

## Really Simple Search with C#

Everyone is talking about Search technology at the moment - how Google has risen to the top of the heap, how Yahoo is trying to regain its former number one spot, and how Microsoft is playing catch-up. But for the average ASP.NET developer, those sites are really about helping people find you on the Web. Once they've visited your website, how do you provide a cheap, fast, customised search to maximise the usability of your content?

There are a number of options available:

Search 'Technology'	Advantages	Disadvantages
Microsoft Index Server	Comes with Windows 2000, XP, 2003	File-system indexing only, doesn't spider website links or database-driven pages (there are tricks around this)
Other server-side software eg. DTSearch, mnoGoSearch	Shop around for features that you need, including multiple language support	Cost May be difficult to setup/customise
'Hosted services' eg. Google, PicoSearch	Often free or low cost Easy to set up	Lack of control Often template driven or host ads which may distract your users

Most website operators will find at least one of these products can meet their needs, but it will always be a trade-off between cost, features and flexibility.

This article describes a simple, free, easy to install Search feature. The goal is to build a simple search tool that can be installed simply by placing three files on a website, and that could be easily extended to rival the features of the products listed above!

There are two main parts to a Search engine:

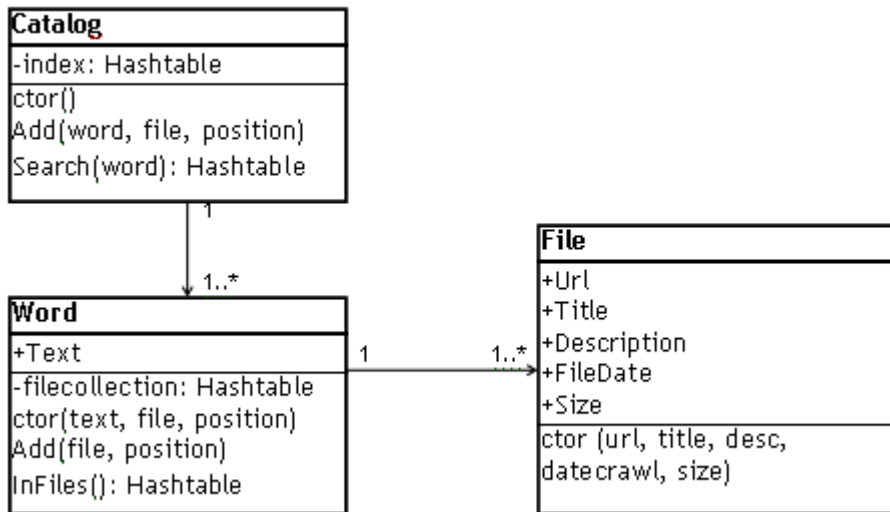
- the build process, which processes files, indexing their contents and creating the 'catalog'
- the search process, which uses the 'catalog' to find the search term and the names of the files it appears in

### Design

A Catalog contains a collection of Words, and each Word contains a reference to every File that it appears in
--

The first step was to think about how to implement the catalog objects. A Binary Search Tree seemed like a good idea (see the [great articles](#) on MSDN), but in order to keep things simple Hashtables will do the job. We can always refactor the code to use a more sophisticated Collection class later on.

The simple object model looks like this:



You can see that some assumptions have been made in this model.

Firstly, we store limited information about the **File** - just enough to produce a familiar search results page:

- **Url** - a web-based address for the file (this will become important later)
- **Title** - the <TITLE> of the document
- **FileDate** - date the file was last modified
- **Size** - in bytes
- **Description** - a 'summary' of the document

The **Word** object is even simpler - the properties are:

- **Text** - the actual word! We will standardise on lowercase for all the data stored we store
- **InFiles** - the collection of Files that this Word was found in

Lastly, the **Catalog** itself has a single property - the collection of Words called **index**. It also has two methods, one to *add* Words to the catalog and another to *search* the catalog and get back a list of files (the *search results*).

There are two important assumptions which aren't immediately apparent from the model - there should only be ONE File object for each physical file, and ONE Word object for each word (so there will only be one Word object that represents the word "microsoft" for example), although that word will appear in many of the files we search. Why this is so, and how we manage it is covered in the catalog build process.

## Code Structure

Searcharoo.cs	Implementation of the object model; compiled into both ASPX pages
SearcharooCrawler.aspx	<pre>&lt;%@ Page Language="C#" Src="Searcharoo.cs" %&gt;</pre> <pre>&lt;%@ import Namespace="Searcharoo.Net" %&gt;</pre> Code to build the catalog using the common classes, and place the resulting Catalog object in the ASP.NET Application Cache
Searcharoo.aspx	<pre>&lt;%@ Page Language="C#" Src="Searcharoo.cs" %&gt;</pre> <pre>&lt;%@ import Namespace="Searcharoo.Net" %&gt;</pre> Retrieves the Catalog object from the Cache and allows searching via an HTML form.

## Object Model [Searcharoo.cs]

This file contains the C# code that defines the object model for our catalog, including the methods to add and search Words. These objects are used by both the crawler and the search page.

```
namespace Searcharoo.Net {
    public class Catalog {
        private System.Collections.Hashtable index;
```

```

public Catalog () {}
public bool Add (string word, File infile, int position){}
public Hashtable Search (string searchWord) {}
}

public class Word {
public string Text;
private System.Collections.Hashtable fileCollection;
public Word (string text, File infile, int position) {}
public void Add (File infile, int position) {}
public Hashtable InFiles () {}
}

public class File {
public string Url;
public string Title;
public string Description;
public DateTime CrawledDate;
public long Size;
public File (string url, string title, string description, DateTime datecrawl, long length) {}
}
}

```

Listing 1 - Overview of the object model (interfaces only - implementation code has been removed)

## Build the Crawler [SearcharooCrawler.aspx]

Now that we have a model and structure, what next? In the interests of 'getting something working', the first build task is to simulate how our 'build' process is going to find the files we want to search. There are two ways we can look for files

- i. Spidering - following 'the web' of links in HTML pages to search an entire website (or sites)
- ii. Crawling - crawling through a set of files and folders and indexing all the files in those folders, using the file system. This can only work when the files are locally accessible.

The big search engines - Yahoo, Google, MSN - all spider the internet to build their search catalogs. However following links to find documents requires us to write an HTML parser that can find and interpret the links, and then follow them! That's a little too much for one article, so we're going to start with some simple file crawling code to populate our catalog. The great thing about our object model is that it doesn't really care if it is populated by Spidering or Crawling - it will work for either method, only the code that populates it will change.

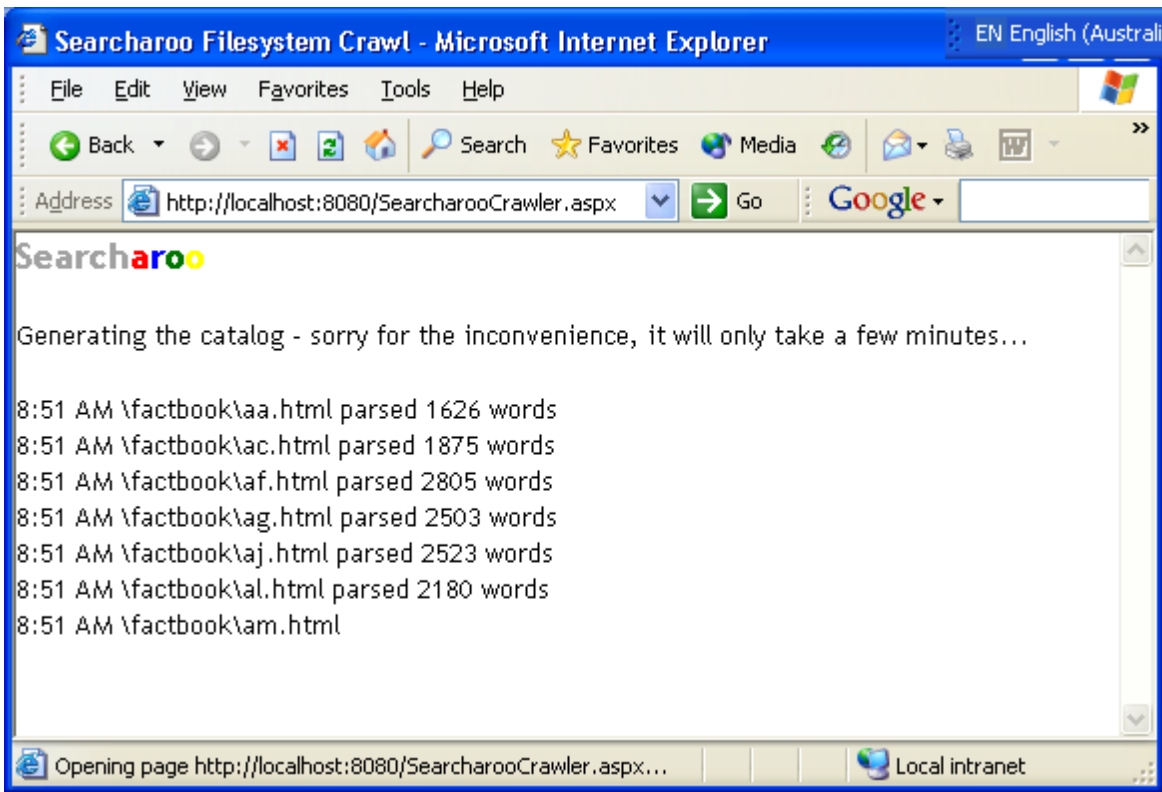
Here is a simple method that we can use to locate the files we want to search by traversing the file system:

```

private void CrawlPath (string root, string path) {
System.IO.DirectoryInfo m_dir = new System.IO.DirectoryInfo (path);
// ### Look for matching files to summarise what will be catalogued ###
foreach (System.IO.FileInfo f in m_dir.GetFiles(m_filter)) {
Response.Write (path.Substring(root.Length) + @"\" + f.Name + "<br>");
} // foreach
foreach (System.IO.DirectoryInfo d in m_dir.GetDirectories()) {
CrawlPath (root, path + @"\" + d.Name);
} // foreach
}

```

Listing 2 - Crawling the filesystem



Screenshot 1 - To test the file crawler we downloaded the HTML from the [CIA World FactBook](#)

Now that we are confident we can access the files, we need to process each one in order to populate the catalog. Firstly, let's be clear about what that process is:

1. get the list of files and folders in the root directory (done)
2. open the first file and read its contents
3. look for the file's Title, Description and calculate its size
4. generate the file's Url (because we're crawling the file-system, but we want the file to have a web address to click on).
5. clean up the text into a collection of words
6. add each word to the catalog, linked to this file
7. close the file and open the next one (or open a directory once all the files are processed)
8. repeat until no more files are found

There's three different coding tasks to do:

- a. opening the files we find - we'll use the System.IO namespace for this
- b. finding specific text in the file (the Title and Description) - either the System.String static methods or the System.RegularExpressions namespaces might help here
- c. cleaning up the text and parsing it into individual words - definitely a job for RegularExpressions.

Getting (a) working was easy:

```
System.IO.DirectoryInfo m_dir = new System.IO.DirectoryInfo (path);
// Look for matching files
foreach (System.IO.FileInfo f in m_dir.GetFiles(m_filter)) {
    Response.Write (DateTime.Now.ToString("t") + " "
        + path.Substring(root.Length) + @"\" + f.Name ); Response.Flush();
    fileurl = m_url + path.Substring(root.Length).Replace(@"\", "/") + "/" + f.Name;

    System.IO.StreamReader reader = System.IO.File.OpenText (path + @"\" + f.Name);
    fileContents = reader.ReadToEnd();
    reader.Close(); // now use the fileContents to build the catalog...
```

Listing 3 - Opening the files

A quick Google helped find a solution to (b).

```
// ### Grab the <TITLE> ###
Match TitleMatch = Regex.Match(fileContents, "<title>([^\<]*)</title>", RegexOptions.IgnoreCase | RegexOptions.Multiline );
```

```

filetitle = TitleMatch.Groups[1].Value;
// ### Parse out META data ###
Match DescriptionMatch = Regex.Match( fileContents, "<META NAME=\"DESCRIPTION\" CONTENT=\"([^\<]*)\">",
RegexOptions.IgnoreCase | RegexOptions.Multiline );
filedesc = DescriptionMatch.Groups[1].Value;
// ### Get the file SIZE ###
filesize = fileContents.Length;
// ### Now remove HTML, convert to array, clean up words and index them ###
fileContents = stripHtml (fileContents);

Regex r = new Regex(@"\s+"); // remove all whitespace
string wordsOnly = stripHtml(fileContents);

// ### If no META DESC, grab start of file text ###
if (null==filedesc || String.Empty==filedesc) {
if (wordsOnly.Length > 350)
filedesc = wordsOnly.Substring(0, 350);
else if (wordsOnly.Length > 100)
filedesc = wordsOnly.Substring(0, 100);
else
filedesc = wordsOnly; // file is only short!
}

```

Listing 4 - Massage the file contents

And finally (c) involved a very simple Regular Expression or two, and suddenly we have the document as an Array of words, ready for processing!

```

protected string stripHtml(string strHtml) {
//Strips the HTML tags from strHTML
System.Text.RegularExpressions.Regex objRegExp
= new System.Text.RegularExpressions.Regex("<(.|\n)+?>");

// Replace all tags with a space, otherwise words either side
// of a tag might be concatenated
string strOutput = objRegExp.Replace(strHtml, " ");

// Replace all < and > with &lt; and &gt;
strOutput = strOutput.Replace("<", "&lt;");
strOutput = strOutput.Replace(">", "&gt;");

return strOutput;
}

```

Listing 5 - Remove HTML

and

```

Regex r = new Regex(@"\s+"); // remove all whitespace
wordsOnly = r.Replace(wordsOnly, " "); // compress all whitespace to one space
string [] wordsOnlyA = wordsOnly.Split(' '); // results in an array of words

```

Listing 6 - Remove unnecessary whitespace

To recap - we have the code that, given a starting directory, will crawl through it (and its subdirectories), opening each HTML file, removing the HTML tags and putting the words into an array of strings.

Now that we can parse each document into words, we can populate our Catalog!

## Build the Catalog

All the hard work has been done in parsing the file - building the catalog is as simple as adding objects to

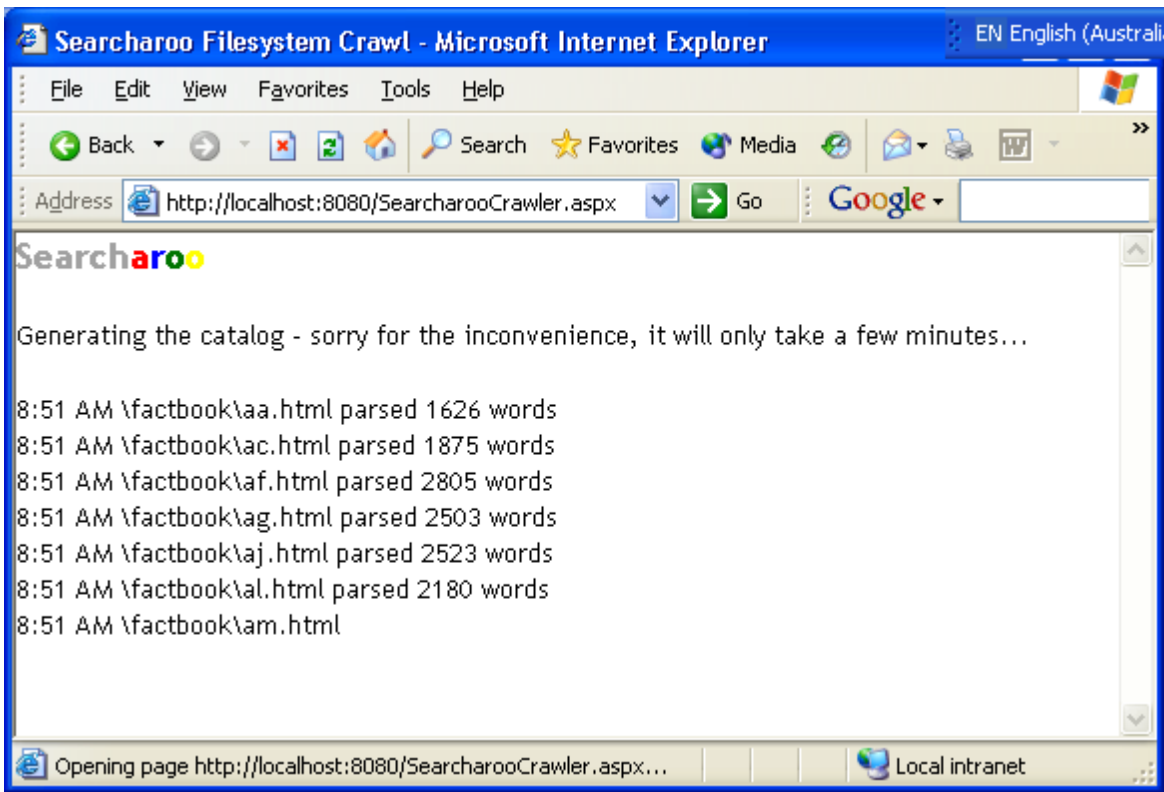
```

// ### Loop through words in the file ###
int i = 0; // Position of the word in the file (starts at zero)
string key = ""; // the 'word' itself
// Now loop through the words and add to the catalog
foreach (string word in wordsOnlyA) {
key = word.Trim(' ', '?', '\'', ':', '\\", ';', ':', ',', '(', ')').ToLower();
m_catalog.Add (key, infile, i);
i++;
} // foreach word in the file

```

Listing 7 - Add words to the catalog

As each file is processed a line is written to the browser to indicate the catalog build progress, showing the File.Url and the number of words parsed.



Screenshot 2 - Processing the CIA World FactBook - it contains 40,056 words according to our code.

After the last file is processed, the Catalog object is added to the Application Cache object, and is ready for searching!

## Build the Search

The finished Catalog now contains a collection of Words, and each Word object has a collection of the Files it was found in. The Search method of the Catalog takes a single word as the search parameter, and returns the Hashtable of File objects where that Word was found. The returned Hashtable keys are File objects and the values are the rank (ie. count of the number of times the words appear).

All the hard work has been done in parsing the file - building the catalog is as simple as adding objects to

```

/// <summary>Returns all the Files which contain the searchWord</summary>
/// <returns>Hashtable </returns>
public Hashtable Search (string searchWord) {
    // apply the same 'trim' as when we're building the catalog
    searchWord = searchWord.Trim('?', '\'', ',', '\'', ':', ';', ':', ',', '(', ')').ToLower();
    Hashtable retval = null;
    if (index.ContainsKey (searchWord) ) { // does all the work !!!
        Word thematch = (Word)index[searchWord];
        retval = thematch.InFiles(); // return the collection of File objects
    }
    return retval;
}

```

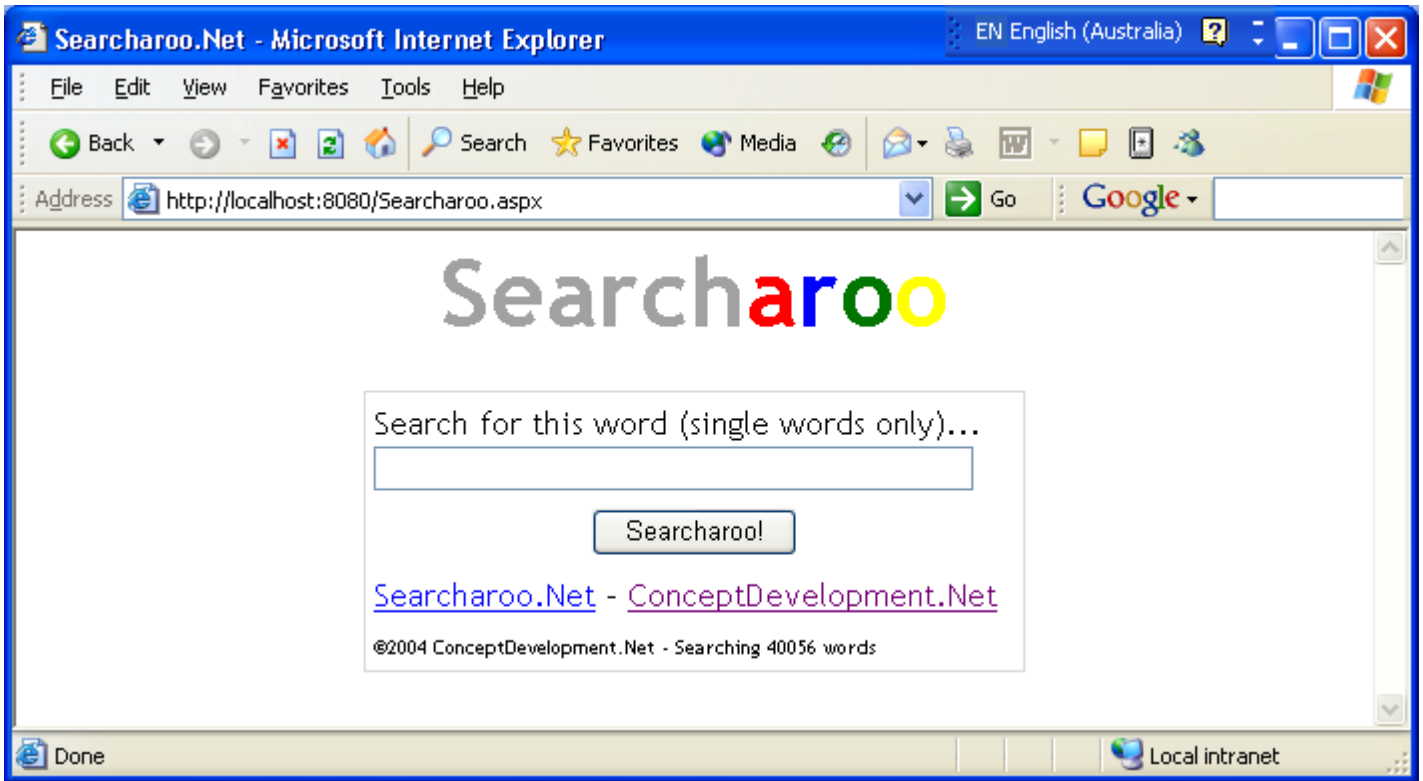
Listing 8 - the Search method

The key point is how simple the Search method can be, because of the amount of work performed during the cataloging.

Obviously there are a number of enhancements we could make here, starting with multiple word searches (finding the intersection of the File Hashtables for each Word), implementing Boolean searches, fuzzy matches (or matching word stems/roots)... the list is (almost) endless, but beyond the scope of this article.

## Build the Results [Searcharoo.aspx]

Searcharoo.aspx initially displays an HTML form to allow the user to enter the search term.



Screenshot 3 - Enter the search term

When this form is submitted, we look for the Word in the index Hashtable using the `ContainsKey()` method, and rely on the efficiency of the .NET Framework's searching a collection for an object using its `HashCode`. The `Hashtable.ContainsKey()` method is actually doing the search for us.

The `Catalog.Search()` method returns a Hashtable containing the matching File objects, so all we have to do is display the them in HTML format!

The display process has been broken into a few steps below:

Firstly, we call the `Search` method to get the result Hashtable. If the result is null skip to Listing 13 because there were no matches, otherwise we have a little more work to do...

```
// Do the search
Hashtable searchResultsArray = m_catalog.Search(searchterm);
// Format the results
if (null != searchResultsArray) {
```

Listing 9 - The actual search is the easy bit

The Dictionary returned from the `Search()` method has File objects as the key and the page rank as the value. The problem is they are not in any particular order!

To access these objects in the foreach loop, we need to cast the key object to a File and the value object to int.

Firstly, we call the `Search` method to get the result Hashtable. If the result is null skip to the end because there were no matches, otherwise we have a little more work to do...

```
// intermediate data-structure for 'ranked' result HTML
SortedList output = new SortedList(searchResultsArray.Count); // empty sorted list
DictionaryEntry fo;
File infile;
string result="";
// build each result row
foreach (object foundInFile in searchResultsArray) {
    // build the HTML output in the sorted list, so the 'unsorted'
    // searchResults are 'sorted' as they're added to the SortedList
    fo = (DictionaryEntry)foundInFile;

    infile = (File)fo.Key;
    int rank = (int)fo.Value;
```

Listing 10 - Processing the results

Firstly, we call the Search method to get the result Hashtable. If the result is null, game over, otherwise we have a little more work to do.

```
// Create the formatted output HTML
result = ("<a href=" + infile.Url + ">");
result += ("<b>" + infile.Title + "</b></a>");
result += (" <a href=" + infile.Url + " target=\"_TOP\" ");
result += ("title=\"open in new window\" style=\"font-size:xx-small\">&uarr; </a>");
result += (" <font color=gray>(" + rank + ")</font>");
result += ("<br>" + infile.Description + "... " );
result += ("<br><font color=green>" + infile.Url + " - " + infile.Size);
result += ("bytes</font> <font color=gray>- " + infile.CrawledDate + "</font><p>" );
```

Listing 11 - Pure formatting

Before we can output the results, we need to get them in some order. We'll use a SortedList and add the HTML result string to it using the page rank as the key. If there is already an result with the same rank, we'll concatenate the results together (they'll appear one after the other).

```
int sortrank = (rank * -1); // multiply by -1 so larger score goes to the top
if (output.Contains(sortrank) ) { // rank exists; concatenate same-rank output strings
    output[sortrank] = ((string)output[sortrank]) + result;
} else {
    output.Add(sortrank, result);
}
result = ""; // clear string for next loop
```

Listing 12 - Sorting the results by rank

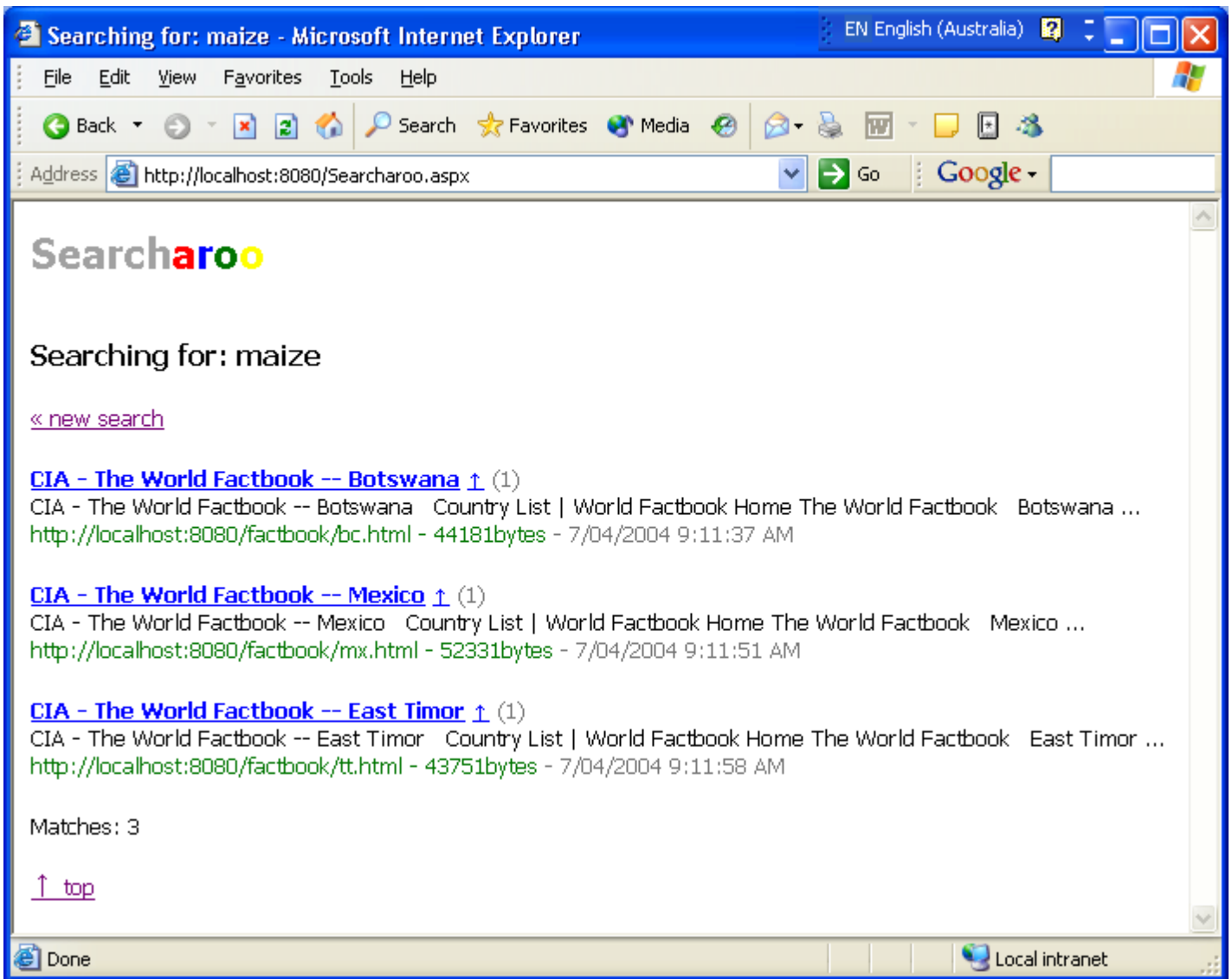
To make sure the highest rank appears at the top of the list, the rank is multiplied by -1!

Now all we have to do is Response.Write the SortedList, string by string, followed by the number of matches.

```
// Now output to the HTML Response
foreach (object rows in output) { // Already sorted!
    Response.Write ( (string)((DictionaryEntry)rows).Value );
}
Response.Write("<p>Matches: " + searchResultsArray.Count);
} else {
    Response.Write("<p>Matches: 0");
}
Response.Write ("<p><a href=#top> ↑ top</a>");
Response.End(); // Stop here
```

Listing 13 - Output the results

The output should look familiar to any web search engine user. We've implemented a simple ranking mechanism (a word count, shown in parentheses after the Title/Url) however it doesn't support paging.



Screenshot 4 - Search results contain a familiar amount of information, and the word-count-rank value. Clicking a link opens the local copy of the HTML file (the ↑ opens in a new window).

## Using the sample code

The goal of this article was to build a simple search engine that you can install just by placing some files on your website; so you can copy Searcharoo.cs, SearcharooSpider.aspx and Searcharoo.aspx to your web root and away you go!

However that means you accept all the default settings, such as only searching .HTML files, and the search starting from the location of the Searcharoo files.

To change those defaults you need to add some settings to web.config:

```
<appSettings>
  <!--physical location of files-->
  <add key="Searcharoo_PhysicalPath" value="c:\inetpub\wwwroot\" />
  <!--base Url to build links-->
  <add key="Searcharoo_VirtualRoot" value="http://localhost/" />
  <!--allowed file extension-->
  <add key="Searcharoo_FileFilter" value="*.html"/>
</appSettings>
```

Listing 14 - web.config

Then simply navigate to <http://localhost/Searcharoo.aspx> (or wherever you put the Searcharoo files) and it will build the catalog for the first time.

If your application re-starts for any reason (ie. You compile code into the /bin/ folder, or change web.config settings) the catalog will need to be rebuilt - the next user who performs a search will trigger the catalog build. This

is accomplished by checking if the Cache contains a valid Catalog and if not using Server.Transfer to start the crawler.

## Future

In the real world, most ASP.NET websites probably have more than just HTML pages, including links to DOC, PDF or other external files and ASPX dynamic/database-generated pages.

The other issue you might have is storing a large blob of data in your Application Cache. For most websites the size of this object will be manageable - but if you've got a lot of content you might not want that in memory all the time. The good news is the code above can be easily extended to cope with these additional scenarios (including spidering web links, and using a database to store the catalog)... check back for future articles.

## Postscript : What about code-behind and Visual-Studio.NET?

You'll notice the two ASPX pages use the src="Searcharoo.cs" @Page attribute to share the common object model without compiling to an assembly, with the page-specific 'inline' using <script runat="server"> tags (similar to ASP3.0).

The advantage of this approach is that you can place these three files in any ASP.NET website and they'll 'just work'. There are no other dependencies (although they work better if you set some web.config settings) and no DLLs to worry about.

However, this also means these pages can't be edited in Visual-Studio.NET, because it does not support the @Page src="" attribute, instead preferring the codebehind="" attribute coupled with CS files compiled to the /bin/ directory. To get these pages working in VisualStudio.NET you'll have to setup a Project and add the CS file and the two ASPX files (you can move the <script> code into the code-behind if you like) then compile.

## Links

[The CIA World Factbook website](http://www.cia.gov/cia/publications/factbook/index.html)

[<http://www.cia.gov/cia/publications/factbook/index.html>](http://www.cia.gov/cia/publications/factbook/index.html)

[Web Forms Code Model](http://msdn.microsoft.com/library/en-us/vbcon/html/vbconWebFormsCodeModel.asp)

[<http://msdn.microsoft.com/library/en-us/vbcon/html/vbconWebFormsCodeModel.asp>](http://msdn.microsoft.com/library/en-us/vbcon/html/vbconWebFormsCodeModel.asp) (about CodeBehind and Src)

[Working with Single-File Web Forms Pages in Visual Studio .NET](http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vstchWorkingWithSingle-FileWebFormsPagesInVisualStudio.asp)

[<http://msdn.microsoft.com/library/en-us/dv\\_vstechart/html/vstchWorkingWithSingle-FileWebFormsPagesInVisualStudio.asp>](http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vstchWorkingWithSingle-FileWebFormsPagesInVisualStudio.asp) (to help those wanting to use VisualStudio)

[Building a Better Binary Search Tree](http://msdn.microsoft.com/library/en-us/dv_vstechart/html/datastructures_guide4.asp)

[<http://msdn.microsoft.com/library/en-us/dv\\_vstechart/html/datastructures\\_guide4.asp>](http://msdn.microsoft.com/library/en-us/dv_vstechart/html/datastructures_guide4.asp) (for version 2!)