Faceted Search with dtSearch – Not Your Average Search Filter

Jeffrey T. Fritz, 8 Apr 2014

In this article, I'm going to show you how to set up dtSearch with an Entity Framework dataset and then use faceted search navigation to add multiple filters to the result set.

Editorial Note

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.

Part 1: Faceted Search with dtSearch (using SQL and .NET)
Part 2: Turbo Charge your Search Experience with dtSearch and Telerik UI for ASP.NET

Related Article: A Search Engine in Your Pocket -- Introducing dtSearch on Android

Download dtSearch.Web.zip - 493.8 KB

Introduction

I've written my fair share of web applications and web sites that have required search. I recently was introduced to dtSearch and started checking out their library for search indexing and fetching. After spending some time using it, I was impressed with the scope of functionality that it offered. In particular, a feature that I never could seem to get quite right, filtering search results, they nail through their faceted search capabilities. In this article, I'm going to show you how to set up dtSearch with an Entity Framework dataset and then use faceted search navigation to add multiple filters to the result set.

Building a Search Index with Entity Framework

In my scenario, I have a large collection of board game products stored in a SQL Server database that I want to index. With dtSearch, you have several options to create an index, including an option that will inspect a database and index content from every table. For this sample, I wanted to index a single table of products that already had an Entity Framework 6 data context configured. To accomplish this task, I wrote my own `dtSearch.Engine.DataSource` object called `ProductDataSource` that would extract products from my database and present it properly to the dtSearch IndexJob. The DataSource is required to override two methods, Rewind and `GetNextDoc`. Rewind is called when the IndexJob starts, initializing the connect and preparing the DataSource for work. `GetNextDoc` does what it sounds like, it advances in the collection to the next item and makes it available for the IndexJob to crawl and index.

Here is what my Rewind method looks like in the `ProductDataSource` class:

```
private GameShopEntities _GameShopContext = null;
private int _RecordNumber = 0;
public override bool Rewind()
{
  // Initialize a single EF context
  if (this._GameShopContext == null)
  {
    // New() is a static method that passes config string
    this._GameShopContext = GameShopEntities.New();
  }

  _RecordNumber = 0;
  this._TotalRecords = _GameShopContext.Products.Count();
  return true;

}
```

Listing 1 - Rewind Method for ProductDataSource

That's pretty vanilla stuff there. Nothing too complicated, just creating the Entity Framework context and setting the total product count. The more interesting code is what happens in the `GetNextDoc` method. In this method, I set some properties on the `DataSource` base object to declare information about the current product being inspected. Additionally, I call out to two other methods to format the additional fields and the document that I want to store to represent this product.

```
public override bool GetNextDoc()
{
  // Exit now if we are at the end of the record set
  if (_TotalRecords <= _RecordNumber + 1) return false;

  // Reset Properties
  DocName = "";
  DocModifiedDate = DateTime.Now;
  DocCreatedDate = DateTime.Now;

  // Get the product from the data source
  _CurrentProduct = _GameShopContext.Products.Skip(_RecordNumber).First();

  FormatDocFields(_CurrentProduct);
  DocName = _CurrentProduct.ProductNo;
  DocModifiedDate = _CurrentProduct.LastUpdated;
  DocCreatedDate = _CurrentProduct.Created;

  DocStream = GetStreamForProduct(_CurrentProduct);

  _RecordNumber++;
  return true;

}
```

Listing 2 - GetNextDoc Method

The `DocName` property is essentially the primary key for this object in the search index. I use a child method, `FormatDocFields` to add the secondary fields to the collection:

```
public static readonly string[] ProductFields = new[] {
  "Name", "Description", "Weight", "LongDesc", "Age", "NumPlayers", "Price", "Manufacturer"
};
private void FormatDocFields(Product product)
{
  DocFields = "";

  var sb = new StringBuilder();
  var format = "{0}\t{1}\t";
  sb.AppendFormat(format, "Name", product.Name ?? "");
  sb.AppendFormat(format, "Description", product.Description ?? "");
  sb.AppendFormat(format, "Weight", product.Weight.HasValue ? product.Weight.Value : 0);
  sb.AppendFormat(format, "LongDesc", product.LongDesc ?? "");
  sb.AppendFormat(format, "Age", product.Age ?? "");
  sb.AppendFormat(format, "NumPlayers", product.NumPlayers ?? "");
  sb.AppendFormat(format, "Price", product.Price.HasValue ? product.Price.Value: 0.00M);
  sb.AppendFormat(format, "Manufacturer", product.Manufacturer ?? "");

  DocFields = sb.ToString();

}
```

Listing 3 - FormatDocFields Method

This method adds fields in tab delimited pairs to the collection of `DocFields` returned to the `IndexJob`. Not bad, and a simple method to follow as it crawls the properties of the Product object, adding those Product properties that I wish to have in my search index.

The final interesting bit to share is the `GetStreamForProduct` method. This takes the product and turns it into a snippet of HTML appropriate for formatting on screen to show the locations of the search hits in the fields that were returned.

```
private Stream GetStreamForProduct(Product product)
{

  var sb = new StringBuilder();
  sb.Append("<html><dl>");
  var ddFormat = "<dt>{0}</dt><dd>{1}</dd>";
  sb.AppendFormat(ddFormat, "Description", product.LongDesc);
  sb.AppendFormat(ddFormat, "Manufacturer", product.Manufacturer);
  sb.Append("</dl></html>");

  var ms = new MemoryStream();
  var sw = new StreamWriter(ms);
  sw.Write(sb.ToString());
  sw.Flush();

  return ms;

}
```

Listing 4 – GetStreamForProduct Method Listing

How's this for simple semantic markup? It is an HTML definition list, with just the description and manufacturer fields being returned inside of a bare HTML tag. With the HTML tag in place, the dtSearch indexer will identify the snippet as an HTML document. This will allow for more optimized storage and presentation as an HTML fragment later. Realistically, searching should only hit the description and manufacturer fields. The rest is simple `System.IO` stream management to return the HTML segment in the stream format that the dtSearch IndexJob requires.

The last step to index my data and allocate the facets in the index is to configure and run the `IndexJob`:

```
using (var indexJob = new IndexJob())
{
  var dataSource = new ProductDataSource();
  indexJob.DataSourceToIndex = dataSource;
  indexJob.IndexPath = _SearchIndexLocation;
  indexJob.ActionCreate = true;
  indexJob.ActionAdd = true;
  indexJob.CreateRelativePaths = false;

  // Create the faceted index
  indexJob.EnumerableFields = new StringCollection() { "Description","LongDesc", "Age", "NumPlayers",
"Price", "Manufacturer" };

  var sc = new StringCollection();
  sc.AddRange(ProductDataSource.ProductFields);

  indexJob.StoredFields = sc;
  indexJob.IndexingFlags = IndexingFlags.dtsIndexCacheTextWithoutFields |
IndexingFlags.dtsIndexCacheOriginalFile;

  ExecuteIndexJob(indexJob);
}
```

Listing 5 - Indexing with a collection of StoreFields

The `StoredFields` property is where I have declared the fields to use as facets in my search.

Page Layout and Searching

In my sample ASP.NET web forms project, I have allocated a panel and a grid to show the results of a search. My markup looks like the following:

```
<div>

  <asp:Label runat="server" ID="lSearch" AssociatedControlID="txtSearch" Text="Search Term:"></asp:Label>
  <asp:TextBox runat="server" ID="txtSearch"></asp:TextBox>
  <asp:Button runat="server" ID="bDoSearch" Text="Search" OnClick="bDoSearch_Click" />

</div>

<asp:Panel runat="server" ID="pFacets" Width="200" style="float: left;">

</asp:Panel>

<asp:GridView runat="server" ID="resultsGrid" OnPageIndexChanging="results_PageIndexChanging"
AllowPaging="true" AllowCustomPaging="true" AutoGenerateColumns="false" ItemType="
FacetedSearch.ProductSearchResult" ShowHeader="false" BorderWidth="0">
  <PagerSettings Mode="NumericFirstLast" Position="TopAndBottom" />
  <Columns>
    <asp:TemplateField>
      <ItemTemplate>
        <a href='http://www.codeproject.com/Products/<%#: Item.ProductNum  %>' class="productName"><%#:
Item.Name %></a><br />
        <%#: Item.HighlightedResults %>

      </ItemTemplate>
    </asp:TemplateField>
  </Columns>
</asp:GridView>
```

Listing 6 - Markup for the Faceted Search page

With a search textbox and button at the top, there is a panel on the left to list the facets and a grid on the right that will virtually page to iterate over my large collection of products in the search index. Searching and binding to the grid is a monster script, due to the number of configuration options available with the dtSearch tool. Let's take a look at that code:

```
public void DoSearch(int pageNum)
{

  // Configure and execute search
  var sj = new SearchJob();
  sj.IndexesToSearch.Add(_SearchIndexLocation);
  sj.MaxFilesToRetrieve = (pageNum+1) * PageSize;
  sj.WantResultsAsFilter = true;
  sj.Request = txtSearch.Text.Trim();

  // Add filter condition if necessary
  if (!string.IsNullOrEmpty(Request.QueryString["f"]))
  {
    sj.BooleanConditions = string.Format("{0} contains {1}", Request.QueryString["f"],
Request.QueryString["t"]);
  }

  sj.AutoStopLimit = 1000;
  sj.TimeoutSeconds = 10;
  sj.Execute();

  ExtractFacets(sj);

  // Present results
  sj.Results.Sort(SortFlags.dtsSortByRelevanceScore, "Name");
  this._SearchResults = sj.Results;

  // Manual Paging
  var firstItem = PageSize * pageNum;
  var lastItem = firstItem + PageSize;
  lastItem = (lastItem > _SearchResults.Count) ? _SearchResults.Count : lastItem;
  var outList = new List<ProductSearchResult>();
  for (int i = firstItem; i < lastItem; i++)
  {
    _SearchResults.GetNthDoc(i);
    outList.Add(new ProductSearchResult
    {
      ProductNum = _SearchResults.DocName,
      Name = _SearchResults.get_DocDetailItem("Name"),
      HighlightedResults = new HtmlString(HighlightResult(i))
    });
  }

  // Configure and bind to the grid virtually, so we don't load everything
  resultsGrid.DataSource = outList;
  resultsGrid.PageIndex = pageNum;
  resultsGrid.VirtualItemCount = sj.FileCount;
  resultsGrid.DataBind();

}
```

Listing 7 - Search Method

There's lots to see here, starting with the initial configuration of the `SearchJob`. This configuration specifies where the index resides on disk, and how many search results to retrieve. The text of the search box is passed in as the Request property of the `SearchJob` object. Next, a filter condition is applied that looks like another search criteria. This is the additional filter based on a selected facet in our UI. More on that later. The important bit of the `SearchJob` configuration is the `WantResultsAsFilter` property being set to true. This allows the results to be used as the input to the code that will construct the facets for this search.

After the search is executed, the `ExtractFacets` method is called to extract the facet information from the `SearchResults` and format them for the screen. Finally, the `SearchResults` are formatted and bound to the GridView. Interestingly, in the formatting of the results is a call to `HighlightResult`. I'll describe that method after the facet description.

The `ExtractFacets` method performs a quick traversal of the search index based on the results of the original query, extracts and aggregates the values in the fields requested.

```
private void ExtractFacets(SearchJob sj)
{

  var filter = sj.ResultsAsFilter;
  var facetsToSearch = new[] { "Manufacturer", "Age", "NumPlayers" };

  // Configure the WordListBuilder to identify our facets
  var wlb = new WordListBuilder();
  wlb.OpenIndex(_SearchIndexLocation);
  wlb.SetFilter(filter);

  // For each facet or field
  for (var facetCounter = 0; facetCounter < facetsToSearch.Length; facetCounter++)
  {

    // Construct a header for the facet
    var fieldValueCount = wlb.ListFieldValues(facetsToSearch[facetCounter], "", int.MaxValue);
    var thisPanelItem = new HtmlGenericControl("div");
    var header = new HtmlGenericControl("h4");
    header.InnerText = facetsToSearch[facetCounter];
    thisPanelItem.Controls.Add(header);

    // For each matching value in the field
    for (var fieldValueCounter = 0; fieldValueCounter < fieldValueCount; fieldValueCounter++)
    {
      string thisWord = wlb.GetNthWord(fieldValueCounter);
      int thisWordCount = wlb.GetNthWordCount(fieldValueCounter);

      if (string.IsNullOrEmpty(thisWord) || thisWord == "-") continue;

      thisPanelItem.Controls.Add(new HtmlAnchor() { InnerText = string.Format("{0} ({1})", thisWord,
thisWordCount) , HRef = "FacetedSearch.aspx?s=" + txtSearch.Text + "&f=" + facetsToSearch[facetCounter] +
"&t=" + thisWord });
      thisPanelItem.Controls.Add(new HtmlGenericControl("br"));

    }

    pFacets.Controls.Add(thisPanelItem);

  }

}
```

Listing 8 - ExtractFacets method to format facet search criteria

The method starts by configuring a `dtSearch.Engine.WordListBuilder` object to use the same search index location and the results from the previous search. The next lines will traverse the collection of `facetsToSearch` and construct a div with a header and the words found in that field below it as hyperlinks.

`HighlightResult` is the final method to share. This method uses a `dtSearch.Engine.FileConverter` object to read the HTML snippet stored in the index and format it with an HTML SPAN tag to highlight the words that were found from the search textbox.

```
private string HighlightResult(int itemPos)
{

  using (FileConverter fc = new FileConverter())
  {

    fc.SetInputItem(_SearchResults, itemPos);
    fc.OutputFormat = OutputFormats.itAnsiitHTML;
    fc.OutputToString = true;
    fc.OutputStringMaxSize = 200000;
    fc.BeforeHit = "<span class='searchHit'>";
    fc.AfterHit = "</span>";

    fc.Flags = ConvertFlags.dtsConvertGetFromCache | ConvertFlags.dtsConvertInputIsHtml;
    fc.Execute();

    return fc.OutputString.Substring(0, fc.OutputString.IndexOf("</dl>")+5);

  }

}
```

Listing 9 - Applying Term Highlighting to Search results in the HighlightResult method

The `FileConverter` is configured with information to indicate which item in the index I want to present, the format of the output desired, and how to wrap any item that was a search hit. The `dtsConvertGetFromCache` flag is passed in to indicate to the `FileConverter` object that I want the original HTML fragment that I stored earlier in my `GetStreamForProduct` method. On my page, I have a CSS class on page for `searchHit` that changes the font color, adds an underline, and sets a yellow background.

The last bits of the method indicate that the document should be fetched from the Searchindex cache and that it is already formatted as HTML. I strip off any extra bits after the closing HTML DL tag in the return statement.

Results

My search page looks like the following after a search for something like CHESS:

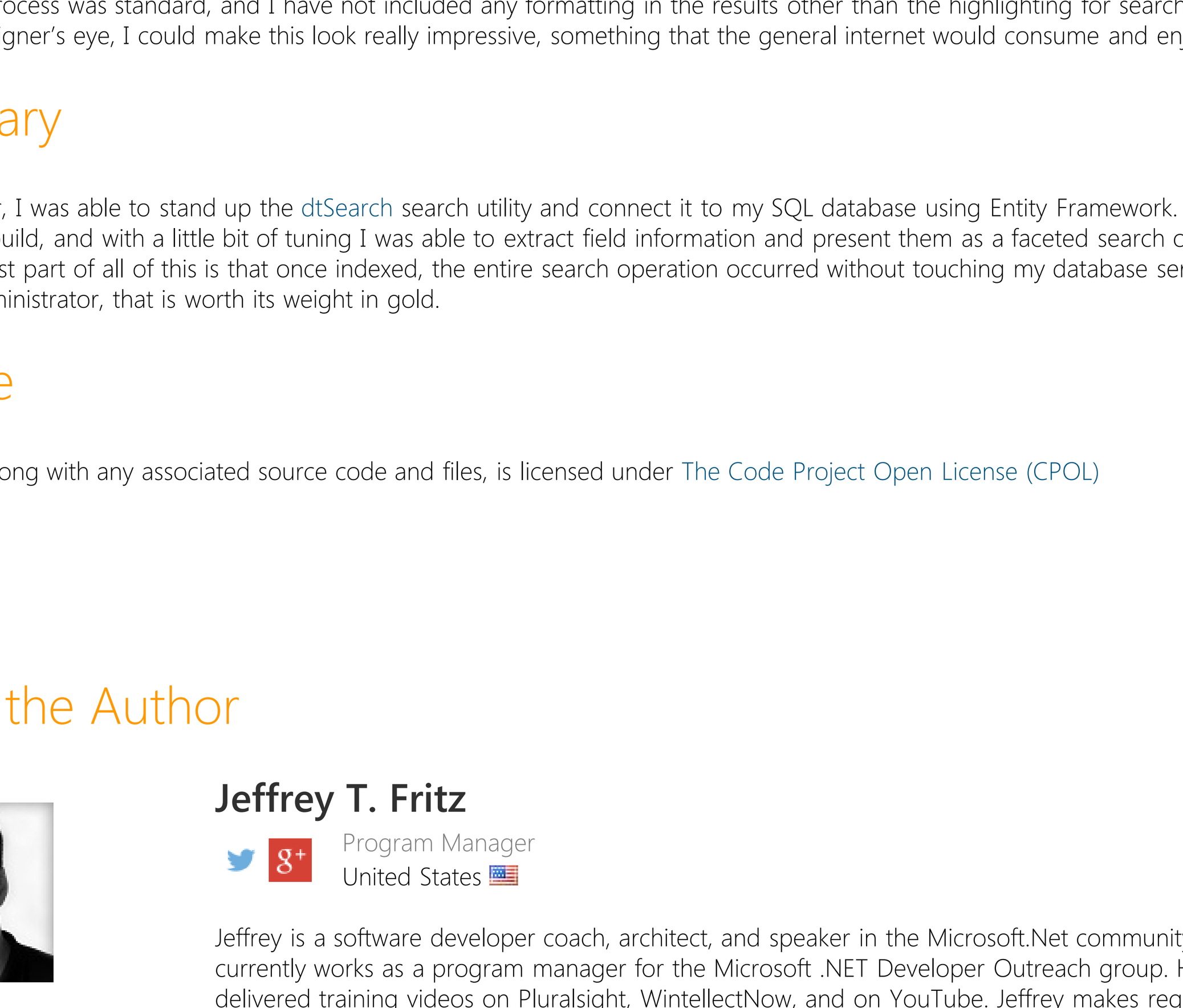


Figure 1 – Search for Chess with facets on the left and Results on the right

This search process was standard, and I have not included any formatting in the results other than the highlighting for search hits. With a little bit a designer's eye, I could make this look really impressive, something that the general internet would consume and enjoy using.

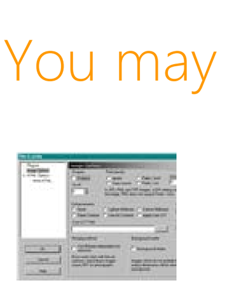
Summary

In short order, I was able to stand up the dtSearch search utility and connect it to my SQL database using Entity Framework. The index was easy to build, and with a little bit of tuning I was able to extract field information and present them as a faceted search option for my users. The best part of all of this is that once indexed, the entire search operation occurred without touching my database server. For my database administrator, that is worth its weight in gold.

License

Share

About the Author



Jeffrey T. Fritz
Program Manager
United States

Jeffrey is a software developer coach, architect, and speaker in the Microsoft.Net community. He currently works as a program manager for the Microsoft .NET Developer Outreach group. He has delivered training videos on Pluralsight, WintellectNow, and on YouTube. Jeffrey makes regular appearances delivering keynotes, workshops, and breakout sessions at conferences such as TechEd, Ignite, DevIntersection, CodeStock, FalafelCon, VSLive as well as user group meetings in an effort to grow the next generation of software developers.

Comments and Discussions

5 messages have been posted for this article Visit **https://www.codeproject.com/Articles/756185/Faceted-Search-with-dtSearch-Not-Your-Average-Sear** to post and view comments on this article, or click **here** to get a print view with messages.