Search Engine for akos.ma

Adrian Kosmaczewski

2020-12-04

Adding a search engine to this website was a nice little weekend project.

I build the static HTML website you're reading now using Hugo. Of course, Hugo being what it is, it is clear that there is no server-side process to handle a dynamic request, like a search query would need to be processed. The whole website consists of rather inert HTML files.

This blog post will describe the idea and the process that led to the search box that you can see in the menu of this website.

Design

I host this website in Hostpoint, a well-known hosting company from Zürich, Switzerland. They offer the usual "PHP + MySQL" hosting combo, coupled with a lot of goodies, like SSH access and cron jobs, the latest PHP version, all nicely located within FreeBSD servers. I like them very much.

(Of course I should be hosting these pages in APPUiO instead, but let's say for historical reasons I'm still with Hostpoint. I will move to APPUiO at some point.)

All of this meant that the easiest path to have a search engine in this website was to create a PHP 7.4 application returning search results, somehow. After a bit of research I found TNTSearch, a fully featured full text search engine written in PHP.

And that was exactly what I needed.

Indexing the Contents

The following challenge was to create a small application that uses TNTSearch to create a suitable index out of the HTML pages generated by Hugo.

TNTSearch stores the index in a PDO-compatible database; usually MySQL or SQLite. I chose the latter since it is extremely simple to deploy, and I could rebuild and `scp` the new index after publishing a new article every weekend.

I created a new PHP project using Composer. I specified TNTSearch as the only requirement, and then created a small PHP script to generate the index, to be used in the command line.

```php
<?php
$tnt = new TeamTNT\TNTSearch\TNTSearch;
$config = [
    'storage'   => __DIR__,
    'driver'    => 'filesystem',
    'location'  => $docs_root,
    'extension' => 'html',
    'exclude'   => $files_to_exclude
];

// Perform the indexing
$tnt->loadConfig($config);
$indexer = $tnt->createIndex('search/index.db');
$indexer->run();
```

Run this script using the usual `php create_index.php` and you end up with a SQLite 3 file called `index.db`.

Making the Search Results Nicer

A quick analysis of the final `index.db` file (for example using DB Browser for SQLite) shows that the indexing process only stores list of words, and their associations to a certain HTML page, referenced by its absolute path. In order to have nice search results, we need to have a way to show the user at least the title and maybe even a snippet of each search result.

For that, I switched to Python, because of Beautiful Soup, a fantastic library that reads HTML and offers a nice API to find out various pieces of information.

So here is how I used it:

```
from bs4 import BeautifulSoup
import sqlite3

def page_title(path):
    f = open(path, "r")
    html = f.read()
    soup = BeautifulSoup(html, 'html.parser')
    title = soup.find('title')
    return title.string.replace(' | akos.ma', '')

def read_index(index):
    conn = sqlite3.connect(index)
    c = conn.cursor()
    data = []
    for row in c.execute('SELECT id, path FROM filemap'):
        path = row[1]
        title = page_title(path)
        snippet = page_first_paragraph(path)
        data.append({
            'path': path,
            'title': title,
            'snippet': snippet
        })
    conn.close()
    return data

def write_db(filename, data):
    conn = sqlite3.connect(filename)
    c = conn.cursor()
    c.execute('CREATE TABLE IF NOT EXISTS files(id INTEGER PRIMARY KEY, path TEXT, title TE
    c.execute('CREATE INDEX paths ON files(path);')
    for row in data:
        params = (row['path'], row['title'], row['snippet'],)
        c.execute('INSERT INTO files (path, title, snippet) VALUES (?, ?, ?);', params)
    conn.commit()
    conn.close()

data = read_index('search/index.db')
write_db('search/files.db', data)
```

And now we have two SQLite files: `index.db` with the TNTSearch index, and `files.db` with a list of files including their title, a snippet, and an index for querying stuff using the file path.

Searching

And now we can write the actual PHP application which, once installed in our server, will return a JSON string with the search results:

```php
<?php
$query = $_GET['q'] ?? '';
$response = [
    'results' => []
];
if ($query === '')
{
    echo(json_encode($response));
}
else
{
    $tnt = new TNTSearch;
    $tnt->loadConfig([
        'storage'   => __DIR__,
        'driver'    => 'filesystem',
    ]);
    $tnt->selectIndex('index.db');
    $tnt->asYouType = true;

    $search_results = $tnt->search($query, 10);

    $files_db_path = realpath(__DIR__ . '/files.db');
    $access = [PDO::SQLITE_ATTR_OPEN_FLAGS => PDO::SQLITE_OPEN_READONLY];
    $db = new PDO('sqlite:' . $files_db_path, null, null, $access);
    $select = $db->prepare('SELECT title, snippet FROM files WHERE path = :path;');

    foreach ($search_results as $key => $value)
    {
        $path = $value['path'];
        $select->bindParam(':path', $path, PDO::PARAM_STR);
        $select->execute();
        $result = $select->fetch();
        $title = $result['title'];
        $snippet = $result['snippet'];
        $response['results'][] = [
            'path' => short_version($path),
            'title' => $title,
            'snippet' => $snippet
        ];
    }
    header('Content-Type: application/json');
    echo(json_encode($response));
}
```

This script provides the backend functionality we need in our website.

User Interface

And now for the frontend part. This site uses a modified version of the Noteworthy theme for Hugo. Apart from adding an HTML `<input>` field to the page and some CSS, the most interesting part is, of course, the JavaScript that handles the searches, itself based on the Vanilla JS framework, the best there is:

```
; (function () {
  'use strict'

  // Creates the DOM structure of a single search result item
  // The website variable contains the current domain where this code is running.
  function createSearchResultsDiv (item, website) {
    var searchParagraph = document.createElement('p')
    searchParagraph.className = 'search-paragraph'

    var searchEntry = document.createElement('a')
    searchEntry.innerText = item.title
    searchEntry.href = item.path
    searchEntry.className = 'search-entry'
    searchParagraph.appendChild(searchEntry)

    // ...

    var searchDiv = document.createElement('div')
    searchDiv.className = 'search-div paragraph'
    searchDiv.onclick = function () {
      window.location.href = item.path
    }
    searchDiv.appendChild(searchParagraph)
    return searchDiv
  }

  // Builds the HTML structure of the list of search results
  // The results variable is an array of objects with 'name', 'href' and 'excerpt' keys.
  // The query variable is a string entered by the user.
  function display (results, query) {
    if (isEmptyOrBlank(query)) {
      // Display the original page in lieu of the search results if not done yet
      if (!mainArticle.parentNode) {
        body.replaceChild(mainArticle, searchArticle)
      }
      return
    }
    // Rebuild the contents of the "search results" page
    removeAllChildren(searchArticle)
    var searchTitle = document.createElement('h1')
    searchTitle.className = 'page'
    searchArticle.appendChild(searchTitle)
    searchTitle.innerText = 'Search Results for "' + query.trim() + '"'
    if (results.length === 0) {
      var searchResult = document.createElement('p')
      searchResult.innerText = 'No results found.'
      searchArticle.appendChild(searchResult)
    } else {
      results.forEach(function (item, idx) {
        var searchDiv = createSearchResultsDiv(item, website)
        searchArticle.appendChild(searchDiv)
      })
    }
    // Replace the current page with a "search results" page if not done yet
    if (!searchArticle.parentNode) {
      body.replaceChild(searchArticle, mainArticle)
    }
  }

  // Performs the actual search
  function search (query, callback) {
    if (isEmptyOrBlank(query)) {
      // Display the original page in lieu of the search results if not done yet
      if (!mainArticle.parentNode) {
        body.replaceChild(mainArticle, searchArticle)
      }
      return
    }
    var XMLHttpRequest = window.XMLHttpRequest
    var xmlhttp = new XMLHttpRequest()

    xmlhttp.onreadystatechange = function () {
      if (xmlhttp.readyState === XMLHttpRequest.DONE) {
        if (xmlhttp.status === 200) {
          callback(JSON.parse(xmlhttp.responseText)['results'])
        } else {
          console.log('Status received: ' + xmlhttp.status)
        }
      }
    }
```

```
    var url = '/search/?q=' + encodeURIComponent(query)
    xmlhttp.open('GET', url, true)
    xmlhttp.send()
  }

  var searchInput = document.querySelector('#search-input')
  // ...

  var timeout = null
  function triggerSearch() {
    if (timeout) clearTimeout(timeout)
    timeout = setTimeout(() => {
      var query = searchInput.value
      search(query, function (results) {
        display(results, query)
      })
    }, 500)
  }

  // Event to be fired everytime the user presses a key
  searchInput.onkeyup = function () {
    triggerSearch()
  }
})()
```

So every time a user types on the field, we wait for half a second after the last character, and send the search to the server. The results are embedded on the DOM of the same page, and if the user cleans the search field, the original page is displayed instead.

I borrowed this client-side code from the VSHN Antora UI Default project, used to build various Antora documentation websites for VSHN.

Conclusion

It was a nice fun project. I reused stuff I already had done in the past; not only the client-side code, but also the approach of a two-stage search engine: first, the creation of the index, and then its use.

This is the approach I used for the search functionality in the Antora websites we have in VSHN, and you can see it in the Antora indexer CLI project. In that case though, it is written in TypeScript, and uses Lunr.js as engine. The index is then used by the embedded search engine, deployed as a sidecar pod in the Kubernetes deployment.

I have added a few `Makefile` here and there, so now I can simply `make index` and `make deploy` every time I update this website with new content. The search results are snappy, fast, and relevant. And everybody is happy.

For the future I plan on changing the backend of the search index, and move it to MySQL instead. For the moment the current approach works, and anyway, I do not have huge amounts of traffic in this website (at least not so far).

Bonustrack: Search Engine Libraries

As a bonus, please find below a list of search engine libraries I have found online while searching for options. Maybe you will find this list useful for your own needs.

Rust

- Tantivy is a full-text search engine library inspired by Apache Lucene and written in Rust.
- Toshi is meant to be a full-text search engine similar to Elasticsearch. Toshi strives to be to Elasticsearch what Tantivy is to Lucene. Written in Rust.
- Sonic is a fast, lightweight and schema-less search backend, written in Rust.
- Bayard is a full text search and indexing server, written in Rust, built on top of Tantivy.
- MeiliSearch Ultra relevant, instant, and typo-tolerant full-text search API.

Go

- Bleve is a modern text indexing library in Go.
- Blast is a full text search and indexing server, written in Go, built on top of Bleve.
- Riot is Go Open Source, Distributed, Simple and efficient full text search engine.

Python

- Whoosh is a fast, pure Python search engine library.

JavaScript

- Lunr, whose description says it all: "A bit like Solr, but much smaller and not as bright". I used this library in the search engine of VSHN Antora websites, such as our Handbook or Project Syn and it also uses an index, but in this case composed of JSON files.

Others

- Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java.

- Apache Solr is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more.
- Elasticsearch is a distributed, open source search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured.