Combined Searcher

by Adam Herrick (07/12/2020)

### Introduction

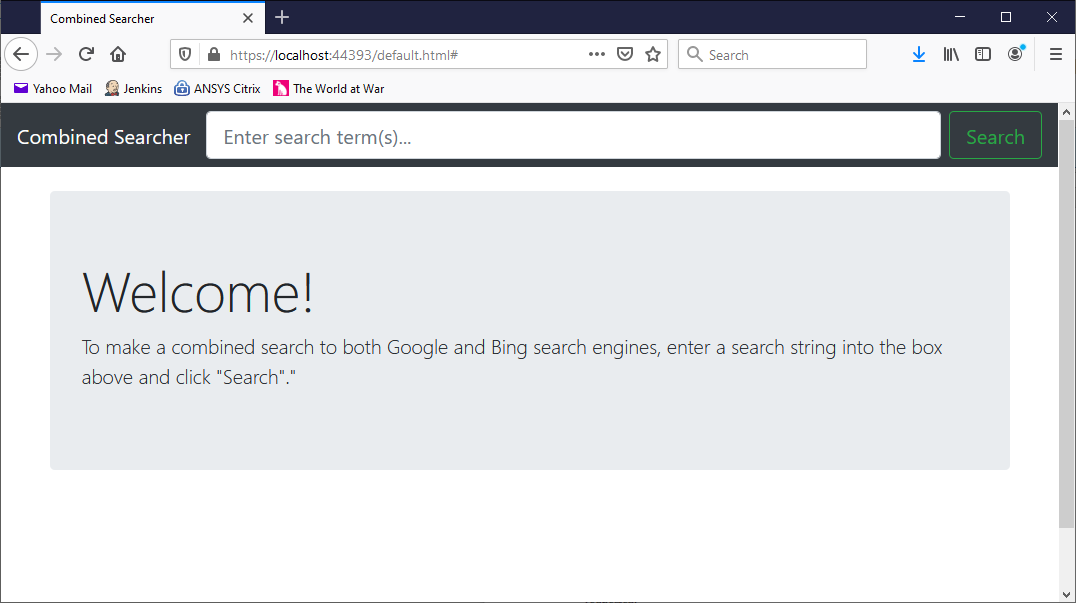
The Combined Searcher application satisfies the requirements of the Blackdot Solutions programming test by making concurrent HTTP requests to the Google and Bing search engines, reading (scraping) the results returned from those pages and amalgamating them into a single result set as required by the user story provided.

It achieves this objective using a variety of code features in C# and JavaScript, including

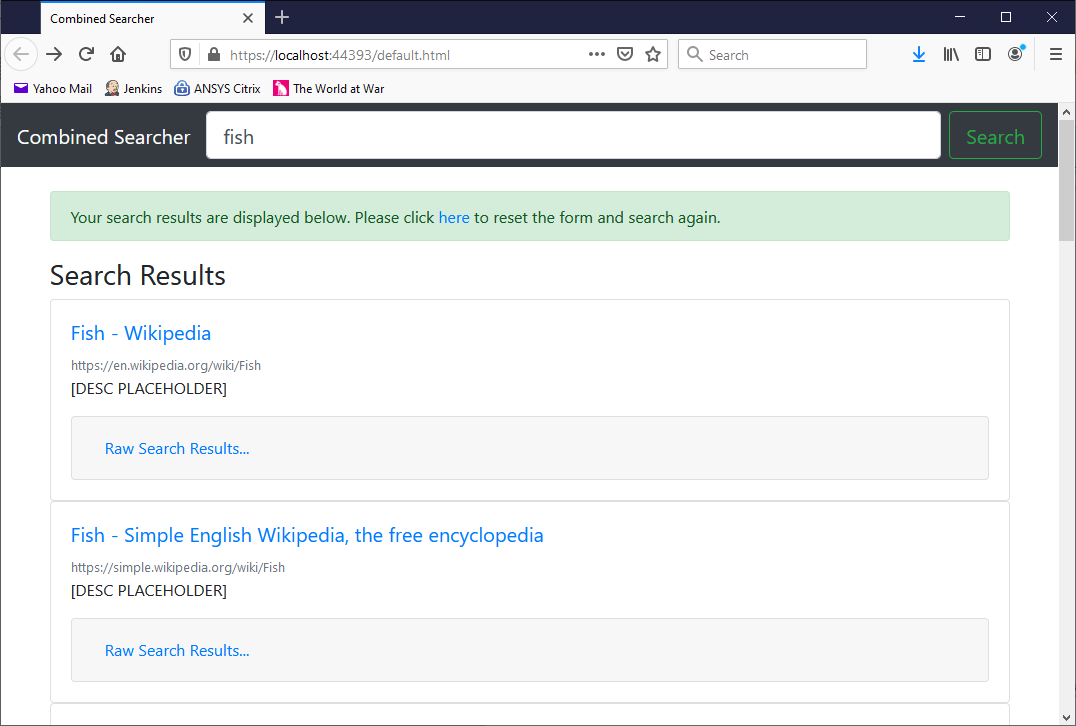
* The HtmlAgilityPack nuget for the formatting / parsing of HTML documents as objects.
* A stripped down version of a generic HTTP client wrapper which I’ve developed in my own projects and used widely in various forms throughout my career.
* An AspNetCore Web API which provides an endpoint via which the search function is requested.
* A Bootstrap / jQuery UI which allows the submission of searches / display of result.

### Running the Solution

Open the **Bds.TechTest.sln** file in Visual Studio and hit F5. You should be presented with a screen which looks like this:



Enter a search term in the box at the top and click “Search”. A results screen similar to the following should be displayed:



### Approach

It has been my experience thus far that 95% of programming problems are structural. That is to say that the vast majority of problems are not solved by writing brand new code (although of course, some are), but by adapting existing code to a new pattern, extending it, modifying it, refactoring it, and so on.

Because of this experience, I always try, as I have tried here, to be rigorous and thoughtful about the structure of the code I produce, to ensure that it is testable, maintainable and easily extensible, such that improvements, corrections and additions can be made without the need for significant refactoring, which increases risk and delays delivery. (That said, refactoring bad code is one of the things I most enjoy!)

Broadly therefore, I have applied the single responsibility principal and the rules of encapsulation to this solution, although there are places where I would have liked, if I’d had more time, to leave “hooks” for future development. These are discussed in “Limitations” below.

Furthermore, I have employed a generic approach to the handling of search results: there is no reason, for example, that the orchestration of a request to any particular search engine should be affected by the exact content of the search result it returns. Thus, classes like the SearchEngineOrchestrator<T> should be able to submit a GET request to the relevant endpoint and apply the appropriate result scraper instance without having to know what that instance is or to specialize on that basis.

Because of these features, tasks like the addition of a new search engine, for example, should be straightforward, where perhaps a new ISearchEngineDefinition<T> instance, ISearchResultScraper<T> instance and perhaps another result class would be the only requirements. Alternatively, the inclusion of the description of the search result should only be a “plumbing” problem rather than any significant piece of work.

### Limitations

Due to significant work and family commitments this week, I only managed to get started on the task late on Thursday evening. As such, there are some things which I would have liked to include, but have not had time for. These include:

* Abstraction of the result matching code. At present a simple algorithm which orders the results by calculating the average rank for all search engines returning it, for any given result, is applied. However, there are certainly more sophisticated approaches which could be written and the conversion of the static method ExtractAndCombineResults() in the controller class behind an interface would be a logical development.
* Either (or both) of the ISearchEngineDefinition<T> or ISearchResultScraper<T> classes could be extended to describe how multiple pages of results, up to a predetermined limit, could be paged from each search engine and accumulated. The most likely starting point would be to include support for more tokens in the ISearchEngineDefinition<T>.UriFormat property whereby a page index or similar could be provided in the URL. Both Google and Bing appear to support this approach.
* At present there is only one implementation of the IResponseContentConverter<T> class, and the instance is defined as a static readonly in the appropriate place, but this might reasonably become a constructor parameter if other means of reading responses became valuable. For example, were the stipulation not to use provider APIs not included, it would be perfectly possible to adapt this pattern to provide other forms of data (JSON, perhaps) and feed that into a ISearchResultScraper<T> class handling that form of input, without changing the overall pattern of the solution.
* The provision of the link description scraped from the search engines’ raw results would be valuable to the user and should be elementary to implement. I have indicated my wish to do so with a placeholder but I have run out of time!
* The top bar of the UI is not a form and thus we don’t get to hit Enter to trigger the search. This should be straightforward, but again, time is limited.
* The CombinedSearchController should be tested and ought, probably, to use some form of DI, rather than creating all of its own resources. The latter would be more justified in a larger solution.