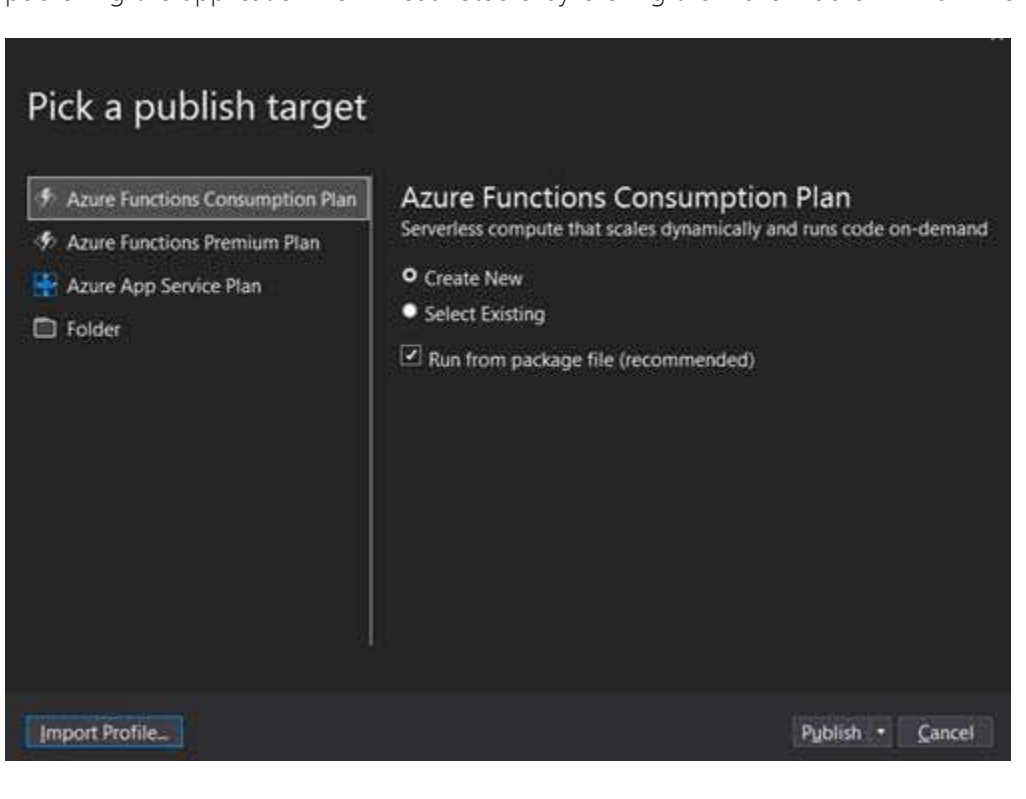

Figure 6 Initial Publish an Azure Function Dialog from Visual Studio

Ignore the text in the middle of the screen, and click the 'Import Profile' button in the bottom-left corner to load the profile configuration you downloaded in the previous step. This will give Visual Studio everything it needs in order to upload your project to Azure. Click through a Finish button and your application will happily be living in the cloud in just a few moments.

With the functions running, we can add documents to be indexed by either programmatically uploading to the Azure Blob storage account we just configured, or by browsing to that location in the Azure portal and using the upload feature there.

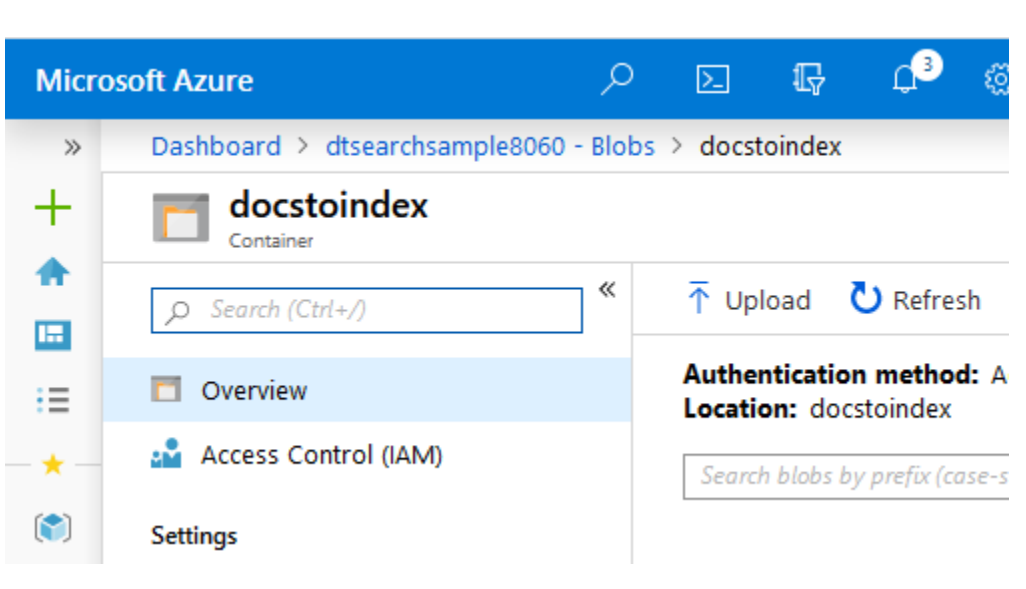

Figure 7 Uploading documents directly to an Azure Blob

Once our documents are uploaded, we can trigger the construction of our index by adding to the 'buildindex' queue that our function is monitoring. You can do this once again programmatically, with a tool, or even through the Azure Portal by navigating directly to the queue and clicking the 'Add Message' button.

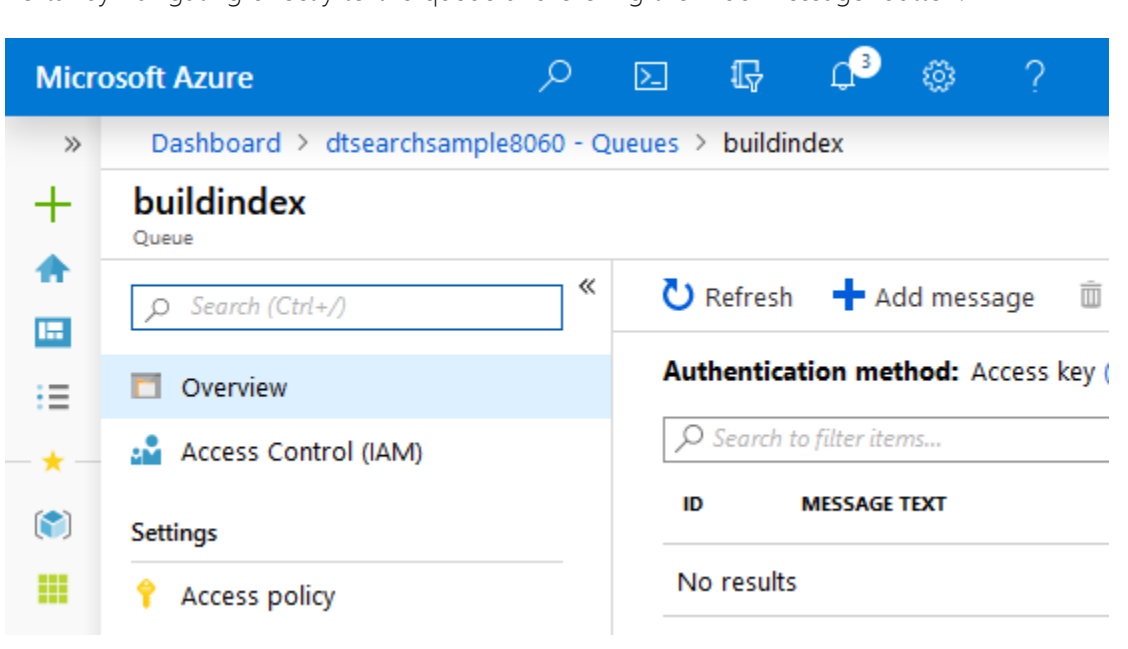

Figure 8 The Add Message button for a queue

The addition of that queue message will cause the index to be built and made available to our consumers. Then we can search the index and build clients to consume the XML search results from the dtSearch Engine.

Summary

Azure functions are an extremely versatile platform to operate on. The dtSearch tools allow us to build and manage a search index anywhere that we need intelligent search capabilities. The integration of these two technologies makes for a swift and pleasant search experience that can be accessed by an internet connected client.

More on dtSearch
dtSearch.com
A Search Engine in Your Pocket – Introducing dtSearch on Android
Blazing Fast Source Code Search in the Cloud
Using Azure Files, RemoteApp and dtSearch for Secure Instant Search Across Terabytes of A Wide Range of Data Types from Any Computer or Device
Windows Azure SQL Database Development with the dtSearch Engine
Faceted Search with dtSearch – Not Your Average Search Filter
Turbo Charge your Search Experience with dtSearch and Telerik UI for ASP.NET
Put a Search Engine in Your Windows 10 Universal (UWP) Applications
Indexing SharePoint Site Collections Using the dtSearch Engine DataSource API
Working with the dtSearch® ASP.NET Core WebDemo Sample Application
Using dtSearch on Amazon Web Services with EC2 & EBS
Full-Text Search with dtSearch and AWS Aurora

License

Share

About the Author

Jeffrey T. Fritz
Program Manager
United States

Jeff Fritz is a senior program manager in Microsoft's Developer Division working on the .NET Community Team. As a long time web developer and application architect with experience in large and small applications across a variety of verticals, he knows how to build for performance and practicality. Four days a week, you can catch Jeff hosting a live video stream called 'Fritz and Friends' …

show more

Comments and Discussions

You must Sign In to use this message board.

Spacing Relaxed Layout Normal Per page 25

-- There are no messages in this forum --