Adding Product Search

*Note: This activity document includes much of the code to complete this activity, but not all.*

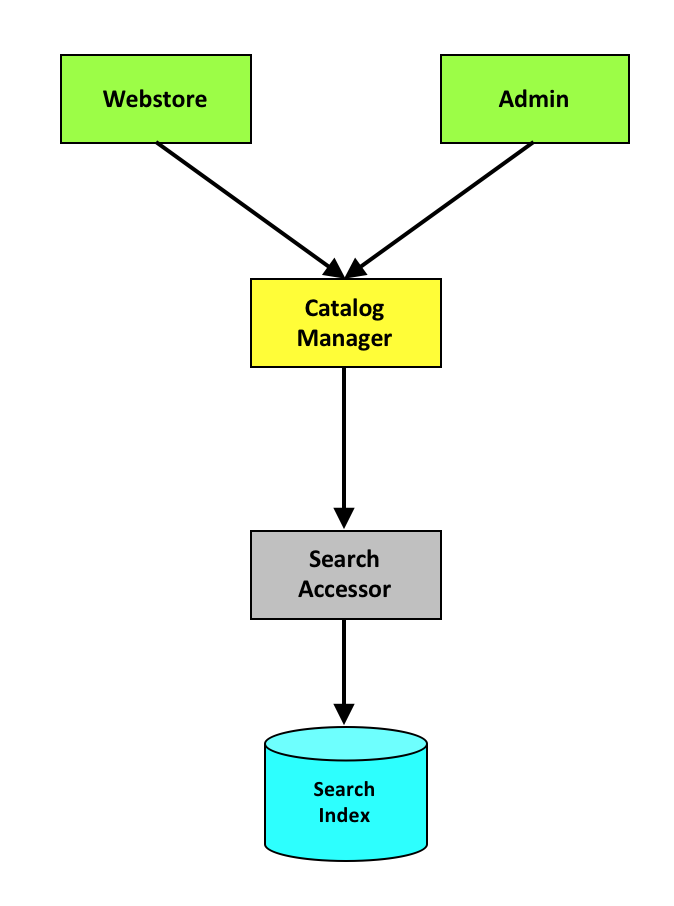
Search is an essential part of any ecommerce application. If you don’t believe me, go to [https://amazon.com](https://amazon.com/). How hard is it to find the search box? Pretty easy right.

If search is essential, we probably need to make a pretty good search, right? Our current ecommerce application doesn’t support search. Now we could use just SQL statements to query the product table, but that won’t be a great option. SQL like statements won’t find misspelled words or do many other things we would expect from a search system.

How should we go about adding search to our application? We could use something like full text search in the database. That might work for very basic use cases, but there are better options out there. For this activity we are going to use Lucene.net to do the searches. Lucene.net is an open source search library based off of a java version called Lucene.

At DPL we have used Lucene on a few projects. The great thing about Lucene is that it is pretty light-weight for a search solution. No services to setup. No web server to setup. No Java to install.

But before we jump into code lets discuss the change we are going to make to the system. We want to add search to our system. Our current architecture doesn't have search, we will need to extend our architecture to support search.



Where do we start? At the beginning, of course.

We will add a NuGet package reference to Lucene.net to our accessors project. Then we will write our search accessor to use Lucene.net. (code snippets below assume Lucene 4.8)

But this might leave us wondering, how do we use Lucene? Luckily, Lucene is pretty easy to use.

We will add some configuration as to where this index should be stored. We will configure this as an environment variable, just like the database connection string. This code is already in place in our reference application, just use the Config.IndexPath variable.

public static string IndexPath

{

get

{

return GetConfigValue("eCommerceIndexPath", "SearchIndex");

}

}

We should start by building out the definition for our SearchAccessor.

public interface ISearchAccessor : IServiceContractBase

{

void RebuildIndex(int catalog);

Product[] Search(int catalogId, string text);

}

We will have to update our dependency injection to support calling the SearchAccessor. You will have to make that change in the AccessorFactory.

The code to build a search index is pretty straight-forward.

public void RebuildIndex(int catalogId)

{

if (System.IO.Directory.Exists(GetIndexDirectory().FullName))

System.IO.Directory.Delete(GetIndexDirectory().FullName, true);

using (var analyzer = new SimpleAnalyzer(LuceneVersion.LUCENE\_48))

using (var indexDir = FSDirectory.Open(GetIndexDirectory()))

{

var config = new IndexWriterConfig(Lucene.Net.Util.LuceneVersion.LUCENE\_48, analyzer);

using (var indexWriter = new IndexWriter(indexDir, config))

{

using (var db = eCommerce.Accessors.EntityFramework.eCommerceDbContext.Create())

{

foreach (var p in db.Products.Where(p => p.CatalogId == catalogId))

{

var doc = new Document();

doc.Add(new Int32Field("Id", p.Id, Field.Store.YES));

doc.Add(new TextField("Name", p.Name, Field.Store.YES));

indexWriter.AddDocument(doc);

}

}

}

}

}

<https://gist.github.com/chadmichel/c425820b8479d7bf222f775a31cdebc9>

Next, we will have to write the actual code to do the searching. This code should be pretty straight-forward too.

public Product[] Search(int catalogId, string text)

{

var result = new List<Product>();

using (var analyzer = new SimpleAnalyzer(LuceneVersion.LUCENE\_48))

using (var indexDirectory = FSDirectory.Open(GetIndexDirectory()))

using(var indexReader = DirectoryReader.Open(indexDirectory))

{

var query = new FuzzyQuery(new Term("Name", text), 2);

var indexSearcher = new IndexSearcher(indexReader);

var searchResults = indexSearcher.Search(query, 10).ScoreDocs;

foreach (var searchResultItem in searchResults)

{

var doc = indexSearcher.Doc(searchResultItem.Doc);

var product = new Product()

{

Id = (int)doc.GetField("Id")?.GetInt32Value(),

Name = doc.GetField("Name")?.GetStringValue()

};

result.Add(product);

}

return result.ToArray();

}

}

<https://gist.github.com/chadmichel/230594873a2948bdd5d42359a2e1c91c>

Once we have those in place we just need to add some unit tests for using Lucene.net, and we should be good to go from a Lucene standpoint. Take a few minutes and write a unit test for the SearchAccessor.

Next, we would need to update our manager level to support calling our new search accessor. We will need to add a new method to our CatalogManager to perform this search, and a method to rebuild the index.

To change CatalogManager we will have to change two interfaces. As part of adding the interfaces below you will have to add some new data types too.

public interface IWebStoreCatalogManager : IServiceContractBase

{

/// <summary>

/// Return the webstore catalog detail and product summary information that is required to show a webstore page

/// </summary>

/// <returns></returns>

WebStoreCatalogResponse ShowCatalog(int catalogId);

/// <summary>

/// Returns the details of a specific product

/// </summary>

/// <param name="catalogId"></param>

/// <param name="productId"></param>

/// <returns></returns>

WebStoreProductResponse ShowProduct(int catalogId, int productId);

/// <summary>

/// Search a webstore catalog

/// </summary>

/// <param name="catalogId"></param>

/// <returns></returns>

WebStoreSearchResponse Search(int catalogId, string query);

}

public interface IAdminCatalogManager : IServiceContractBase

{

/// <summary>

/// Shows the list of catalogs for the authenticated seller

/// </summary>

/// <returns></returns>

AdminCatalogsResponse FindCatalogs();

/// <summary>

/// Shows a specific catalog for the authenticated seller

/// </summary>

/// <returns></returns>

AdminCatalogResponse ShowCatalog(int catalogId);

/// <summary>

/// Updates a specific catalog for the authenticated seller

/// </summary>

AdminCatalogResponse SaveCatalog(WebStoreCatalog catalog);

/// <summary>

/// Shows a specific product for the authenticated seller

/// </summary>

/// <returns></returns>

AdminProductResponse ShowProduct(int catalogId, int productId);

/// <summary>

/// Updates a specific product for the authenticated seller

/// </summary>

AdminProductResponse SaveProduct(int catalogId, Product product);

/// <summary>

/// Rebuild search catalog

/// </summary>

void RebuildCatalog(int catalogId);

}

[DataContract]

public class ProductSearchItem

{

[DataMember]

public int Id { get; set; }

[DataMember]

public string Name { get; set; }

[DataMember]

public decimal Price { get; set; }

}

[DataContract]

public class WebStoreSearchResponse : ResponseBase

{

[DataMember]

public ProductSearchItem[] Products { get; set; }

}

Now go through and implement the two manager methods. They should be pretty basic methods.

public void RebuildCatalog(int catalogId)

{

try

{

AccessorFactory.CreateAccessor<ISearchAccessor>().RebuildIndex(catalogId);

}

catch(Exception ex)

{

Logger.Error(ex);

}

}

WebStore.WebStoreSearchResponse WebStore.IWebStoreCatalogManager.Search(int catalogId, string query)

{

try

{

var products = AccessorFactory.CreateAccessor<ISearchAccessor>().Search(catalogId, query);

var list = new List<WebStore.ProductSearchItem>();

foreach(var product in products)

{

var searchProduct = new WebStore.ProductSearchItem()

{

Id = product.Id,

Name = product.Name,

Price = product.Price

};

list.Add(searchProduct);

}

return new WebStore.WebStoreSearchResponse()

{

Success = true,

Products = list.ToArray(),

};

}

catch (Exception ex)

{

Logger.Error(ex);

return new WebStore.WebStoreSearchResponse() { Success = false };

}

}

We should also add a few unit tests for this code. But for the purpose of this walkthrough, lets write a single integration test.

[TestMethod]

[TestCategory("Managers-WebStore")]

public void CatalogManager\_Search()

{

// ## Arrange ##

// Create managers

var context = new AmbientContext() { SellerId = 1, SessionId = Guid.NewGuid(), AuthToken = "MyToken" };

var webStoreMgr = GetManager<IWebStoreCatalogManager>(context);

var adminMgr = GetManager<DPLRef.eCommerce.Contracts.Admin.Catalog.IAdminCatalogManager>(context);

// Create a catalog using admin manager

var saveCatalogResponse = adminMgr.SaveCatalog(new DPLRef.eCommerce.Contracts.Admin.Catalog.WebStoreCatalog()

{

Name = "integration\_test",

Description = "integration\_test"

});

// Save a product using admin manager

var saveProductResponse = adminMgr.SaveProduct(

saveCatalogResponse.Catalog.Id,

new DPLRef.eCommerce.Contracts.Admin.Catalog.Product()

{

Name = "hello world",

IsAvailable = true,

Price = 10.0m,

});

// ## Act ##

// Rebuild catalog using admin manager

// Search using web store manager

// ## Assert ##

}

Last, we need to update our webstore UI to support our new search interfaes added to our CatalogManager. To accomplish this, we will need to update our program.cs file.

// Rebuild Search calls the admin contract

// I included it here to make working with this code easier.

private static void RebuildSearch(int catalogId)

{

var context = new AmbientContext() { SellerId = 1 };

var managerFactory = new ManagerFactory(context);

var webStoreCatalogManager = managerFactory.CreateManager<Contracts.Admin.Catalog.IAdminCatalogManager>();

webStoreCatalogManager.RebuildCatalog(catalogId);

}

private static void Search(int catalogId)

{

Console.WriteLine("Search Text:");

var query = Console.ReadLine();

var context = new AmbientContext() { SellerId = 1 };

var managerFactory = new ManagerFactory(context);

var webStoreCatalogManager = managerFactory.CreateManager<IWebStoreCatalogManager>();

var response = webStoreCatalogManager.Search(catalogId, query);

ShowResponse(response, StringUtilities.DataContractToJson<ProductSearchItem[]>(response.Products));

}

This activity is a great example of how our architecture could be extended with minimal effort. We were able to extend the system by only changing one existing architecture piece (CatalogManager). And the change to CatalogManger was only an extension. We added a new SearchAccessor that hid the dependency of Lucene. Based off of our implementation, we should be able to change our search technology, without changing anything else in the system.