Azure Developer Immersion

You may have noticed that your app has a UI for search, but you will have been disappointed if you tried it. It is currently just a placeholder. You are going to fix that now by using Azure's search features.

There are two ways we could use Azure Search. It can integrate with SQL Server, automatically walking through the database periodically to build an index. In some ways that would be convenient—all the data is already in the database, so we will need some mechanism to load all that data into a searchable index. Why not use the SQL integration? There is one downside with the automated database crawler: there can be a significant delay between new information entering your system and it becoming available through search. For your purposes—a site designed to facilitate interactions between people— you want more immediacy. So you are going to use the search service in a mode that takes slightly more effort to configure, but will guarantee near-instant availability of data.
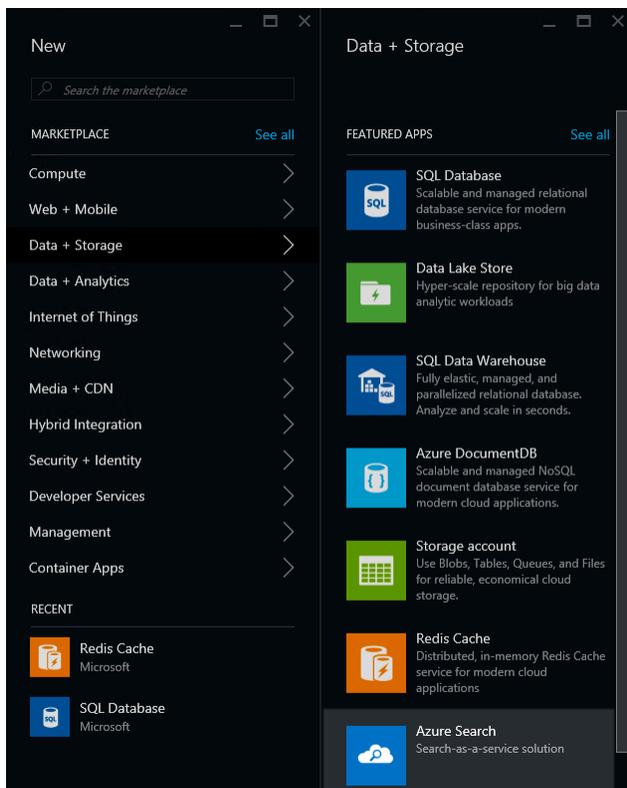
There is one exercise in this walkthrough:

1. Add Azure Search

Exercise 1: Add Azure Search

In this exercise you will add Azure Search to your app.

1. As before, grab the Task for this lab and move it to the In Progress state.

2. In the Azure portal, click the **+ New** button and click **Data + Storage**. Select **Azure Search**.

3. Enter a name for the service. This name will need to be globally unique.

4. Select the **rGroup** resource group.

5. Choose the same **Location** as for all the other steps.

6. Select the free **Pricing tier**.

7. Click **Create**.

8. Return to Visual Studio. As it happens, the code to load existing data into the search index has already been written. In **Solution Explorer**, right-click the Solution and choose **Add Existing Project**. Locate the **Rg.PopulateSearch** project in C:\VSTS\Repos\rGroup\Before and add it.

9. Set this application as the start-up project.

10. Double-click its **Properties** node in the **Rg.PopulateSearch** project and go to the **Debug** tab. It requires three arguments to run. Enter the name of your search service, its primary admin key (both available from your search service's panes in Azure), and the connection string to your SQL Azure database. It should look something like this:

Obviously you will need your own service's name and key, and your own SQL connection string.

Note: You will need to put the database connection string in quotes.

The code walks through all the messages, albums, pictures, and comments, and adds index entries containing their text, along with some contextual information enabling the relevant item to be located.

11. Run the program by pressing **F5**. It should take around half a minute. The exact time depends on how good your internet connection is.

12. Now that we have data in the index, we can add search capabilities to the app. Right-click on the **Rg.ServiceCore** project's **References** node and choose **Manage NuGet Packages**.

13. Make sure **Include prerelease** is checked.

14. Type **Microsoft.Azure.Search** into the search box.

15. For the version, select 0.13.0.preview and click Install.
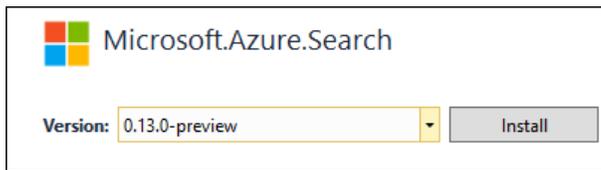
16. Close the **NuGet Package Manager** window.

17. In the **Rg.ServiceCore** project's **Operations** folder, open **SearchOperations**. This is the code that the existing search UI uses. As you can see there are some places with **TBD** comments, which we need to replace with working code.

18. Add these using directives at the top of the file:

```
using System;
using System.Configuration;
using System.Threading;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using SearchResult = Rg.ApiTypes.SearchResult;
using Hyak.Common;
```

19. Add this field and method to the class:

```
private static readonly Lazy<SearchIndexClient> _searchIndexClient = new
Lazy<SearchIndexClient>(
    CreateSearchIndex, LazyThreadSafetyMode.ExecutionAndPublication);

private static SearchIndexClient CreateSearchIndex()
{
    string searchServiceName = ConfigurationManager.AppSettings["SearchService"];
    string searchServiceKey = ConfigurationManager.AppSettings["SearchServiceKey"];
    if (searchServiceName == null || searchServiceKey == null)
    {
        return null;
    }

    var credentials = new SearchCredentials(searchServiceKey);
    var searchClient = new SearchServiceClient(searchServiceName, credentials);

    bool exists = false;
    try
    {
        IndexDefinitionResponse getResponse = searchClient.Indexes.Get(IndexName);
        exists = getResponse?.Index != null;
    }
    catch (CloudException)
    {
        // We expect this if the index does not yet exist.
    }

    if (!exists)
    {
        IndexDefinitionResponse createIndexResponse = searchClient.Indexes.Create(new
Index(
            IndexName,
            new[]
            {
                new Field("ItemId", DataType.String) { IsKey = true },
                new Field("Title", DataType.String) { IsSearchable = true },
                new Field("Content", DataType.String) { IsSearchable = true },
                new Field("CommentThreadId", DataType.Int32),
                new Field("TimelineEntryId", DataType.Int32),
                new Field("MediaAlbumId", DataType.Int32),
                new Field("UserMediaId", DataType.Int32)
            }));
    }

    return new SearchIndexClient(searchServiceName, IndexName, credentials);
}
```

This ensures the index is present and correctly configured. It is not strictly necessary since we have already populated the index but, if you were deploying a brand new instance of the site, this would enable it to configure the index correctly without needing to run the **Rg.PopulateSearch** program.

20. Add the **async** keyword after the **static** keyword of the **SearchAsync** method.

21. Replace the **SearchAsync** method's content with this:

```csharp
SearchIndexClient searchClient = _searchIndexClient.Value;
var results = new SearchResults
{
    TimelineMatches = new List<SearchResult>(),
    AlbumMatches = new List<SearchResult>(),
    MediaMatches = new List<SearchResult>()
};
if (searchClient != null)
{
    DocumentSearchResponse<MessageIndexEntry> response =
        await searchClient.Documents.SearchAsync<MessageIndexEntry>(term);
    foreach (SearchResult<MessageIndexEntry> resultEntry in response)
    {
        MessageIndexEntry result = resultEntry.Document;
        if (result.TimelineEntryId.HasValue)
        {
            results.TimelineMatches.Add(new SearchResult
            {
                Id = result.TimelineEntryId.Value,
                Text = result.Content
            });
        }
        else if (result.UserMediaId.HasValue)
        {
            results.MediaMatches.Add(new SearchResult
            {
                Id = result.UserMediaId.Value,
                Text = result.Title
            });
        }
        else if (result.MediaAlbumId.HasValue)
        {
            results.AlbumMatches.Add(new SearchResult
            {
                Id = result.MediaAlbumId.Value,
                Text = result.Title
            });
        }
    }
}
return results;
```

This uses the Azure Search service to look for matching items and packages the results up in a way that the client-side code will be able to process and display.

22. Find the **IndexItemAsync** method.

23. Add the **async** keyword after its **static** keyword.

24. Replace the method body with this:

```
SearchIndexClient indexClient = _searchIndexClient.Value;

if (indexClient == null)
{
    return;
}
var batchActions = new[]
{
    new IndexAction<MessageIndexEntry>(
            IndexActionType.Upload,
            entry)
};
await indexClient.Documents.IndexAsync(
    new IndexBatch<MessageIndexEntry>(batchActions));
```

This will add new items to the index, keeping it permanently up to date. The code to call this method has always been in place, so there is no need to do any work to plumb this in.

25. Find the **IndexMediaAsync** method. Replace the whole method with this:

```
public static async Task IndexMediaAsync(
    IEnumerable<UserMedia> media)
{
    var mediaToIndex = media
        .Where(m => !string.IsNullOrWhiteSpace(m.Title) &&
!string.IsNullOrWhiteSpace(m.Description?.Content))
        .ToList();
    if (mediaToIndex.Count == 0)
    {
        return;
    }
    SearchIndexClient indexClient = _searchIndexClient.Value;
    if (indexClient == null)
    {
        return;
    }
    var batchActions = mediaToIndex.Select(entry => new
IndexAction<MessageIndexEntry>(
        IndexActionType.Upload,
        new MessageIndexEntry
        {
            ItemId = "media-" + entry.UserMediaId,
            Title = entry.Title,
            Content = entry.Description.Content,
            UserMediaId = entry.UserMediaId,
            MediaAlbumId = entry.MediaAlbumId
        }));
    await indexClient.Documents.IndexAsync(
        new IndexBatch<MessageIndexEntry>(batchActions));
}
```

This does the same sort of job as the previous method, but it does it for media items, which require slightly different handling.

26. The code you have just added relies on some configuration settings being present. Go to your web site's pane in the Azure portal and open the **Application Settings**.

27. Add two entries called **SearchService** and **SearchServiceKey**. Set the values with your service name and key (you used them earlier to populate the index data). Click **Save**.

28. Back in Visual Studio, right-click on **Rg.Web**, select **Publish**, then click the **Publish** button**.**

29. When the site appears in a browser, log in if necessary.

> If you receive an error, you'll need to use the NuGet Package Manager and use the Consolidate feature and make sure the Rg.ServiceCore project is using the 6.0.4 version of the Newtonsoft.Jason library. Once you fix this publish the web site again.

30. Expand the menu.

31. Type some text that will match something you have already entered, such as a message, or the title or description of a photo or album, or a comment. Then click the **Search** button. You should see one or more results.
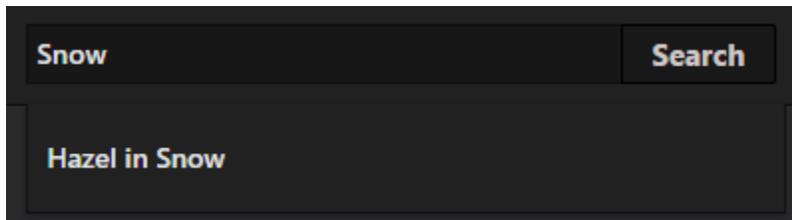
You should be able to click on search results to go to the relevant item.

32. To verify that the index is being updated live, go back to the home page and add a new message with a distinctive word in it. Then search for that distinctive word. It should find the newly added message.

> Don't forget to remove your password from any of the web.config files before you commit.

33. If everything's working, go back to Visual Studio. Commit and push your changes and mark your task as done.

Just to recap what you've done. You've created an instance of Azure's Search service and loaded it with your site's existing data. You then added code to use the search service to find items of interest and code to keep the index up to date at all times.