Article
Browse Code
View Stats
Comments
Posted 15 Jul 2019

Tagged as

ASP.NET
Windows
.NET
Amazon
Cloud
AWS

Stats

811 views
25 downloads

Using dtSearch on Amazon Web Services with EC2 & EBS

Mike V Baker
15 Jul 2019 CPOL

This article will demonstrate the use of Elastic Cloud Compute (EC2) to create virtual machines and deploy applications on them, and Elastic Block Store (EBS) to create virtual disk volumes and attach them to the EC2 instances.

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 dtSearchConsoleApp.zip - 183.9 KB

Imagine harnessing the power of the dtSearch Engine to index and search Microsoft Office documents, PDFs, email, and other data with the worldwide accessibility and storage capacity of Amazon Web Services (AWS). This article will demonstrate the use of Elastic Cloud Compute (EC2) to create virtual machines and deploy applications on them, and Elastic Block Store (EBS) to create virtual disk volumes and attach them to the EC2 instances.

We'll use the dtSearch Engine to create a console app and deploy it to EC2. We'll use the console app to create an index of a data collection, then use the index to leverage the advanced search capabilities of the dtSearch Engine.

Project Prerequisites

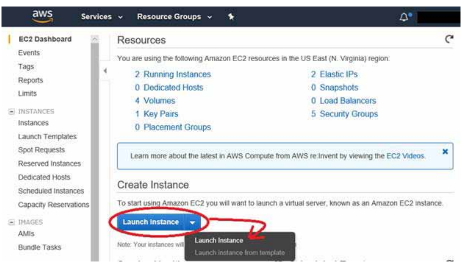
Setting up the project, we'll create a single EC2 instance and attach two EBS volumes to that instance. One EBS volume will contain data to be indexed and the other EBS volume will hold the completed index. Note that both the source material and the index could be saved on the same EBS volume. However, since the source material or real-world application requirements could expose the need for separate stores, we'll demonstrate how to implement storing the index on a separate EBS volume.

This article assumes we already have an AWS account, so start by logging into the 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.

Note that EBS doesn't exist outside of EC2 since you can't use EBS volumes outside the context of EC2 instances. To create an EC2 instance and the EBS volumes that we need for this project, start at the EC2 dashboard. You can find the EC2 item in the "Compute" section.

Create an EC2 Instance

In the EC2 dashboard, use the "Launch Instance" button to create a new instance.



[redacted]

Resources

You are using the following Amazon EC2 resources in the US East (N. Virginia) region:

2 Running Instances
0 Dedicated Hosts
4 Volumes
1 Key Pairs
0 Placement Groups
2 Elastic IPs
0 Snapshots
0 Load Balancers
5 Security Groups

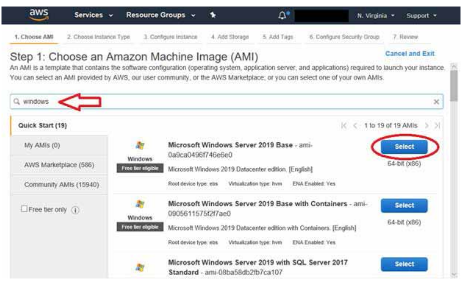
Learn more about the latest in AWS Compute from AWS re:Invent by viewing the EC2 Videos.

Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Note: Your instances will

For this demo we'll use a Microsoft Windows Server 2019 Base image.



[redacted]

Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

windows

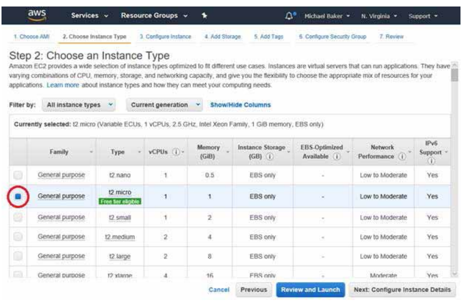
Microsoft Windows Server 2019 Base - ami-0a9ca0496f746e6e0
Microsoft Windows 2019 Datacenter edition. [English]
Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

Microsoft Windows Server 2019 Base with Containers - ami-0905611575f2f7ae0
Microsoft Windows 2019 Datacenter edition with Containers. [English]
Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

Microsoft Windows Server 2019 with SQL Server 2017 Standard - ami-08ba58db2fb7ca107

There is a list of steps needed for setting up an EC2 instance that the AWS console walks us through. We can see the list across the top of the screen.

For this example, we're going to select the t2.micro instance type. Note that this instance includes an EBS storage volume.



Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

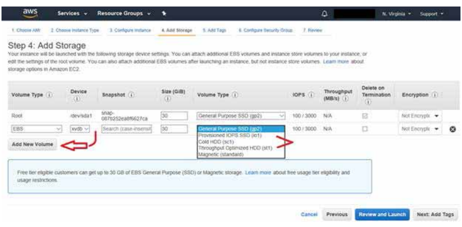
Filter by: All instance types Current generation Show/Hide Columns

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
●	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.large	2	8	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.xlarge	4	16	EBS only	-	Moderate	Yes

Choose the default for step 3 — all the defaults will work for what we're doing.

In step 4 we can add more storage. This is where we'll set up the second EBS volume for the index.



[redacted]

Step 4: Add Storage

Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. Learn more about storage options in Amazon EC2.

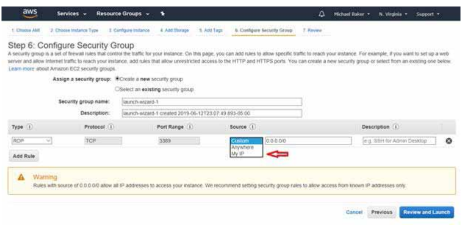
Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encryption
Root	/dev/sda1	snap-087b252ea8f6627ca	30	General Purpose SSD (gp2)	100 / 3000	N/A	☑	Not Encrypted
EBS	xvdb	Search (case-insensit	30	General Purpose SSD (gp2)	100 / 3000	N/A	☐	Not Encrypted

Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. Learn more about free usage tier eligibility and usage restrictions.

For this specific demo, the default 8 GB volume will be more than enough. Refer to EBS volume types for more information about different types of EBS volumes.

We don't need to add any custom tags so skip step 5.

All EC2 machines are assigned a Security Group. The default type that the wizard creates has a firewall rule for RDP, which will be the one we use to connect and control the machine after it's launched. To limit this rule to our own machine, select the "My IP" option from the select box and the wizard will load the IP.



Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

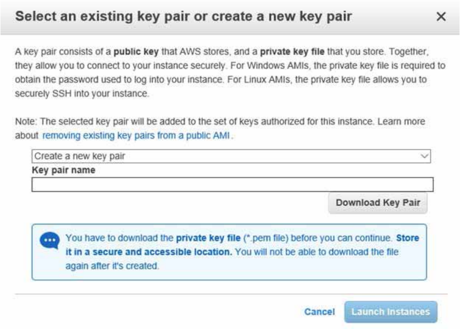
Assign a security group: ● Create a **new** security group
○ Select an **existing** security group

Security group name: launch-wizard-1
Description: launch-wizard-1 created 2019-06-12T23:07:49.893-05:00

Type	Protocol	Port Range	Source		Description
RDP	TCP	3389	Custom	0.0.0.0/0	e.g. SSH for Admin Desktop

Warning
Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

Click the "Review and Launch" button to navigate to step 7. Then click the "Launch Instance" button. The console displays a new box asking about a key pair.



Select an existing key pair or create a new key pair

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about removing existing key pairs from a public AMI.

Create a new key pair
Key pair name

Download Key Pair

You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

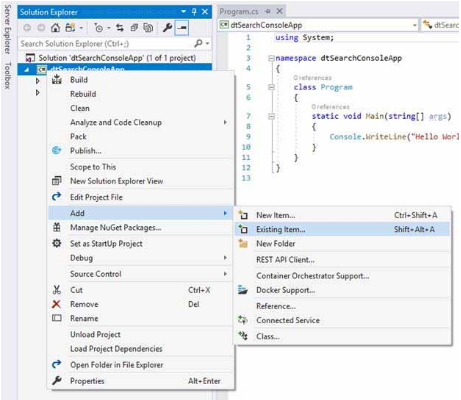
Cancel Launch Instances

For details on the EC2 key pairs review the Amazon EC2 Key Pairs documentation. We can use an already generated EC2 key pair if one exists. We can also select the option of creating a key pair now. If we create one now, the browser should automatically download the private key.

Using the dtSearch API in an App

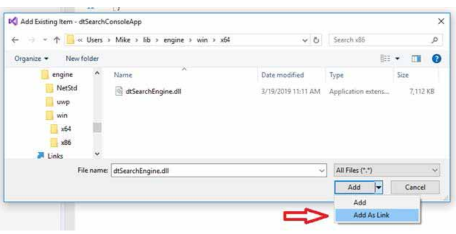
We created a very simple console application project to demonstrate how to connect the dtSearch Engine to an app and use the search engine. We'll walk through some details of the application here. Download the source code for the project to get started.

Start by opening the sample project in Visual Studio. In order to use the dtSearch Engine from an app we're going to want to deploy the dtSearchEngine.dll file. Right-click the project and choose Add > Existing Item from the context menu.



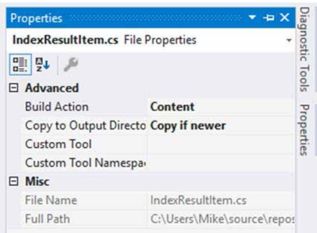
```
using System;

namespace dtSearchConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello Worl
        }
    }
}
```

For the purposes of this article, we included the lib folder with the code and used a relative reference in the project file. For building your own projects, you should load directly from dtSearch installation folder ***/lib***. If you're building a .NET Framework C# app like the console app in the sample, then you navigate to ***/lib/engine/win/x64*** and select the DLL from that folder.



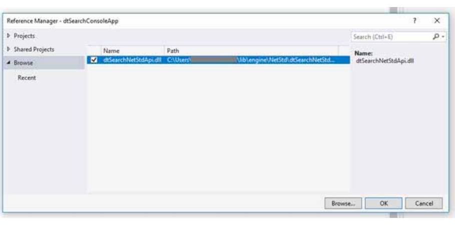
Be sure to select "Add As Link" so the project will reference the DLL in the installed location (so we don't wind up with copies of the DLL in various project locations).

We want the DLL to be copied to the build folder, so we have to update the properties for the DLL in the project. Set "Build Action" to "Content" and the "Copy" property to "Copy if Newer".



Build Action	Content
Copy to Output Directo	Copy if newer
Custom Tool	
Custom Tool Namespa	
File Name	IndexResultItem.cs
Full Path	C:\Users\Mike\source\repos

Next we add a reference to ***dtSearchNetStdApi.dll*** to the project dependencies. Right-click and choose Add Reference, then click Browse and find the DLL in the ***/lib/engine/NetStd/*** folder. Click OK. That's all we need to do to connect the dtSearch Engine to the program.



☑ dtSearchNetStdApi.dll C:\Users\[redacted]\lib\engine\NetStd\dtSearchNetStd...

Name:
dtSearchNetStdApi.dll

Program Startup, Check for the Engine

Now let's look at the program itself. We're going to look at some supporting pieces first, and build our way up to the program. But first we need a check that ensures the dtSearch Engine is accessible. Open the sample app and look at "*VersionInfo.cs*".

Hide Copy Code

```
private void GetEngineVersion()
{
  try
  {
    dtSearch.Engine.Server server = new dtSearch.Engine.Server();
    EngineMajorVersion = server.MajorVersion;
    EngineMinorVersion = server.MinorVersion;
    EngineBuild = server.Build;
    EngineVersion = server.MajorVersion +
                    "." +
                    server.MinorVersion +
                    " Build " +
                    server.Build;
    LoadedOK = true;
  }
  //code to catch errors if dtSearch.Engine fails to load not shown
}
```

This function calls the dtSearch Engine, and if it fails to load there are catch handlers to set the `LoadError` status appropriately. Our app will call this and check the condition to ensure that it loads correctly.

Close that file and open ***IndexResultItem.cs***.

Hide Copy Code

```
public IndexResultItem(IndexFileInfo info, MessageCode updateType)
{
  Filename = info.Name;
  Location = info.Location;

  if (updateType == MessageCode.dtsnIndexFileOpenFail)
  {
    Error = info.OpenFailMessage;
    Detail = "Not indexed: " + Error;
  }
  else if (updateType == MessageCode.dtsnIndexFileDone)
  {
    Success = true;
    WordCount = info.WordCount;
    Detail = info.Type + " " + info.WordCount + " words";
  }
  else if (updateType == MessageCode.dtsnIndexFileEncrypted)
  {
    Encrypted = true;
    Detail = "Encrypted";
  }
}
```

The constructor expects an `IndexFileInfo` item. If we select the item and press F12 (or right-click and select "Go to Definition"), we can see the items available in this class. The dtSearch Engine keeps a lot of information on each item being indexed. Find details on this in the IndexFileInfo Members documentation.

Return to ***IndexResultItem.cs*** and do the same "Go to Definition" step for `MessageCode`. We can see there are roughly 40 different codes. Find details on these in the dtSearch.Engine.MessageCode Enumeration documentation. We're going to be using a few of these codes later in the project.

Handling Status Updates From The Engine

Next we'll look at ***IndexStatusHandler.cs***. We can see that it implements the `dtSearch.Engine.IIndexStatusHandler` interface, which has two methods, `CheckForAbort` and `OnProgressUpdate`.

Hide Copy Code

```
class IndexStatusHandler : IIndexStatusHandler
```

The dtSearch Engine calls `CheckForAbort` to see if it should quit as a result of some failure that we determine should be fatal or a user action that allows the user to stop. In this class we'll see Cancelled and Stopped as two boolean flags. In the `CheckForAbort` function we see that Cancelled is used to abruptly terminate the process and Stopped is used to quit the process when it finishes the current task.

Hide Copy Code

```
public AbortValue CheckForAbort()
{
  if (Cancelled)
    return AbortValue.CancelImmediately;
  else if (Stopped)
    return AbortValue.Cancel;
  else
    return AbortValue.Continue;
}
```

The `OnProgressUpdate` function receives an `IndexProgressInfo` with information about the current status of an index update.

Hide Copy Code

```
public void OnProgressUpdate(IndexProgressInfo info)
{
  // sample of putting something in the log.
  Server.AddToLog("Index progress: " + info.PercentDone + "%");

  // call the progress reporter (dtSearchApp) when info.PercentDone changes
  if ((ProgressReporter != null) && info.PercentDone !=
      percentDoneReported && !Cancelled)
  {
    percentDoneReported = info.PercentDone;
    ProgressReporter.Report(info.PercentDone);
  }
  ...
```

See IndexProgressInfo Members in the dtSearch documentation for details.

If we look at the class declaration for `IndexProgressInfo`, we can see at the very top that it has an associated `IndexFileInfo` object, so these progress items pertain to a particular file being indexed.

Hide Copy Code

```
public class IndexProgressInfo
{
  public IndexFileInfo File;

  public IndexProgressInfo();

  public uint CurrMergePercent { get; set; }
  public uint EstRemainingSeconds { get; set; }
  public uint ElapsedSeconds { get; set; }
  ... //more
}
```

The `ProgressReporter` item in `OnProgressUpdate` is just a function passed in that reports on changes in the `info.PercentDone` property.

Below the progress reporting there's a switch that handles the `info.UpdateType` property.

Hide Copy Code

```
switch (info.UpdateType)
{
    case MessageCode.dtsnIndexFileDone:
    case MessageCode.dtsnIndexFileOpenFail:
    case MessageCode.dtsnIndexFileEncrypted:
        if (FileList.Count < MaxListSize)
            FileList.Add(new IndexResultItem(info.File, info.UpdateType));
        break;
    case MessageCode.dtsnIndexDone:
        Result = info;
        break;
    default:
        break;
}
```

Remember `MessageCode` from earlier? The `UpdateType` property is of type `MessageCode`, so here we can react to any of those roughly 40 different items. In this sample it's handling the ones that indicate a file is finished by adding that info to the `FileList`. A production app would need a more robust mechanism to handle storing info on what could be millions of files.

The Index Job

In ***SampleIndexJob.cs*** we can see that the file extends the `IndexJob` class, a `dtSearch.Engine` class.

Hide Copy Code

```
public class IndexJob : JobBase

{...
```

As before, we can use "Go to Definition" on `IndexJob` and view the properties available and the one and only method we can override: `Execute()`. The full documentation for these members can be found under IndexJob Members.

For this simple example we're only going to touch on a few of these members. We will override Execute and we'll set a couple of the options, in addition to telling the dtSearch Engine what folder to index.

Return to ***SampleIndexJob.cs*** and look at the constructor. It accepts the progress handler that we discussed earlier. `IndexStatusHander` needs that, so in the constructor we pass it on. The `dtSearch.Engine.IndexJob.StatusHandler` is expecting the `IndexStatusHander`, so we assign that here.

Hide Copy Code

```
public SampleIndexJob(Action aProgressHandler = null)
{
  indexStatusHandler = new IndexStatusHandler();
  if (aProgressHandler != null)
    indexStatusHandler.ProgressReporter = new Progress(aProgressHandler);
```

```
  StatusHandler = indexStatusHandler;
}

public override bool Execute()
{
  indexStatusHandler.BeforeExecute();

  return base.Execute();
}
```

Next we see our override of the `IndexJob.Execute()` function. It calls `indexStatusHandler.BeforeExecute()` to clear out data leftover from a previous run. This allows the program to run multiple indexing jobs in a single run of the program.

The other items in this class are self-explanatory.

Calling the Job

Finally, actual work begins here. Open up the ***dtSearchApp.cs*** file.

The `Run()` function is the main entry point for the index job. First, it calls the `VersionInfo` object to ensure the dtSearch Engine is present. If the load fails, we exit.

The program then asks which operation to perform, (I)ndex, (S)earch, or (Q)uit. We'll look at (I)ndex first.

The `BuildIndex()` function asks where the docs are that we want indexed. For this example, we provide works by Shakespeare in the "docs" folder. Then it asks for the folder where the index should be stored. For this example, we want to store the index on another EBS volume.

The following section creates a new instance of the `SampleIndexJob` class and passes in the "`ShowProgress`" function (this is our `ProgressReporter` function). Then it sets the options we need, `IndexPath`, `ActionCreate`, and `ActionAdd`. Then we add the folder to index to `FoldersToIndex`. (`FoldersToIndex` is a `List<string>`, so we could add a number of different folders, anywhere on our system.) Then it calls `Execute()`.

Hide Shrink ▲ Copy Code

```
// create a job to build the index
using (SampleIndexJob job = new SampleIndexJob(ShowProgress))
{
  // fill in the options
  job.IndexPath = indexPath;
  job.ActionCreate = true;
  job.ActionAdd = true;

  // put in one (or more) folders to index
  job.FoldersToIndex.Add(docsPath);

  // At last, some activity!
  job.Execute();

  // indexing finished here, post results
  var fileList = job.GetFilesIndex();
  int logCount = 0;
  Console.WriteLine(Environment.NewLine + "Results:");
  foreach (var file in fileList)
  {
    Console.WriteLine(file.Filename + " " + file.Detail);
    logCount++;
    if (logCount > 25)
    {
      if (AskYesNo("See more items?"))
        logCount = 0;
      else
        break;
    }
  }
  Console.WriteLine(job.SummarizeIndexResult());
}
Console.WriteLine("Indexing complete");
```

That's it. Create a handler for index status (the `IndexStatusHander`), extend the `IndexJob (SampleIndexJob)`, provide a UI for obtaining the information we need to run, and call the job's `Execute()` function. This is the simplest case of using dtSearch Engine in your own application.

We have only one more step, connect it to `Main()`.

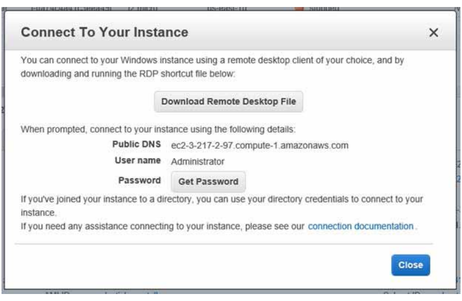
Open ***Program.cs*** and you'll see that all that's needed here is to create a dtSearchApp instance and call `Run()`.

Deploying to AWS

This console app will run on the EC2 instance without any special modifications. We need to build the release version, copy it to the EC2 instance, and call dotnet to run it. Switch the Visual Studio target to Release and use Rebuild All. Then copy the files from the output directory to the EC2 instance.

We connect to EC2 using a Remote Desktop Connection. We get the connection details by downloading a file from AWS.

Log back into AWS, find the EC2 dashboard, select the EC2 instance to use, and click the Connect button. A new dialog will appear that allow us to download the RDP file needed to connect to the EC2 instance. We can also click the button to get the Administrator password for that machine.



Connect To Your Instance

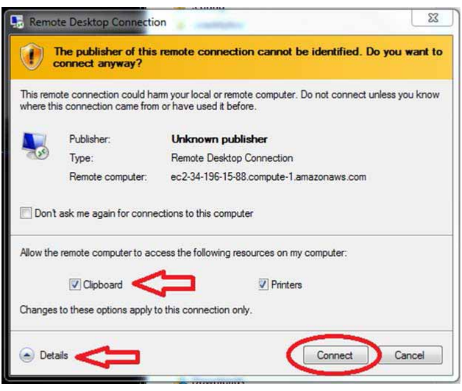
You can connect to your Windows instance using a remote desktop client of your choice, and by downloading and running the RDP shortcut file below:

When prompted, connect to your instance using the following details:

Public DNS ec2-3-217-2-97.compute-1.amazonaws.com
User name Administrator
Password

If you've joined your instance to a directory, you can use your directory credentials to connect to your instance.
If you need any assistance connecting to your instance, please see our connection documentation.

Download the file and double-click it to open the Remote Desktop Connection. Expand the details, make sure that sharing the Clipboard is enabled, and connect using the password provided by AWS.



The publisher of this remote connection cannot be identified. Do you want to connect anyway?

This remote connection could harm your local or remote computer. Do not connect unless you know where this connection came from or have used it before.

Publisher: **Unknown publisher**
Type: Remote Desktop Connection
Remote computer: ec2-34-196-15-88.compute-1.amazonaws.com

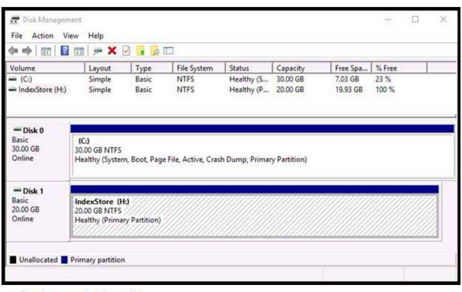
☐ Don't ask me again for connections to this computer

Allow the remote computer to access the following resources on my computer:

☑ Clipboard ☑ Printers

Changes to these options apply to this connection only.

When we first connect to the machine, the second drive might need to be formatted and assigned a drive letter. We used the volume name "`IndexStore`" and assigned drive H: This is what we set as the default index drive in the program. Windows Explorer will show the two drives.



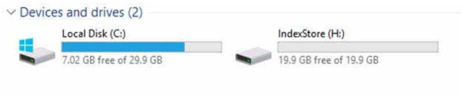
Volume	Layout	Type	File System	Status	Capacity	Free Spa...	% Free
(C:)	Simple	Basic	NTFS	Healthy (S...	30.00 GB	7.03 GB	23 %
IndexStore (H:)	Simple	Basic	NTFS	Healthy (P...	20.00 GB	19.93 GB	100 %

Disk 0
Basic
30.00 GB
Online

(C:)
30.00 GB NTFS
Healthy (System, Boot, Page File, Active, Crash Dump, Primary Partition)

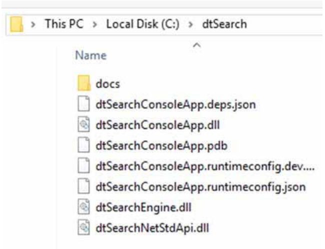
Disk 1
Basic
20.00 GB
Online

IndexStore (H:)
20.00 GB NTFS
Healthy (Primary Partition)



Open drive C:, create a folder named "dtSearch", and open it. Switch back to the local machine, copy the files from the build folder, then paste them into the window on the EC2.

Switch back to the local machine and copy the "docs" folder from the zip file, then paste them into the window on the EC2. The folder should look like this:



This PC › Local Disk (C:) › dtSearch

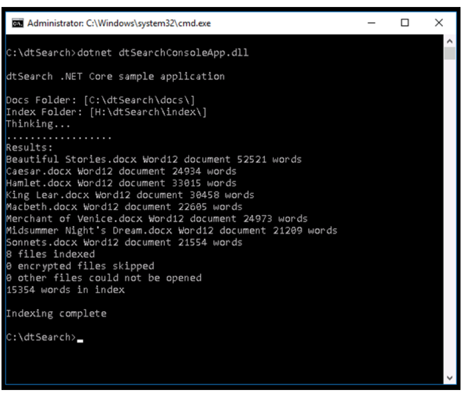
Name
docs
dtSearchConsoleApp.deps.json
dtSearchConsoleApp.dll
dtSearchConsoleApp.pdb
dtSearchConsoleApp.runtimeconfig.dev....
dtSearchConsoleApp.runtimeconfig.json
dtSearchEngine.dll
dtSearchNetStdApi.dll

We can run the .NET Core console app from the command line using dotnet. Open up a command window in the dtSearch folder (Shift-Right-Click) and run this command:

Hide Copy Code

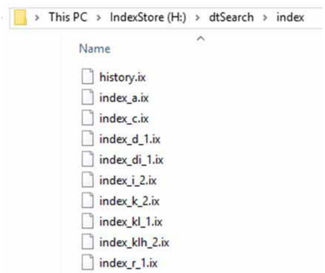
```
dotnet dtSearchConsoleApp.dll
```

The program asks for the docs and index paths. The results will display in the window.



```
C:\dtSearch>dotnet dtSearchConsoleApp.dll

dtSearch .NET Core sample application

Docs Folder: [C:\dtSearch\docs\]
Index Folder: [H:\dtSearch\index\]
Thinking...
..................
Results:
Beautiful Stories.docx Word12 document 52521 words
Caesar.docx Word12 document 24934 words
Hamlet.docx Word12 document 33015 words
King Lear.docx Word12 document 30458 words
Macbeth.docx Word12 document 22605 words
Merchant of Venice.docx Word12 document 24973 words
Midsummer Night's Dream.docx Word12 document 21209 words
Sonnets.docx Word12 document 21554 words
8 files indexed
0 encrypted files skipped
0 other files could not be opened
15354 words in index

Indexing complete

C:\dtSearch>_
```

The destination folder is created and the index will be placed in the index path we entered.



This PC › IndexStore (H:) › dtSearch › index

Name
history.ix
index_a.ix
index_c.ix
index_d_1.ix
index_di_1.ix
index_i_2.ix
index_k_2.ix
index_kl_1.ix
index_klh_2.ix
index_r_1.ix

index_sv.ix
index_v.ix
index_v.ix
indexlog.dat

Seeing the Results

Now that we can build an index, we can use the index for searching. The `SearchIndex()` function does this part.

Hide Copy Code

```
// Run search on the index at the indexPath
private void SearchIndex()
{
    // gather info and init
    indexPath = GetIndexPath();
    searchRequest = GetInput("Search for", searchRequest);
    if (searchResults != null)
        searchResults.Dispose();
    searchResults = new SearchResults();

    // create a search job, set the parameters, and execute it
    using (SearchJob searchJob = new SearchJob())
    {
        searchJob.Request = searchRequest;
        searchJob.MaxFilesToRetrieve = 10;
        searchJob.IndexesToSearch.Add(indexPath);
        searchJob.SearchFlags = SearchFlags.dtsSearchDelayDocInfo;
        searchJob.Execute(searchResults);
    }
    ShowResults();
}
```

`SearchIndex()` requests an `indexPath` and asks the user for search text. It creates a new instance of the `SearchResults` class. We then create a new SearchJob and set the parameters including the search text, the index path, and search flags.

Details on `SearchJob` can be found here. For more information on the search flags, see SearchFlags Enumeration in the documentation.

The final step in the search is to show the results. For that we created a separate function.

Hide Copy Code

```
// Show the results of search on the console window
private void ShowResults()
{
    Console.WriteLine(searchResults.TotalHitCount +
        " Hits in " + searchResults.TotalFileCount + " Files");
    SearchResultsItem item = new SearchResultsItem();
    for (int i = 0; i < searchResults.Count; ++i)
    {
        searchResults.GetNthDoc(i, item);
        Console.WriteLine(item.HitCount + " " + item.Filename);
    }
}
```

The `ShowResults()` function runs through the `SearchResults` item and writes out just the `Filename` and `HitCount` for each item. There are many more properties in the `SearchResults` item object, which we can see in the `SearchResultsItem` Members documentation.

Seeing search results is great, but in a production application users will expect to access search results via a web interface. Fortunately, dtSearch includes a sample project that demonstrates how to use a dtSearch index from an ASP.NET Core web application. If we look in the dtSearch installation directory, we'll find the application in the ***examples\NetStd\WebDemo*** folder. Next, read Working with the dtSearch® ASP.NET Core WebDemo Sample Application for a deeper explanation of how the WebDemo app works. Finally, we can see WebDemo in action at search.dtsearch.com.

Wrapping Up

In this demonstration, we created an EC2 instance with a secondary EBS storage volume. We created a new console project and learned how to attach the dtSearchEngine DLL so that it's included in the output. We also added a reference to the dtSearchNetStdApi to be used by the program.

We reviewed a finished sample console program and learned how to use IndexJob to build an index from any folder to any other folder. We deployed the finished product to an Amazon EC2 instance, ran the program using dotnet on the command line, and saw the index built to the index folder we specified. Finally, we ran a simple search on the index.

The sample accompanying this article was derived from the demo app found in the dtSearch installation folder, ***\Program Files (x86)\dtSearch Developer\examples\NetStd\ConsoleDemo***. Browse the Program Files ***(x86)\dtSearch Developer\examples\*** folder for many sample programs 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