Article
Browse Code
View Stats
Comments
Posted 1 Aug 2019

Tagged as
SQL
MySQL
Amazon
AWS
Caching

Stats
1.1K views
27 downloads
2 bookmarked

Full-Text Search with dtSearch and AWS Aurora

Mike V Baker
1 Aug 2019 CPOL

In this article, we'll extend the dtSearch Engine-based example to use Amazon's Aurora storage service, which is a hosted MySQL solution available through AWS.

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.

Download source - 1.7 MB

In a previous article, we demonstrated how to harness the power of the dtSearch Engine to index and search Microsoft Office documents with the worldwide accessibility and storage capacity of Amazon Web Services (AWS). In that example, we used EBS volumes to store our source documents and search index. It's easy, however, to extend the same indexing and search features to other cloud storage services.

In this article, we'll extend the dtSearch Engine-based example to use Amazon's Aurora storage service, which is a hosted MySQL solution available through AWS. We build on the index and search example using EC2 and attached EBS volumes that we created in the article "Using dtSearch on Amazon Web Services with EC2 & EBS," so we recommend working through that example first.

MySQL is great at many things, but it's not great at full-text search. This makes the dtSearch Engine the perfect complement to Aurora. We'll briefly discuss setting up the Aurora database and other services from AWS, then we'll look at the implementation of two applications. One reads documents, inserts them into the Aurora database, then creates the index. The other allows end users to search the index.

Project Prerequisites

Setting up the project, we'll use the EC2 instance created for the previous article. We'll also set up an Aurora MySQL database used for storing documents and index data.

This article assumes we already have an AWS account, so start by logging into the AWS Management Console. Once we're in the console we can see the list of services available with the more recently used services at the top for easy access.

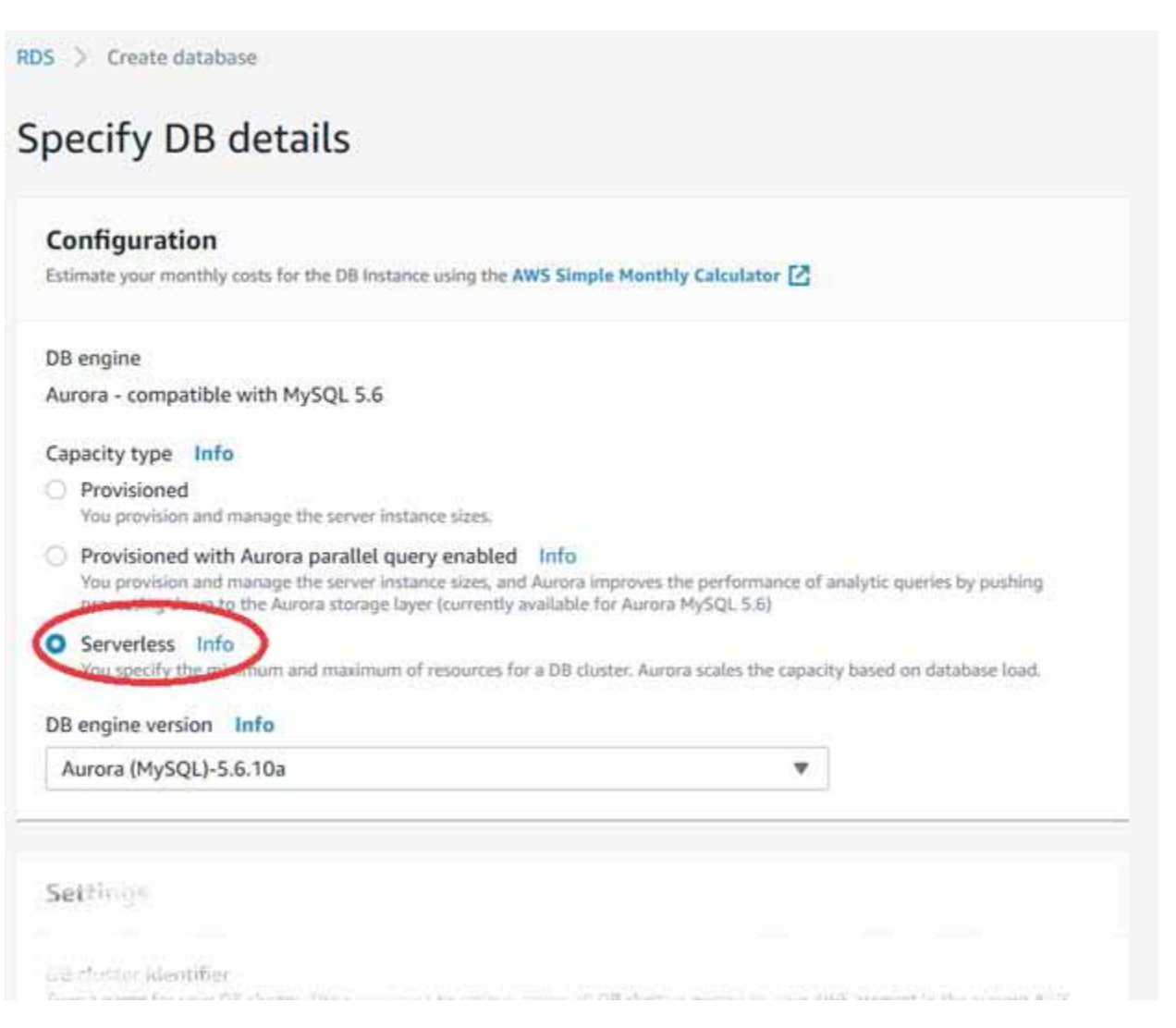
Creating the Aurora Database

We're going to start by setting up the Aurora database. You can find documentation in the Amazon Aurora User Guide.

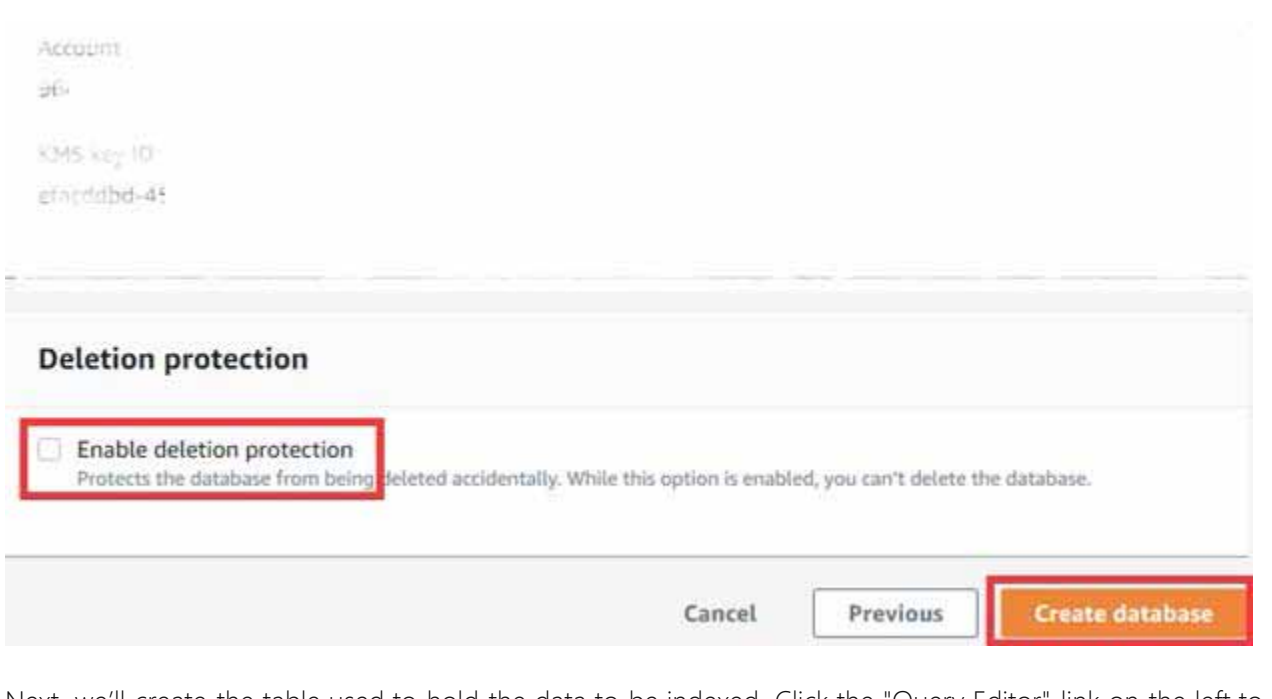
When you reach the AWS Management Console, click on "RDS".

Click on "Create Database", make sure "Aurora (MySQL)" is selected as the DB engine, and click "Next". Run through the steps to create the database. We selected a "Serverless" Capacity type and used "dtsearchtest" for the ID.

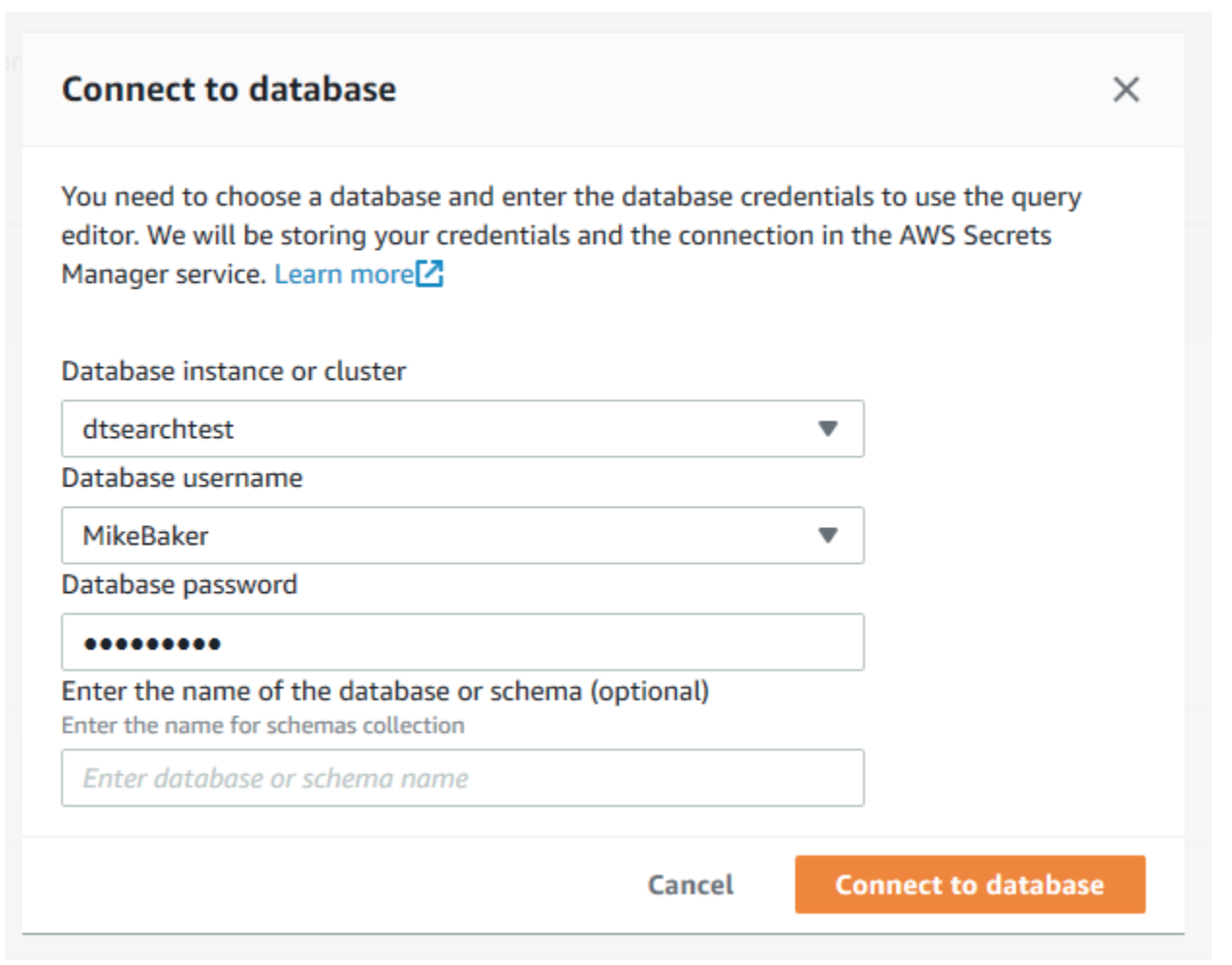
Pay attention to the security group. We need to add the security group used by the EC2 instance from the previous article to the security group used for the Aurora database, so applications running on the EC2 instance can reach the database.



Clear the checkmark on "Enable deletion protection" so we can delete the database when we're finished using it. Then click "Create database".



Next, we'll create the table used to hold the data to be indexed. Click the "Query Editor" link on the left to bring up the "Connect to Database" dialog. Only databases that are set up in the Serverless environment work with the Query Editor.



After we connect we'll see a window where we can enter SQL statements and execute them on the database. The SQL statement to create this database is:

USE dtSearchTest;

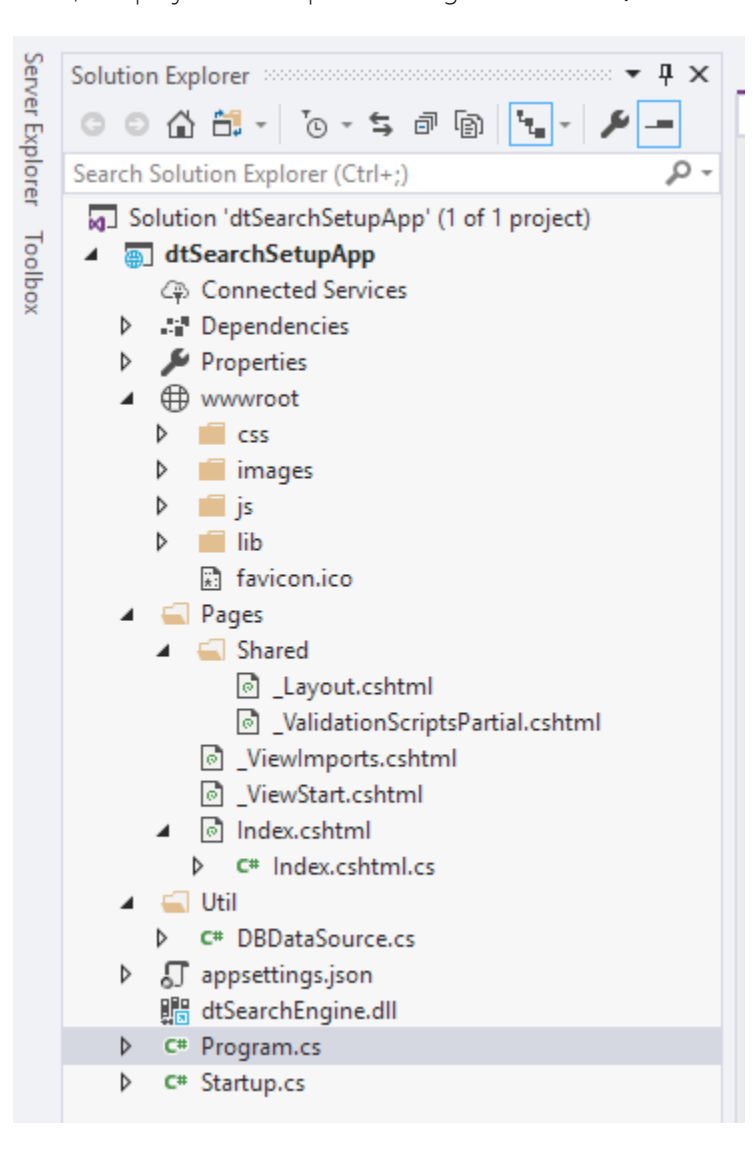
```
USE dtSearchTest;
CREATE TABLE ShakespeareDoc ( doc_id INT AUTO_INCREMENT,
                              doc_name VARCHAR(255),
                              doc_file VARCHAR(2048),
                              doc_content MEDIUMTEXT,
                              PRIMARY KEY (doc_id) );
```

This statement specifies the `doc_id` (which is an auto-id field), a friendly name, and the filename referring back to the source data. `doc_content` contains the actual contents of the file.

Server Setup App: dtSearchSetupApp

We created two simple application projects. We'll walk through some details of the applications here. Download the source code for the project to get started.

Let's look at the first of the two applications, `dtSearchSetupApp`. As with the console app from the previous article, the project is set up in a sibling folder to the */lib* folder that contains *dtSearchEngine.dll*.

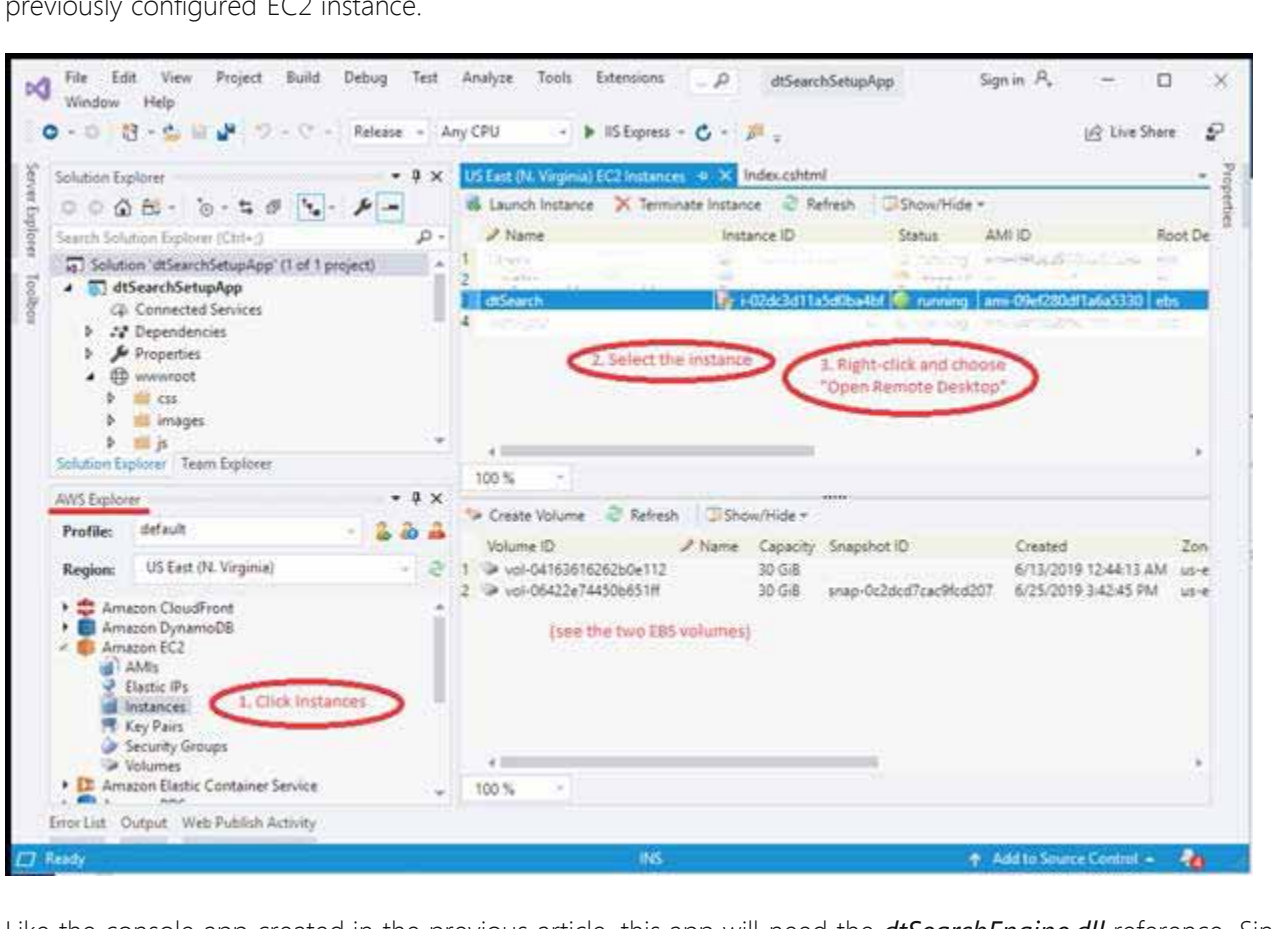


We created a .NET Core Web Application project with the default settings, but without the HTTPS option. After Visual Studio created the project, we removed all pages except for "Index" and the partial for the cookie policy. This left all the code in place for button handlers and cross-site-forgery protection. We also opened the "_Layout" partial and removed the nav bar, the reference to the cookie policy partial, and basically any content other than code.

The application will need a connector to work with MySQL. We chose the MySql.Data connector NuGet package from NuGet. Documentation for using the connector is available on the MySQL Connector/NET site.

We also added the AWS Toolkit for Visual Studio, which lets us browse through the services attached to an AWS account. It's particularly useful for connecting to EC2 instances. Install the toolkit through the Visual Studio "Extensions > Manage Extensions" menu option.

Open AWS Explorer from the View menu. Once configured, click on "Amazon EC2 > Instances" and connect to the previously configured EC2 instance.



Like the console app created in the previous article, this app will need the ***dtSearchEngine.dll*** reference. Since .NET Core is cross-platform, we may want to deploy this on a system other than Windows Server. To make the reference cross-platform, modify the .csproj file directly and paste in the following lines. (For more information see Native Libraries in the dtSearch .NET documentation.)

```
<ItemGroup
Condition="'$([System.Runtime.InteropServices.RuntimeInformation]::IsOSPlatform($([System.Runtime.InteropServices.OSPlatform]::Linux)))' == 'true'">
  <Content Include="..\..\lib\engine\linux\x64\libdtSearchEngine.so"
           Link="libdtSearchEngine.so">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </Content>
</ItemGroup>

<ItemGroup
Condition="'$([System.Runtime.InteropServices.RuntimeInformation]::IsOSPlatform($([System.Runtime.InteropServices.OSPlatform]::OSX)))' == 'true'">
  <Content Include="..\..\lib\engine\macos\x64\libdtSearchEngine.dylib"
           Link="libdtSearchEngine.dylib">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </Content>
</ItemGroup>

<ItemGroup Condition="'$(OS)' == 'Windows_NT'">
  <Content Include="..\..\lib\engine\win\x64\dtSearchEngine.dll"
           Link="dtSearchEngine.dll">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </Content>
</ItemGroup>

<ItemGroup>
  <Reference Include="dtSearchNetStdApi">
    <HintPath>..\..\lib\engine\NetStd\dtSearchNetStdApi.dll</HintPath>
  </Reference>
</ItemGroup>
```

Note that these entries all reference the x64 versions of the libraries.

Deploying the Application

The program reads the text files and populates the database. The text files are included in the shakespeare-text.zip file. A handy feature of connecting through AWS Toolkit is that you can check a box to map your local drives as resources that you can access from the remote system. Unzip the file into *C:\dtSearch*.

We also need to set up the dtSearch Engine for use while the search program runs. Instructions for setting up the dtSearch Engine with your application can be found in the Installing the dtSearch Engine help topic.

With the files on the EC2 instance and the dtSearch Engine installed, we're ready to deploy the application. We'll publish the application to the "publish" folder, then use Remote Desktop Connection to copy the files over to the EC2 instance. See the steps documented under "How to create a new Web application" for details.

The user account "IIS AppPool\dtSearchSetupApp" will need permissions on the folder. Use the Security tab for the folder properties and set Read & Execute, List Folder Contents, and Read permissions.

We specified a different port for each application. You'll need to add the port to the AWS security group and open the port in the firewall settings on the EC2 instance. Then the application can run from a local machine using the EC2 instance's public DNS and port number.

dtSearchSetupApp Application Details

The setup application (dtSearchSetupApp) locates the files to be indexed, sets them up in a database, and indexes the database using the dtSearch Engine's DataSource API. In ***Index.cshtml*** we see four buttons on a form.

```
<form method="post">
  <button type="submit" class="btn btn-default"
    asp-page-handler="EnumFiles">Find Files</button>
  <button type="submit" class="btn btn-default"
    asp-page-handler="ClearDB">Clear DB</button>
  <button type="submit" class="btn btn-default"
    asp-page-handler="ImportFiles">Import</button>
  <button type="submit" class="btn btn-default"
    asp-page-handler="IndexContent">Index</button>
</form>
```

Here's what each option does:

- "Find Files" reads the list of files in the folder. If no files display, then check that they're in the correct folder.
- "Clear DB" runs a query to delete all items from the database.
- "Import" loads text from the supplied files and inserts one record for each file.
- "Index" reads the records from the database and builds the index.

Let's take a closer look at the indexing operation.

```
/// React to the Index button. Create the index from the database contents

public void OnPostIndexContent()
{
    bool result = false;
    // get connection
    MySqlConnection conn = GetConnection();
    try
    {
        conn.Open();

        // create our custom data source, pass in connection
        DBDataSource dataSource = new DBDataSource(conn);

        // create the index job and set basic params
        IndexJob indexJob = new IndexJob();
        indexJob.ActionAdd = true;
        indexJob.ActionCreate = true;
  indexJob.IndexingFlags |= IndexingFlags.dtsIndexCacheOriginalFile;
  indexJob.IndexingFlags |= IndexingFlags.dtsIndexCacheText;

        // Instead of "FoldersToIndex" we use "DataSourceToIndex"
        // and set it to our derived class
        indexJob.DataSourceToIndex = dataSource;
        // Index destination is hard coded here for this example
        indexJob.IndexPath = "H" + Path.VolumeSeparatorChar
            + Path.DirectorySeparatorChar
            + "dtSearch" + Path.DirectorySeparatorChar
            + "index" + Path.DirectorySeparatorChar;

        // execute the job and capture the result
        result = indexJob.Execute();

        indexErrors = indexJob.Errors != null ?
            indexJob.Errors.ToString() : "";
    }
    catch (Exception ex)
    {
        dbError = ex.ToString();
        System.Diagnostics.Debug.WriteLine(ex.ToString());
    }

    Message("DONE INDEXING with result = " + result.ToString());
}
```

This function sets up an IndexJob (see the previous article for details). In this case, however, we used the provided `IndexJob` class rather than extending it.

When indexing databases, it is often useful to cache the documents in the index so hit-highlighted results can be displayed easily and quickly after a search. There are two types of caching:

- caching of plain text, used with `SearchReportJob` to efficiently generate a brief hits-in-context display for search results.
- caching of original documents, used with `FileConverter` to efficiently generate hit-highlighted versions of a complete document to display when a user selects an item in the search results.

To enable both types of caching, set the flags `dtsIndexCacheText` and `dtsIndexCacheOriginalFile` in `IndexJob`. You can find more information about caching in the Caching documents topic in the dtSearch documentation.

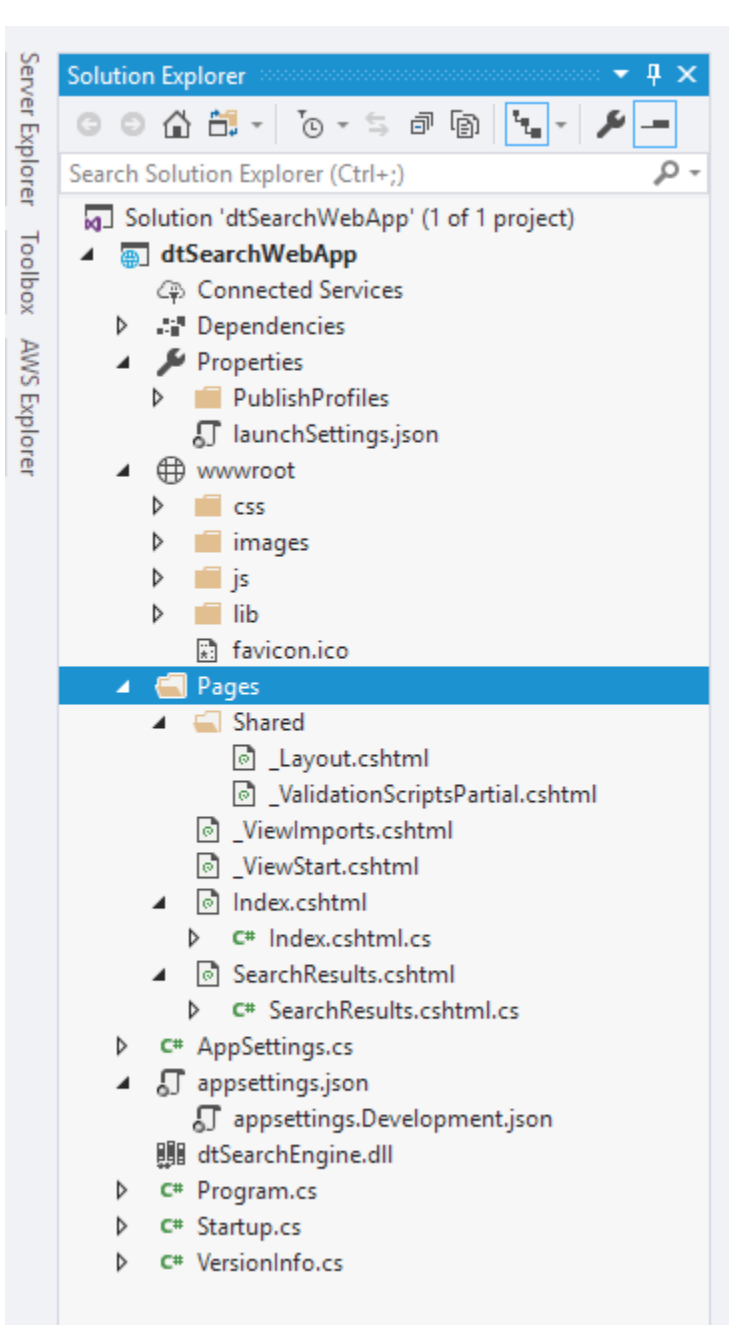
```
/// GetNextDoc override. The engine calls this to see if it
/// should continue indexing, and to set up the next item

public bool GetNextDoc()
{
    skip++;
    string sql = "SELECT doc_id, doc_file, doc_name, doc_content FROM ShakespeareDoc
ORDER BY doc_id LIMIT " + skip + ", 1";

    // create command, read database
    MySqlCommand cmd = new MySqlCommand(sql, connection);
    MySqlDataReader rdr = cmd.ExecuteReader();
    // set basic settings about index item
    DocIsFile = false;

    // we know in this case that all records have data
    // if rdr returns true then we're good, otherwise we're done
    if (rdr.Read())
    {
        DocId = (int)rdr[0];
        DocName = rdr[1].ToString();
        DocDisplayName = rdr[2].ToString();
        DocText = rdr[3].ToString();
        rdr.Close();
        return true;
    }
    else
    {
        return false;
    }
}
```

We extended the `dtSearchEngine.DataSource` class so that we could control the text being fed into the `IndexJob`. We use the "skip" variable to control where we are in the records using the LIMIT SQL clause. Each time `IndexJob` calls `GetNextDoc`, our class reads another record from the database, then sets up the data accordingly. When we run out of data in the database, we return false to let `IndexJob` know that the job is finished.

Once this is complete, the next step is searching the index.

Creating the Search App

Open the dtSearchWebApp solution in your development folder.



As with the setup app, we started with a .NET Core Web Application and removed unnecessary components.

VersionInfo.cs is explained in the previous article: it checks version information for the dtSearch Engine.

There are a couple of things to point out in Startup.cs.

```
public class WebDemoIndexCache : IndexCache
{
  public WebDemoIndexCache(IOptions<AppSettings> settings) :
    base(settings.Value.IndexCache.MaxIndexCount)
  {
    AutoReopenTime = settings.Value.IndexCache.AutoReopenTime;
    AutoCloseTime = settings.Value.IndexCache.AutoCloseTime;
  }
}

public class Startup
{
  private void EnableDebugLogging()
  {
    string DebugLogName = Path.Combine(Path.GetTempPath(),
      "dtSearchWebApp.log");
    Server.SetDebugLogging(DebugLogName, DebugLogFlags.dtsLogDefault);
  }
  public Startup(IConfiguration configuration)
  {
    // Un-comment to generate a diagnostic log
    EnableDebugLogging();
    Configuration = configuration;
  }
  ...
}
```

The `IndexCache` object used here is included in the dtSearch Engine API to improve performance in applications that do a lot of searching. It maintains a cache of already-opened indexes that can be re-used in searches. We set some options for the cache here, along with options for the log file. There's an `AppSettings` class to hold the options, but the actual values are saved in *appsettings.json*.

Let's take a look at a part of ***Index.cshtml*** and the corresponding code in ***Index.cshtml.cs***.

```
<input asp-for="SearchRequest" id="SearchRequest"
  class="typeahead form-control" autocomplete="off"
  type="text" placeholder="Search request"
  value="@Model.SearchRequest" />

  [BindProperty(SupportsGet = true)]
  public string SearchRequest { set; get; }
```

There's an input for `SearchRequest` in the cshtml. In the code-behind file, there's a corresponding bound property. This is the pattern followed for the search terms and all the options needed for the search job.

Only a few of the available search options are present in this example.

- `SearchType` controls whether the search job looks for indexed items that match any word, all words, or Boolean conditions such as "dream AND caesar".
- Stemming allows the search job to locate terms based on a stem term such as dreamer, dream, and dreaming all by searching for "dream".
- Phonic searching finds words that sound like what is written in the search term.

Searching!

Finding the documents with matches in the index is done by the `SearchJob` class.

The search job can search more than one index, so the top section of code builds a list of indexes. This example only uses a single index, so the index property is a hidden input in the form.

Next, we set the options for the search job. The path to the index we created goes into the `IndexesToSearch` property. We set the search terms into Request along with any Boolean conditions.

```
/// Run the search using the words entered on the form and some options.

private IActionResult DoSearch()
{
  ...

  // all values for IxId into one comma-delimited string
  string IxIdString = "";
  foreach (var id in IxId)
  {
    if (IxIdString.Length > 0)
      IxIdString += ",";
    IxIdString = IxIdString + id;
  }
  if (string.IsNullOrWhiteSpace(IxIdString))
    IxIdString = Settings.IndexTable.GetDefaultIndexIds();
  IndexIds = IxIdString.Split(",");
  IndexesToSearch = Settings.IndexTable.GetIndexPaths(IxIdString);

  using (SearchJob searchJob = new SearchJob())
  {
    searchJob.IndexCache = indexCache;
    searchJob.IndexesToSearch = IndexesToSearch;
    searchJob.Request = SearchRequest;
    searchJob.BooleanConditions = BooleanConditions;

    searchJob.SearchFlags = dtsSearchDelayDocInfo;
    if (SearchType == SearchType.AllWords)
      searchJob.SearchFlags |= dtsSearchTypeAllWords;
    else if (SearchType == SearchType.AnyWords)
      searchJob.SearchFlags |= dtsSearchTypeAnyWords;
    if (Stemming)
      searchJob.SearchFlags |= dtsSearchStemming;
    if (PhonicSearching)
      searchJob.SearchFlags |= dtsSearchPhonic;

    searchJob.SearchFlags |= (SearchFlags)SearchFlags;

    bool ok = ExecuteSearch(searchJob);
    if (!ok)
    {
      string message = searchJob.Errors.ToString();
      return ShowError(message);
    }
  }
  stopwatch.Stop();

  // optionally generate a synopsis for the results
  if (Settings.Synopsis.GenerateSynopsis)
    GenerateSynopsisForThisPage();

  return Page();
}
```

`SearchFlags` is a collection of different items, only a few of which are demonstrated in this example. See the SearchFlags Enumeration documentation for details.

When the index search is complete, the `ExecuteSearch` function returns. If it returns false, then any errors are set into the `Message` displayed for the user. If true (success!), then the program optionally builds a synopsis for each item. (In this case it does because that setting is true.)

`GenerateSynopsisForThisPage` uses `SearchReportJob` to generate a brief hits-in-context display for each document showing a few hits with some context around each hit. Because we built the index with caching of text enabled, `SearchReportJob` can generate this quickly and without going back to the database to get the original documents.

```
private void GenerateSynopsisForThisPage()
{
  Stopwatch stopwatch = new Stopwatch();
  stopwatch.Start();
  using (SearchReportJob reportJob = new SearchReportJob())
  {
    reportJob.SetResults(SearchResults);
    reportJob.OutputFormat = OutputFormat.itUnformattedHTML;
    reportJob.BeforeHit = "";
    reportJob.AfterHit = "";

    reportJob.WordsOfContextExact = Settings.Synopsis.WordsOfContext;
    reportJob.ContextFooter = Settings.Synopsis.ContextFooter;
    reportJob.ContextHeader = Settings.Synopsis.ContextHeader;
    reportJob.ContextSeparator = Settings.Synopsis.ContextSeparator;
    reportJob.MaxContextBlocks = Settings.Synopsis.MaxContextBlocks;
    reportJob.MaxWordsToRead = Settings.Synopsis.MaxWordsToRead;
    reportJob.SelectItems(0, SearchResults.Count);
    reportJob.Flags = ReportFlags.dtsReportGetFromCache |
                      ReportFlags.dtsReportLimitContiguousContext |
                      ReportFlags.dtsReportStoreInResults;
    if (Settings.Synopsis.IncludeFileStart)
      reportJob.Flags |= ReportFlags.dtsReportIncludeFileStart;
    reportJob.Execute();
  }
  stopwatch.Stop();
  _log.LogInformation(EventId.SearchReport,
"SearchReport: \"{SearchRequest}\" Time: {SearchTime} Results count: {Count}",
SearchRequest, stopwatch.ElapsedMilliseconds, SearchResults.Count);
}
```

The synopsis is generated by the SearchReportJob class. We use `SetResults` to focus the report on the `SearchResults` returned by the IndexJob. We use `BeforeHit` and `AfterHit` to make the word bold in the output.

It's important to note that we're dealing with the complete search results all at once in the `GenerateSynopsisForThisPage` function. In a production environment you might want to set up a paging mechanism or some other limit to the displayed results.

Showing the Search Results

At the bottom of Index.cshtml we see a call to the partial view called `SearchResults`. Open the *SearchResults.cshtml* file.

```
<div class="panel-heading">
  <h4>@Model.SearchResults.Request</h4>
</div>
<div class="panel-body">
  @Model.SearchResults.TotalFileCount.ToString("#,#") files with
@Model.SearchResults.TotalHitCount.ToString("#,#") hits
</div>

<table class="table table-hover">
  <thead class="blue-grey lighten-4">
    <tr>
      <th>Hits</th>
      <th>Document</th>
      <th>Sample</th>
    </tr>
  </thead>
  <tbody>
    <!-- Show each item in a table row with the hit count and the synopsis which
shows some sample hits from the file -->
    @{
      for (int i = 0; i < Model.SearchResults.Count; ++i)
      {
        SearchResultsItem item = new SearchResultsItem();
        if (Model.SearchResults.GetNthDoc(i, item))
        {
          <tr>
            <td class="HitsColumn">@item.HitCount</td>
            <td class="HitsColumn">@item.DisplayName</td>
            <td class="HitsColumn">
              @if (!string.IsNullOrWhiteSpace(item.Synopsis))
              {
                @Html.Raw(item.Synopsis)
              }
            </td>
          </tr>
        }
      }
    }
  </tbody>
</table>
```

The first thing it does is check to see if there was an error loading the results. If there are no errors, it checks to see if there are any results. (These two checks are not shown.) Then it refers to the `SearchResults` item in the model to build up the rest of the screen.

- `@Model.SearchResults.Request` shows what was typed in for search terms
- `@Model.SearchResults.TotalFileCount` provides the number of files that had at least one hit
- `@Model.SearchResults.TotalHitCount` provides the total of all the hits in all files

The last thing the screen does is to set up a table of the results. Check the SearchResultsItem Class for details on what we can show.

Wrapping Up

In this demonstration we set up an Aurora MySQL database, created the table, then deployed a program to populate the table with text from files and created an index from the text data. We used another program to search the index and display the search results on a web page.

The sample accompanying this article was derived from the WebDemo app found in the dtSearch Engine installation folder, ***\Program Files (x86)\dtSearch Developer\examples\NetStd\WebDemo***. The WebDemo demonstrates more features of index searching including faceted search and paging through results. Browse the ***Program Files (x86)\dtSearch Developer\examples*** folder for many examples of using the dtSearch Engine.

More on dtSearch
dtSearch.com
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
Put a Search Engine in Your Windows 10 Universal (UWP) Applications
Indexing SharePoint Site Collections Using the dtSearch Engine DataSource API
Working with the dtSearch® ASP.NET Core WebDemo Sample Application
Using dtSearch on Amazon Web Services with EC2 & EBS
Full-Text Search with dtSearch and AWS Aurora

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Share

About the Author

Mike V Baker
Software Developer (Senior) Reliance Interactive Training Solutions Inc.
United States

Software engineer / eLearning developer.
Since 1993
Hoover Dam, Media Shoppe, Pitsco Innovative Education, Macromedia, Adobe, Reliance Interactive
Authorware, Flash, Director, C/C++, VB, C#, Objective-C, Swift, Java

Comments and Discussions

You must Sign In to use this message board.

Spacing Relaxed Layout Normal Per page 25

-- There are no messages in this forum --

Article Copyright 2019 by Mike V Baker
Everything else Copyright © CodeProject, 1999-2019

Web01 2.8.190806.1