Put a Search Engine in Your Windows 10 Universal (UWP) Applications

Jeffrey T. Fritz, 5 Jul 2016

Introducing the dtSearch Engine for UWP. Also available from this author (see article for links): the dtSearch Engine for Android and advanced faceted searching using the dtSearch Engine.

Editorial Note

This article is in the Product Showcase section for our sponsors at CodeProject. These articles are intended to provide you with information on products and services that we consider useful and of value to developers.

I've spent some time working with the dtSearch library to add search functionality to a number of applications on the web and on my Android device. In this article, I'll show you how to add a simple full-text document search to a Windows 10 Universal application.

Configuration

To get started, I grab a copy of the dtSearch library and native DLL to add to my project. The dtSearch Engine for UWP has two components:

dtSearchEngine_uwp_Win32.dll (C/C++ API)
dtSearchUwpApi.dll (managed code API for C#, Visual Basic, or C++)

dtSearchUwpApi.dll provides a managed API wrapper around *dtSearchEngine_uwp_Win32.dll*, with an API that is very similar to the .NET API. I add the managed library as a reference in my project just like any other DLL and add the native DLL to my project and set its "Build Action" property to "Content" so it will deploy with my application.

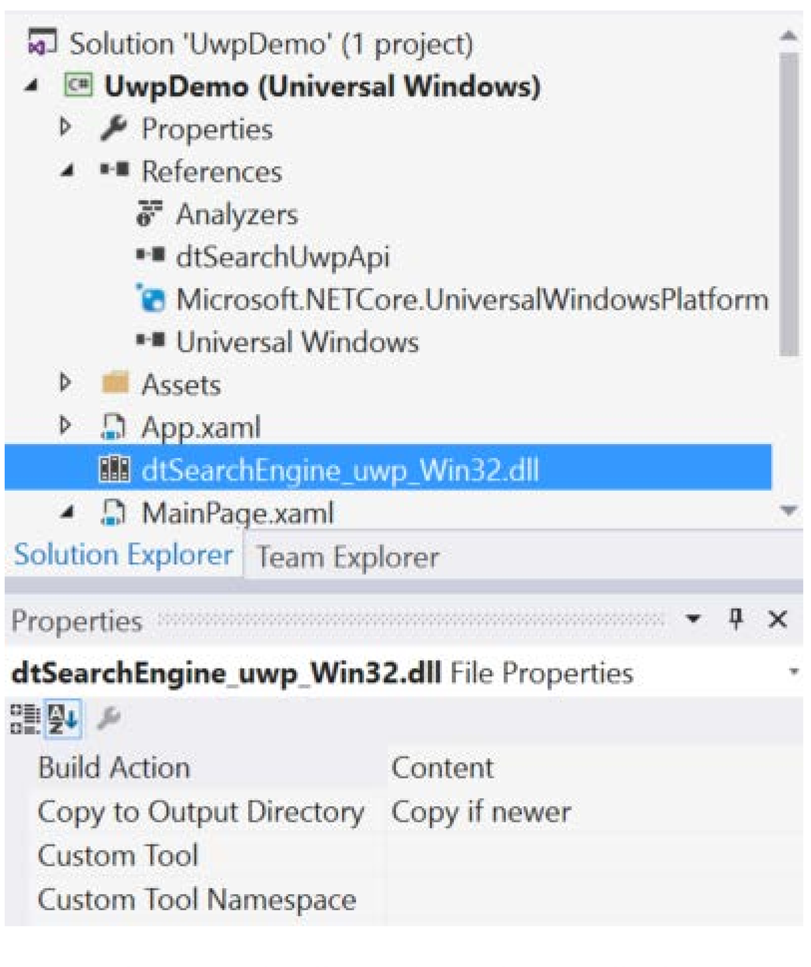

Figure 1- Native Library reference in my application

I want to build a searchable index of some famous American documents so I can reference them very quickly whenever one of those social media discussions comes up where folks like to reference the laws of the land. I start by copying some of the transcriptions of these famous documents from the United States Archives online and generating some text files or Word docs in a Docs folder in my project. As with the native DLL, I mark each of these documents as "Content" and "Copy if newer". This way, they're copied into the APPX and will be available in the application folder when the application is deployed. This is important because I want to be able to display the entire document and to highlight search terms.

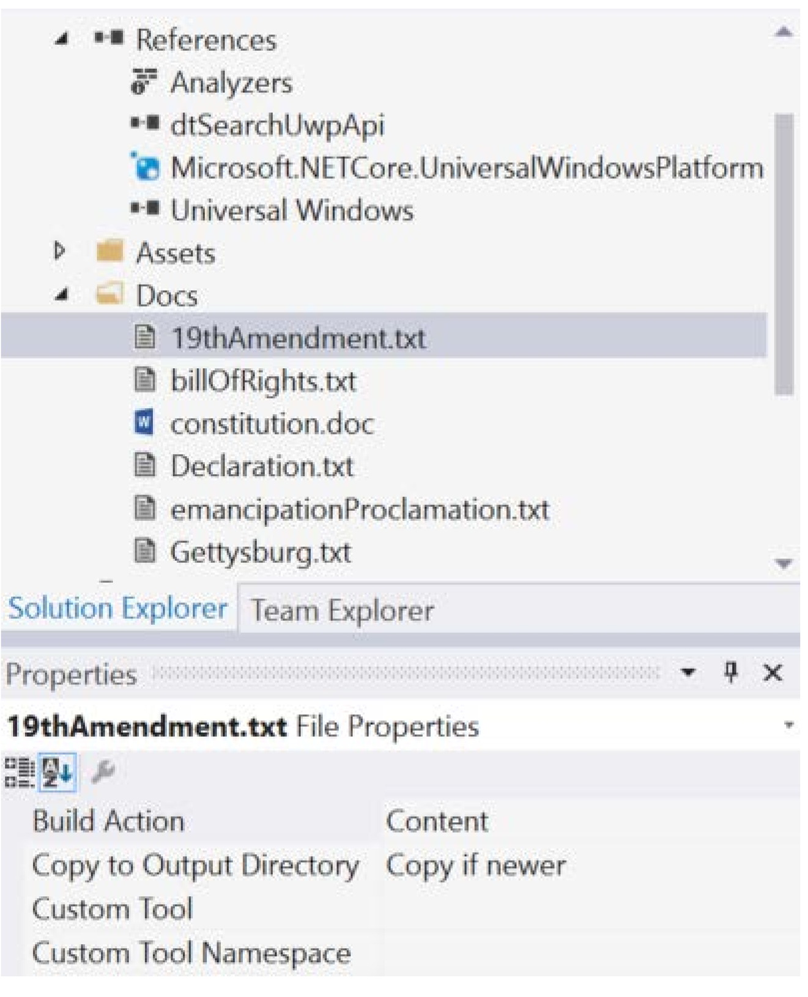

Figure 2 - Configuration of document content

Indexing Our Documents

Before any search operations can take place, I need dtSearch to build an index of my documents. For this demo, I'll add the index operation into the application. If my users want to add documents to the index from the running application, then they will need the ability to update the index after adding documents. To implement this, I add a "Build Index" button that will add the contents of my Docs folder to the index. The application can infer the Docs and Index folders' locations from the standard ApplicationData and Package objects. The index folder has to go under the ApplicationData object's LocalFolder because it is read/write, and the Docs folder will go under the Package object's *InstalledLocation* folder.

```
private String indexPath;
private String docFolder;

public MainPage()
{
  this.InitializeComponent();
  indexPath = Windows.Storage.ApplicationData.Current.LocalFolder.Path + Path.DirectorySeparatorChar +
"index";
  docFolder = Windows.ApplicationModel.Package.Current.InstalledLocation.Path + Path.DirectorySeparatorChar +
"Docs";
}
```

With those locations, I can now connect the `buildIndexButton_Click` event to the dtSearch IndexJob operation. The event handler method starts by defining some other folder location options necessary for dtSearch to be able to build an index:

```
private void buildIndexButton_Click(object sender, RoutedEventArgs e)
{

        // Setting Options.TempFileDir and Options.HomeDir is required to tell the dtSearch Engine
        // where to find configuration files and where to store temp data.
        Options options = new Options();
        options.TempFileDir = Windows.Storage.ApplicationData.Current.TemporaryFolder.Path;
        options.HomeDir = Windows.ApplicationModel.Package.Current.InstalledLocation.Path;
        options.Save();
```

Creating a search index of the important documents in my *Docs* folder is just a few lines of code then:

```
// Build the index
dtSearch.Engine.IndexJob indexJob = new IndexJob();
indexJob.IndexPath = indexPath;
indexJob.ActionCreate = true;
indexJob.ActionAdd = true;
indexJob.FoldersToIndex.Add(docFolder);
indexJob.IncludeFilters.Add("*.txt");
indexJob.IncludeFilters.Add("*.doc");
indexJob.Execute();
```

With these eight lines, I've created a job to build the index and instructed it where to write its output. I've told it to create the index if it doesn't exist and to add records to it. The `FoldersToIndex` collection has one folder, *docFolder*, that I captured in the constructor of this class and assigned to the deployed location of the *Docs* folder. The `IncludeFilters` collection specifies the filename extensions of the documents to index. Finally, the `Execute()` call builds the index.

Type-Ahead Searching

The dtSearch engine provides a word list feature that can offer some quick type-ahead or intellisense behavior for a term that you are looking for. In my case, I'll create a search textbox on the page and provide type-ahead lookup as the text in the textbox changes.

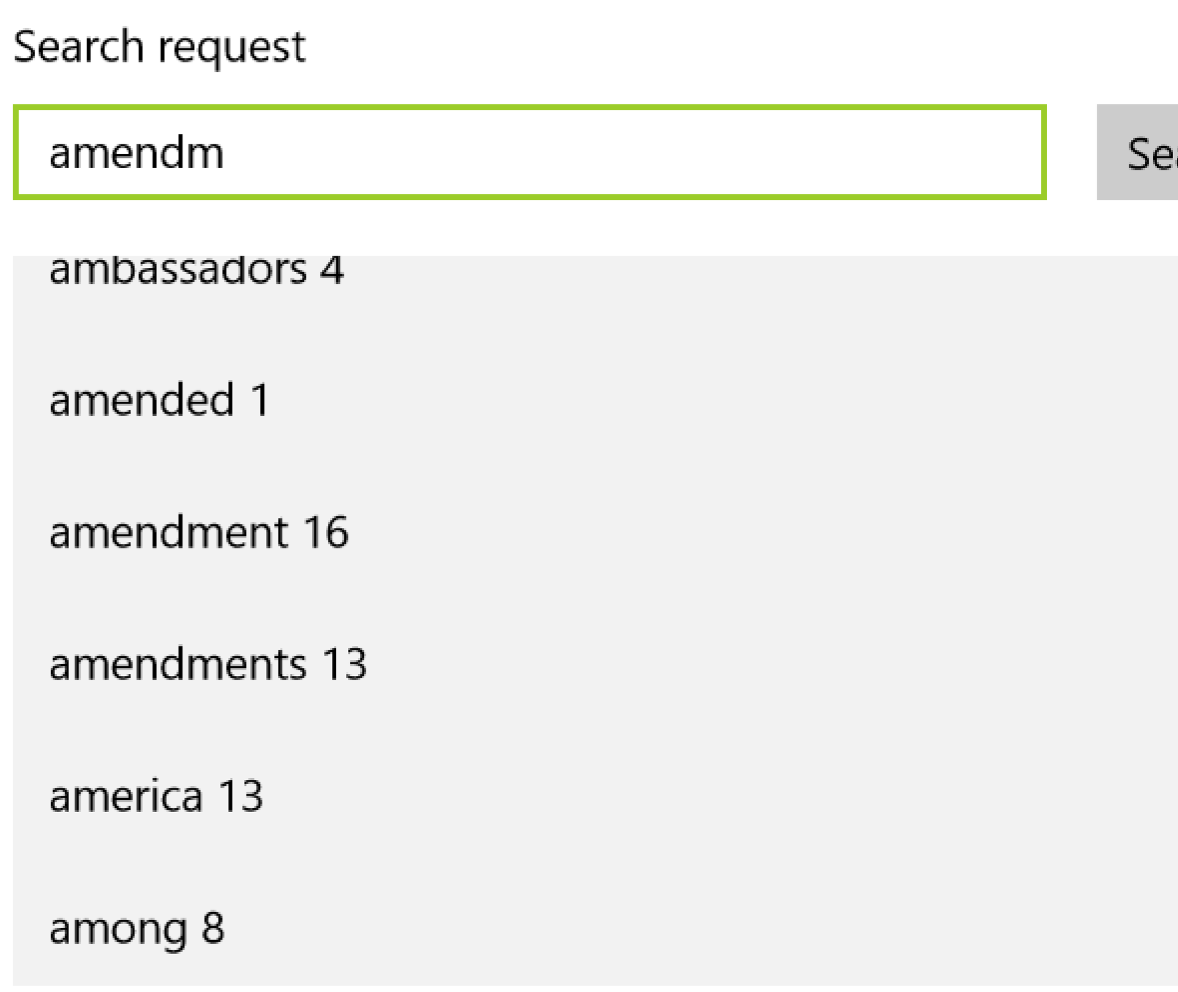

Figure 3 - TypeAhead searching from a textbox in a UWP application

I can provide these simple results in a ListBox by configuring the `TextChanged` eventhandler for the Search Request textbox. Listing the results of the search involves just a few more lines of code:

```
private void searchRequest_TextChanged(object sender, TextChangedEventArgs e)
{
  if (wordListBuilder == null)
  {
    wordListBuilder = new WordListBuilder();
    wordListBuilder.OpenIndex(indexPath);
  }
  wordListBuilder.ListWords(searchRequest.Text, 5);
  fileList.Items.Clear();
  for (int i = 0; i < wordListBuilder.Count; ++i)
  {

  fileList.Items.Add(wordListBuilder.GetNthWord(i) +
     " " + wordListBuilder.GetNthWordCount(i));
  }
}
```

In this code, I construct a `wordListBuilder` object at the class level that references the index on disk so I don't have to re-initialize it on every keypress. The ListWords method grabs the five words before and after the search term in the index. After clearing the fileList ListBox, I populate the ListBox with the words found and the number of occurrences of each word.

Searching Files and Reporting Results

After a search, I want to be able to display the list of documents, so I can locate that term's use and win my arguments on social media. To search for the request in the textbox, I create a handler for the search button click to perform the search:

```
private dtSearch.Engine.SearchResults searchResults;
private void searchButton_Click(object sender, RoutedEventArgs e)
{
  fileList.Items.Clear();
  dtSearch.Engine.SearchJob searchJob = new SearchJob();
  if (searchResults != null)
  {
    searchResults.Dispose();
    searchResults = null;
  }
  searchResults = new SearchResults();
  searchJob.IndexesToSearch.Add(indexPath);
  searchJob.Request = searchRequest.Text;
  searchJob.MaxFilesToRetrieve = 10;
  searchJob.AutoStopLimit = 100;
  searchJob.TimeoutSeconds = 3;
  searchJob.Execute(searchResults);
```

This code clears the ListBox that may have contained suggested words or the results from a previous search. Then it creates a `SearchJob` and initializes the `SearchResults` to receive the output from the search engine. It points the `IndexesToSearch` collection at the `indexPath` created earlier, and defines some sensible limits for the maximum files, the number of files to stop after searching, and the timeout for the search in seconds. Calling the `Execute()` method executes the search and populates the `SearchResults` object with the results of the search.

I can then use a standard for-loop to iterate over the `SearchResults` and place those values into the ListBox in my user interface:

```
for (int i = 0; i < searchResults.Count; ++i)
{
  SearchResultsItem item = new SearchResultsItem();
  if (searchResults.GetNthDoc(i, item))
  {
    string name = item.Filename;
    if (name.StartsWith(docFolder))
    {
      name = name.Substring(docFolder.Length + 1);
    }
    fileList.Items.Add(name);
  }
}
```

The "If" statement in this block checks that the relevant document records can be read from the index, which should always be the case unless the index has unexpectedly become inaccessible.

Showing Content

Next I'd really like to display the context of the searched terms in the document. I create a final event handler on the `SelectionChanged` event for the ListBox and use the dtSearch `FileConverter` to output some nicely formatted HTML with some context highlighting applied to the terms. I add a WebView to my interface and use that to display the output of my highlighting:

```
private void fileList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
  int iItem = fileList.SelectedIndex;
  using (FileConverter fc = new FileConverter())
  {
    if (fc.SetInputItem(searchResults, iItem))
    {
      fc.BeforeHit = "<b style='background-color: yellow'>";
      fc.AfterHit = "</b>";
      fc.OutputToString = true;
      fc.OutputFormat = OutputFormat.itHTML;
      fc.Execute();
      webView.NavigateToString(fc.OutputString);
    }
  }
}
```

This code sets up the `FileConverter` with the syntax highlighting HTML markup in the `BeforeHit` and `AfterHit` properties and HTML as the `OutputFormat`. After calling `Execute()`, the `FileConverter`'s `OutputString` property has an HTML view of the document, with hits marked, to display in the WebView control using `webView.NavigateToString`.

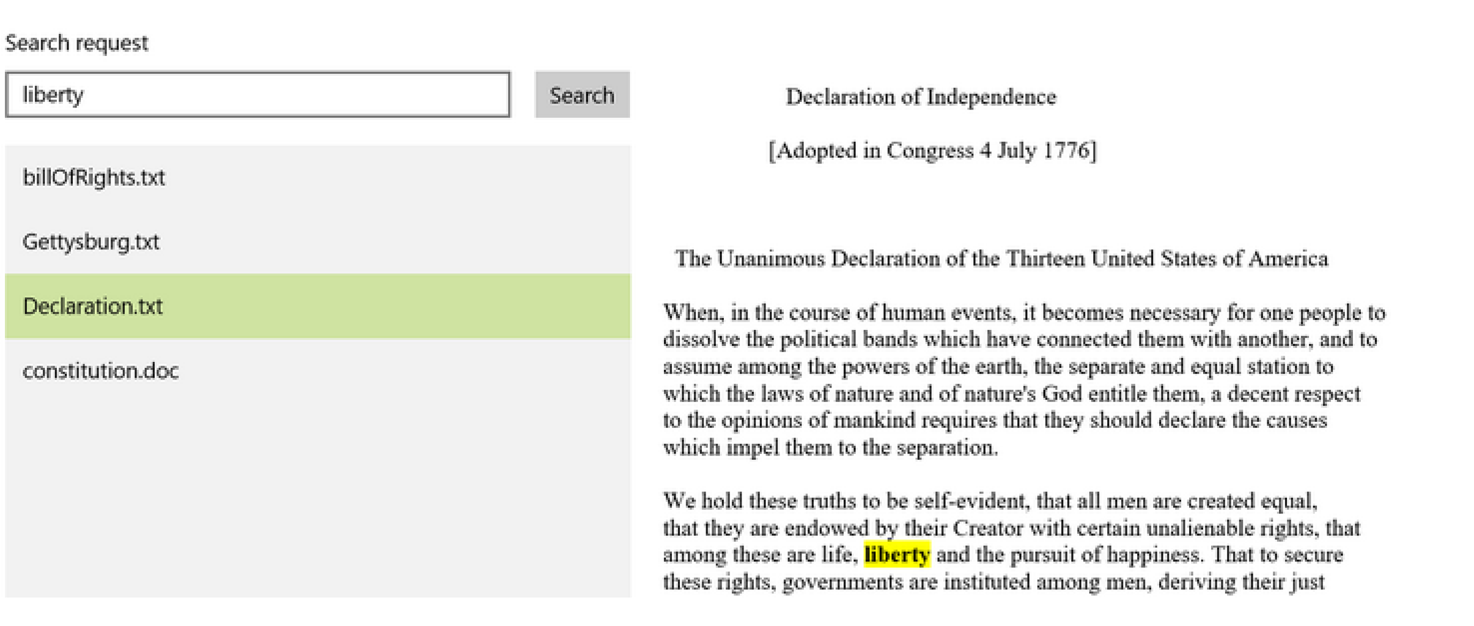

Figure 4 - Search result with highlighting

Summary

dtSearch has made it easy for me to add rich search functionality to this project. With additional features like search across fields and faceted search, there are many options for my next project. The evolution of dtSearch to add UWP support along with its other supported frameworks cements its position in my developer toolbox as my go-to search library.

More on dtSearch

- 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

License

Share

About the Author

Jeffrey T. Fritz
Program Manager
United States

Jeffrey is a software developer coach, architect, and speaker in the Microsoft.Net community. He currently works as a program manager for the Microsoft .NET Developer Outreach group. He has delivered training videos on Pluralsight, WintellectNow, and on YouTube. Jeffrey makes regular appearances delivering keynotes, workshops, and breakout sessions at conferences such as TechEd, Ignite, DevIntersection, CodeStock, FalafelCon, VSLive as well as user group meetings in an effort to grow the next generation of software developers.

Comments and Discussions

2 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1110623/Put-a-Search-Engine-in-Your-Windows-Universal-UW** to post and view comments on this article, or click **here** to get a print view with messages.