Notion Database Status Viewer using a Blazor Server App

# Features

Authentication

Notion API access

# Assumptions

You have a GitHub account

You have an Azure account

You have a Notion account

# Steps

1. Create a Visual Studio 2022 Blazor Server App project
   1. **NotionDatabaseStatusViewer**
   2. .NET 7.0
   3. Authentication: Individual Accounts
   4. Configure for HTTPS
2. Move ConnectionStrings from appsettings.json to secrets.json
   1. Expand ‘Connected Services’ in the Solution Explorer
   2. Right-click on ‘Secrets.json (local)
   3. Click ‘Manage user secrets
   4. Cut the ConnectionStrings entry in appsettings.json and paste in secrets.json
3. Remove items from project
   1. SurveyPrompt
      1. Delete SurveyPrompt.razor
      2. Remove SurveyPrompt from Index.razor
   2. Counter
      1. Delete Counter.razor
   3. FetchData
      1. Delete FetchData.razor
      2. In the Data folder
         1. Delete WeatherForecast.cs
         2. Delete WeatherForecastService.cs
      3. In Program.cs
      4. Remove builder.Services.AddSingleton<WeatherForecastService>();
      5. In MainLayout.razor, remove the About link.
   4. NavMenu
      1. Delete NavMenu.razor
      2. In MainLayout.razor, remove the div that contains <NavMenu />
4. Setup the Entity Framework database that is used for registering and logging in users
   * 1. Add a scaffolded item to the project
     2. Click Identity tab, click Identity section, click Add
     3. Select Account\Login, Account\Logout, and Account\Register
     4. Choose the default DbContext
     5. You can choose to overwrite the existing Logout
     6. ~~In the Package Manager Console~~
        1. ~~Add-Migration InitialCreate~~
        2. ~~Update-Database~~
5. Publish to Azure
   1. Create a publish profile for Azure by right-clicking on the project and clicking on ‘Publish’
   2. Click ‘Azure’ and ‘Next’
   3. Click ‘Azure App Service (Windows)’ and ‘Next’
   4. Select your subscription
   5. Click ‘+’ to create a new App Service (this is the application)
      1. Fill in the name. This must be unique.
      2. Create a resource group. This is a grouping of the items that will be created for all of the Azure resources we will create for this application. Take note of the location to be close to the users.
   6. Select a Hosting Plan by clicking ‘New’ next to it. This is the physical server where the application will reside.
      1. Fill in the name. This must be unique.
      2. Choose the location to be close to the users.
      3. Choose ‘Free’ for a simple application.
      4. Click OK.
      5. Click ‘Create’
      6. Click ‘Finish’
      7. Click ‘Close’
   7. Set up the SQL Server Database
      1. Click the three dots (…) to the right of ‘SQL Server Database’
      2. Click ‘Connect’
      3. Click ‘Azure SQL Database’ and ‘Next’
      4. Click ‘+’ to create a new database.
      5. Fill in name
      6. Click ‘New’ next to ‘Database Server’ to create the database server. This is the physical server were the database resides.
      7. Choose a database server name. This must be unique.
      8. Choose the location to be close to the users.
      9. Fill in the administrator username and password.
      10. Click ‘OK’.
      11. Click ‘Create’.
      12. Click ‘Next’ to select the newly created database.
   8. Connect to the Azure SQL Database
      1. Enter the admin username and password.
      2. Copy the connection string value
      3. Select Save connection string value in ‘Azure Key Vault’. Click ‘Next’.
   9. Create a new Azure Key Vault
      1. Click ‘Create New’
      2. Choose the location to be close to the users.
      3. Choose ‘Standard’ SKU.
      4. Click ‘Create’.
      5. Click ‘Next’ to choose the newly created Key Vault.
   10. Connect to Azure Key Vault
       1. Click ‘Next’.
       2. Click ‘Finish’.
          1. If it fails
             1. Click ‘Cancel’
             2. Manually install NuGet package ‘Azure Identity’.
             3. Repeat the section ‘Set up the SQL Server Database’ above.
       3. Click ‘Close’.
   11. Update the database in Azure
       1. Change the connection string in secrets.json to the connection string for the database in Azure.
       2. Go to the Package Manager Console
       3. Enter ‘Add-Migration InitialCreate’.
       4. Enter ‘Update-Database’.
   12. Click ‘Publish’ (near the top of the Publish screen).
6. Create Git repository
   1. In ‘Git Changes’ panel, click on ‘Create Git Repository’
7. Setup SendGrid
   1. Sign up for SendGrid at <https://www.sendgrid.com>
   2. Go to Settings 🡪 API Keys
   3. Create API key
   4. Copy the value of the Key that is shown (only once)
   5. Put in Key Vault
8. Implement Authentication
   1. Create a ‘Services’ folder
   2. Add a class called AuthMessageSenderOptions.cs

public class AuthMessageSenderOptions

{

public string? SendGridKey { get; set; }

}

* 1. Add NuGet packages for the next step where we add the EmailSender class
     1. SendGrid
  2. Add an implementation of IEmailSender called EmailSender.cs in the Services folder. Correct the namespace.

using Microsoft.AspNetCore.Identity.UI.Services;

using Microsoft.Extensions.Options;

using SendGrid;

using SendGrid.Helpers.Mail;

namespace TeliriteStatusWebsite.Services

{

public class EmailSender : IEmailSender

{

private readonly ILogger \_logger;

public AuthMessageSenderOptions Options { get; }

public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor, ILogger<EmailSender> logger)

{

Options = optionsAccessor.Value;

\_logger = logger;

}

public async Task SendEmailAsync(string toEmail, string subject, string message)

{

if (string.IsNullOrEmpty(Options.SendGridKey))

{

throw new Exception("Null SendGridKey");

}

await Execute(Options.SendGridKey, subject, message, toEmail);

}

public async Task Execute(string apiKey, string subject, string message, string toEmail)

{

SendGrid.Response? response = null;

var client = new SendGridClient(apiKey);

var msg = new SendGridMessage

{

From = new EmailAddress("<sender email address>", "<optional name>"),

Subject = subject,

PlainTextContent = message,

HtmlContent = message

};

msg.AddTo(new EmailAddress(toEmail));

// Disable click tracking

// See See https://sendgrid.com/docs/User\_Guide/Settings/tracking.html

msg.SetClickTracking(enable: false, enableText: false);

response = await client.SendEmailAsync(msg);

\_logger.LogInformation(response.IsSuccessStatusCode

? $"Email to {toEmail} queued successfully"

: $"Email to {toEmail} failed");

}

}

}

* 1. Add dependency injection for EmailSender in Program.cs

builder.Services.AddTransient<IEmailSender, EmailSender>();

builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

1. Add the Azure Key Vault to Program.cs and update the Uri.

SecretClientOptions options = new()

{

Retry =

{

Delay = TimeSpan.FromSeconds(2),

MaxDelay = TimeSpan.FromSeconds(16),

MaxRetries = 5,

Mode = Azure.Core.RetryMode.Exponential

}

};

var client = new SecretClient(new Uri("<Key Vault Uri>"), new DefaultAzureCredential(), options);

builder.Configuration.AddAzureKeyVault(client, new AzureKeyVaultConfigurationOptions());

1. Change the application to read the database using the API
   1. Add a folder called ‘NotionTypes’ to the Solution Explorer
   2. Add a class to the ‘NotionTypes’ called DataBaseQueryResponse.cs (this is one of the classes that is used for parsing the value returnd from the API call to Notion



* 1. Add a class to the ‘NotionTypes’ called TableRow.cs (this is the class that holds the names of the columns in the database).

internal class TableRow

{

public string Project { get; set; } = string.Empty;

public string Status { get; set; } = string.Empty;

}

* 1. Continue to add classes for the NotionTypes
     1. Add class Page.cs

internal class Page

{

public Properties properties { get; set; } = new();

}

* + 1. Add class Properties.cs

internal class Properties

{

public TitleType Project { get; set; } = new();

public RichTextType Status { get; set; } = new();

}

* + 1. Add class TitleType.cs

internal class TitleType

{

public List<TextType> title { get; set; } = new();

}

* + 1. Add class RichTextType.cs

internal class RichTextType

{

public List<TextType> rich\_text { get; set; } = new();

}

* + 1. Add class TextType.cs

internal class TextType

{

public Text text { get; set; } = new();

}

* + 1. Add class Text.cs

internal class Text

{

public string content { get; set; } = string.Empty;

}

* 1. Add the dependency injection for the HttpClient in Program.cs

builder.Services.AddHttpClient();

* 1. Add the following to Index.razor

@using NotionDatabaseViewer.NotionTypes;

@using System.Text

@using System.Text.Json

@inject HttpClient httpClient

@inject IConfiguration config

<h1>Status</h1>

<**AuthorizeView** **Roles**="status-viewer">

<**Authorized**>

<table>

<tr>

<th class="project-column">Project</th>

<th class="status-column">Status</th>

</tr>

@foreach (TableRow row in results)

{

<tr>

<td class="project-column">@row.Project</td>

<td class="status-column">@row.Status</td>

</tr>

}

</table>

</**Authorized**>

<**NotAuthorized**>

<p>To see the status, you will need to be logged in and have authorization to view it. If you need authorization, please request it by contacting 'contact email address' for authorization.</p>

</**NotAuthorized**>

</**AuthorizeView**>

private string databaseId = "";

private string bearerToken = "";

private string responseMessage = "";

private string notionContent = "";

private string reasonPhrase = "";

private List<TableRow> results = new();

protected override async Task OnInitializedAsync()

{

databaseId = config["DatabaseId"] ?? "";

bearerToken = config["BearerToken"] ?? "";

if (databaseId == string.Empty || bearerToken == string.Empty)

{

responseMessage = "Error getting configuration";

return;

}

await GetDatabaseContent(databaseId);

ProcessNotionResult(notionContent);

}

private async Task GetDatabaseContent(string databaseId)

{

string requestUri = $"https://api.notion.com/v1/databases/{databaseId}/query";

httpClient.DefaultRequestHeaders.Add(name: "Authorization", value: $"Bearer {bearerToken}");

httpClient.DefaultRequestHeaders.Add(name: "Notion-Version", value: "2022-06-28");

// Note on using relations in the Notion API:

// database\_id is the id of the related database; open the related database as a page and get the id from the Url.

// contains is the page of the item of interest in the related database; open the item in the related database as a page and get the id from the Url

// The related database needs to also have the Notion integration.

string contentString = "";

HttpContent content = new StringContent(contentString, Encoding.UTF8, "application/json");

HttpResponseMessage response = new HttpResponseMessage();

try

{

response = await httpClient.PostAsync(requestUri, content);

}

catch (Exception ex)

{

responseMessage = $"Exception: {ex}";

}

if (response.IsSuccessStatusCode)

{

responseMessage = "Success";

notionContent = await response.Content.ReadAsStringAsync();

}

else

{

responseMessage = $"Error: {response.Content}";

reasonPhrase = $"Error: {response.ReasonPhrase}";

}

}

private void ProcessNotionResult(string result)

{

DataBaseQueryResponse response = new();

try

{

response = JsonSerializer.Deserialize<DataBaseQueryResponse>(result, JsonSerializerOptions.Default) ?? new();

}

catch (JsonException ex)

{

responseMessage = $"Error deserializing data. {ex.Message}";

return;

}

if (response is null)

{

responseMessage = "Error deserializing data";

return;

}

for (int row = 0; row < response.results.Count; row++)

{

string project = string.Empty;

string status = string.Empty;

if (response.results[row].properties.Project.title.Count() > 0)

{

project = response.results[row].properties.Project.title[0].text.content;

}

if (response.results[row].properties.Status.rich\_text.Count() > 0)

{

status = response.results[row].properties.Status.rich\_text[0].text.content;

}

results.Add(new TableRow { Project = project, Status = status });

}

}

* 1. Add a role for viewing. I’ll call it ‘status-viewer’.
  2. Add the role item to Program.cs

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)

.AddRoles<IdentityRole>()

.AddEntityFrameworkStores<TeliriteStatusWebsiteContext>();

1. Set up Notion
   1. Create an Integration with READ access.
   2. Settings & Members 🡪 My Connections 🡪 Develop or manage integrations
   3. Click ‘+ New Integration’
      1. Basic Information
         1. Enter a name.
      2. Capabilities
         1. Read content
      3. Click ‘Submit’.
      4. Copy the Internal Integration Secret for later
      5. Add it to the Azure KeyVault. It is called BearerToken.
   4. Create a new page
   5. Add a database to it by entering ‘/database inline
   6. Click the three dots (…) for the page and then ‘Add connections’
   7. Click your integration and click ‘Confirm’.
2. Add some content to the database
   1. Add a title (starts as ‘Untitled’)
   2. The first column has a fixed type called ‘Title’. Change the name to ‘Project’.
   3. Change the second column name to Status and make it type ‘Text’
   4. Fill out two rows with information
   5. Delete other rows
3. Clean up the application
   1. Register.cshtml
      1. Remove the div about Using another Service
   2. Login.cshtml
      1. Remove div about Using another Service
   3. Add styling in site.css

