Adding Product Search

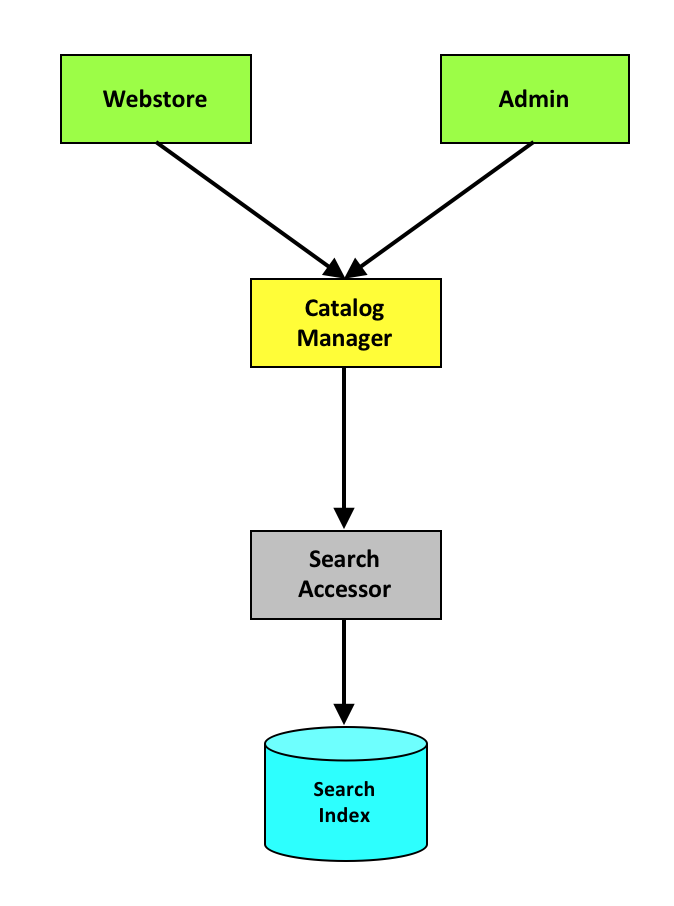
Search is an essential part of any ecommerce application. If you don’t believe me, go to [https://amazon.com](https://amazon.com/). How hard is it to find the search box? Pretty easy right.

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.

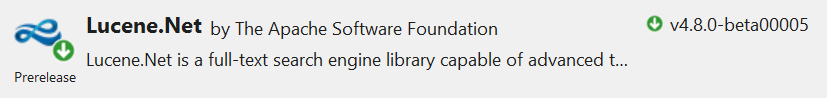
At DPL we have used Lucene on a few projects. The great thing about Lucene is that it is pretty light-weight for a search solution. No services to setup. No web server to setup. No Java to install.

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.

Add a NuGet package reference to Lucene.net to our accessors project. You will need to have “include prerelease” checked to pull the Lucene.Net nuget package that supports .NET Core. You will also need “Lucene.Net.Analysis.Common”.



Configuration for where Lucene should write its index files is already in place with the IndexPath property on the. Config class.

public static string IndexPath

{

get

{

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

}

}

Start by building out the definition for the SearchAccessor. Place this in the Catalog directory in the accessors project.

public interface ISearchAccessor : IServiceContractBase

{

void RebuildIndex(int catalogId);

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

}

Create a SearchAccessor that implements ISearchAccessor.

class SearchAccessor : AccessorBase, ISearchAccessor

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

AddType<ISearchAccessor>(typeof(SearchAccessor));

Implement the RebuildIndex method in SearchAccessor.

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);

}

}

}

}

}

Implement the Search method in SearchAccessor.

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();

}

}

The above code assumes you will have a GetIndexDirectory method in SearchAccessor.

private DirectoryInfo GetIndexDirectory()

{

if (!string.IsNullOrWhiteSpace(Config.IndexPath))

{

var dir = new DirectoryInfo(Config.IndexPath);

if (dir.Exists)

return dir;

}

throw new InvalidOperationException("eCommerceIndexPath is not configured.");

}

Once we have those in place we need to add some unit tests for the SearchAccessor. Take a few minutes and write a unit test for the SearchAccessor.

[TestClass]

public class SearchAccessorTests

{

[TestMethod]

public void SearchAccessor\_Test()

{

// arrange

var accessor = new AccessorFactory(null, null).CreateAccessor<ISearchAccessor>();

// act

accessor.RebuildIndex(1);

var results = accessor.Search(1, "Toyta"); // Toyota

// assert

Assert.AreEqual(10, results.Length);

}

}

Next, update the catalog manager level to support calling our new search accessor. Add a new method to CatalogManager to perform this search, and a method to rebuild the index. These two methods will be broken across two interfaces.

IWebStoreCatalogManager

WebStoreSearchResponse Search(int catalogId, string query);

IAdminCatalogManager

void RebuildCatalog(int catalogId);

This solution will require a few new DTOs. These should be placed in the same directory as the IWebStoreCatalogManager contract. The namespace should be DPLRef.eCommerce.Contracts.WebStore.Catalog.

[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.

void Admin.IAdminCatalogManager.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("Integration-WebStore")]

public void WebStore\_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 = "toyota",

IsAvailable = true,

Price = 10.0m,

});

// ## Act ##

// Rebuild catalog using admin manager

adminMgr.RebuildCatalog(saveCatalogResponse.Catalog.Id);

// Search using web store manager

var result = webStoreMgr.Search(saveCatalogResponse.Catalog.Id, "toyta");

// ## Assert ##

Assert.AreEqual(1, result.Products.Length);

}

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

Around line 110

case 17: // Rebuild Search

RebuildSearch(1);

break;

case 18:

Search(1);  
 break;

Arround line 156

Console.WriteLine("Search =====================");

Console.WriteLine(" 17: Rebuild Search");  
Console.WriteLine(" 18: Search");

Add RebuildSearch and Search methods to class Program.

// 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.

# Bonus

Bonus 1: Fix the price. When running the searches using the console application, we should get a price, but we are always getting $0.00. You will need to update the search accessor to make this fix. Also, note that we are not populating all the fields of the product DTO when calling SearchAccessor.Search. Update the Search method to return a custom DTO.

Bonus 2: The current solution isn’t multi-tenant. All sellers are sharing the same index. Make it so each customer can have their own index. To accomplish you could do many options. The easiest solution is probably placing each catalog’s index into its own directory. While doing this add a unit test that verifies that one catalog’s index doesn’t affect another catalog’s index.