
S#arp Architecture Documentation

Release 2.1rc

S#arp Architecture team

June 01, 2016

1	General	3
1.1	Overview	3
1.2	Frequently Asked Questions	3
1.3	Architecture	5
1.4	Thanks	6
2	Getting started	7
2.1	Installation	7
2.2	Simple CRUD application	7
3	Updating	13
3.1	Updating dependencies via nuget	13
4	Features	15
4.1	NHibernate	15
5	Additional resources	23
5.1	Additional Information	23



If you're getting into S#arp Architecture for the first time, there's a bit of learning and background materials to digest. This is the place to start! Follow the documentation links below from top to bottom to get a firm understanding of S#arp Architecture and to get yourself fully qualified to develop your own S#arp project.

1.1 Overview

Pronounced “Sharp Architecture,” this is a solid architectural foundation for rapidly building maintainable web applications leveraging the ASP.NET MVC framework with NHibernate. The primary advantage to be sought in using any architectural framework is to decrease the code one has to write while increasing the quality of the end product. A framework should enable developers to spend little time on infrastructure details while allowing them to focus their attentions on the domain and user experience. Accordingly, S#arp Architecture adheres to the following key principles:

- Focused on Domain Driven Design
- Loosely coupled
- Preconfigured Infrastructure
- Open Ended Presentation

The overall goal of this is to allow developers to worry less about application “plumbing” and to spend most of their time on adding value for the client by focusing on the business logic and developing a rich user experience.

Absolutely essential reading is Eric Evans’ [Domain Driven Design](#). For a quick introduction to the subject, see [Domain Driven Design Quickly](#) which is a concise summary of Evans’ classic work. Other useful background material, albeit dated, includes [NHibernate Best Practices](#). Although there are major infrastructural changes from the referenced article when compared to the current S#arp Architecture, the general structure is very similar and the background reading is helpful in understanding many of the ideas behind this development foundation.

1.2 Frequently Asked Questions

Q: Is there a continuous integration available for downloading built S#arp Architecture artifacts?

A: Great question! In fact, S#arp Architecture is part of the CI environment hosted at <http://teamcity.codebetter.com>.

Q: How do I register an NHibernate IInterceptor?

A: To register an IInterceptor, simply invoke the registration after initializing NHibernateSession in Global.asax.cs; e.g.,

```
NHibernateSession.Init(new WebSessionStorage(this),  
  
    new string[] { Server.MapPath("~/bin/MyProject.Data.dll") });  
  
NHibernateSession.RegisterInterceptor(new MyAuditLogger());
```

Q: How do I “downgrade” my S#arp Architecture project to not use Fluent NHibernate Auto Mapping?

A: Starting with S#arp Architecture RC, Fluent NHibernate’s auto persistence mapping is supported and included by default. If you have no idea what I’m talking about, take a quick look at <http://www.chrisvandesteeg.nl/2008/12/02/fluent-nhibernates-autopersistencemodel-i-love-it/>. If you don’t want to use the Auto Mapping capabilities, you can use Fluent NHibernate ClassMaps or HBMs as well.

- Delete YourProject.Infrastructure/NHibernateMaps/AutoPersistenceModelGenerator.cs from your project
- Modify YourProject.Web.Mvc/Global.asax.cs to no longer load the auto persistence model and feed it to the NHibernate initialization process. The following example code demonstrates how this may be accomplished:

```
AutoPersistenceModel autoPersistenceModel = new AutoPersistenceModelGenerator().Generate();

NHibernateSession.Init(new WebSessionStorage(this), new string[] { Server.MapPath("~/bin/Northwind.Infrastructure.dll")
});
```

Likewise, within YourProject.Tests/YourProject.Infrastructure/NHibernateMaps/MappingIntegrationTests class, remove the passing of the auto-persistence model to the NHibernate initialization process:

```
[TestFixture]
[Category("DB Tests")]
public class MappingIntegrationTests
{
    [SetUp]
    public virtual void Setup() {
        string[] mappingAssemblies =
            RepositoryTestsHelper.GetMappingAssemblies();

        AutoPersistenceModel autoPersistenceModel =
            new AutoPersistenceModelGenerator().Generate();

        NHibernateSession.Init(new SimpleSessionStorage(),
            mappingAssemblies,
            "../../../app/Northwind.Web.Mvc/NHibernate.config");
    }
}
```

Compile and hope for the best. ;)

Note that Fluent NHibernate Auto Mapper can live in happy coexistence with Fluent NHibernate ClassMaps and HBMs. Simply exclude the applicable classes from the Auto Mapper and add their mapping definition, accordingly. Alternatively, it’s very simple to provide mapping overrides as well, which is the approach I personally prefer to take.

Q: How do I run my S#arp Architecture project in a medium trust environment?

A: If you must run in a Medium Trust environment, the following modifications must be made to the NHibernate configuration:

```
... managed_web ... Additionally, all class mappings need to be set to lazy="false".
```

Q: How do I run my S#arp Architecture project in 64 bit (x64) environment?

A: There are a couple of options for running in a 64 bit environment. The first is to switch IIS7 to have the website run in a classic .NET application pool. Alternatively, you can create a separate app pool and ensure that the “Enable 32 bit application” is checked under the advanced settings for the app pool.

In addition to modifying IIS for a 64 bit environment, also note that that you should modify YourProject.Tests to build as an x86 assembly, as the included SQLite assembly will only work as x86. Alternatively, you could download an x64 compliant version of SQLite.

Q: Because the Entity’s Id property has a protected setter, the Id property isn’t being included during XML serialization? What can we do to include it?

A: Having the Id setter being protected is a fundamental principle of S#arp Architecture. But there are times when the Id is needed to be included during XML serialization. The following base class can be added to YourProject.Core to facilitate the Id being included with XML serialized objects:

```
public class EntityWithSerializableId : Entity
{
    public virtual int ItemId {
        get { return Id; }
        set { ; }
    }
}
```

1.3 Architecture

The project is divided into the following layers:

- Tasks
- Domain
- Infrastructure
- Presentation
- Specs/Tests

Despite their divergent names, the layers and their functions should be familiar to anyone with experience with the MVC pattern.

1.3.1 Tasks

Previous known as “Application Services,” the *Tasks Layer* serves to tie together any non-business logic from a variety of third-party services or persistence technologies. While setup and defining a service such as Twitter would occur in the *Infrastructure Layer*, executing and combining the results with, say, a local NHibernate database, would occur in the *Tasks Layer*. The resulting viewModel or DTO would be sent down to the *Presentation Layer*

Additional occupants

- CommandHandlers
- Commands

1.3.2 Presentation

The *Presentation Layer* contains the familiar MVC project with views, viewmodels and controllers, along with application setup tasks. Previous iterations of Sharp Architecture spun out controllers to a separate application, but as of 2.0 this is no longer the case.

Additional occupants

- ViewModels
- Query Objects

- Controllers
- Views

1.3.3 Infrastructure

The *Infrastructure Layer* setups up third party data sources, along with items such as NHibernate maps. Previously, methods on a repository would exist here, but are superseded by IQuery.

1.3.4 Domain

The *Domain Layer* is where business entities and other business logic resides. The domain layer should be persistence ignorant, with any persistence existing in the *Tasks Layer* or in the *Presentation Layer* (for populating viewModels).

Additional occupants

- Contracts for Tasks
- Contracts for IQuery

1.4 Thanks

Sharp Architecture attempts to represent the combined wisdom of many software development giants. Included patterns and algorithms reflect best practices described by the GoF, Martin Fowler, Uncle Bob Martin, Steve McConnell, many gurus in the blogosphere and other industry leaders. Many have been personally involved with helping to shape Sharp Architecture's current form including and in no particular order: Alec Whittington, Chris Richards, Seif Attar, Jon George, Howard van Rooijen, Billy McCafferty, Frank Laub, Kyle "the coding hillbilly" Baley, Simone Busoli, Jay Oliver, Lee Carter, Luis Abreu, James Gregory, and Martin Hornagold ... along with many others who have asked WTF at all the right times. A special thanks to Roy Bradley who was brave crazy enough to commission Billy to develop the first version 0.1. Finally, none of this would have been possible and/or applicable without the tireless, unpaid efforts of the teams behind projects such as NHibernate, Fluent NHibernate, NHibernate Validator, MvcContrib, and Castle.

Getting started

2.1 Installation

2.1.1 Deploy Sharp Architecture with Templify

Download and install [Templify](#). Make sure to install the latest Sharp Architecture package by going to the Sharp Architecture “Downloads” page and downloading the Templify zip. Extract the contents and run the *.cmd file to install the package.

Next, select the folder you wish to deploy Sharp Architecture. In our case, let’s create a folder and name the project, IceCreamYouScreamCorp. After Templify has finished deploying the package, go ahead and launch the solution in Visual Studio.

2.1.2 Using Nuget

TODO

2.2 Simple CRUD application

Sharp Architecture makes it simple to do CRUD applications right out of the box. While we recommend the use of another framework such as Dynamic Data for CRUD heavy sites, take a look at how to do simple CRUD operations in Sharp Architecture.

Install S#arp Architecture using your preferred method

2.2.1 Create your first entity

Let’s jump back over to our Visual Studio solution. Navigate to your Domain project and define a class called Product that inherits from Entity. All domain objects that are persisted in Sharp inherit from this base class. Entity does many things, but for the purposes of this example we only care about the fact that:

- The property Id is already set for us.
- We are now inheriting a ValidatableObject, which gives us the IsValid() method.

Due to NHibernate, all properties must be marked virtual. We’ll only have one property for this class, a string titled Name. When you’re done, your class should look like this:

```
namespace IceCreamYouScreamCorp.Domain
{
    using System;
    using System.ComponentModel.DataAnnotations;

    using SharpArch.Domain.DomainModel;

    public class Product : Entity
    {
        [Required(ErrorMessage = "Must have a clever name!")]
        public virtual string Name { get; set; }
    }
}
```

2.2.2 Configure NHibernate.config

Before we go any further, navigate to your Mvc project and find NHibernate.config. I'm assuming you're running a local instance of SQL Express with a database named IceCreamDb. You'll need to figure out your own connection string, but once you do, find the "connection.connection_string" tags:

```
<property name="connection.connection_string">Server=localhost\SQLEXPRESS;Initial Catalog=IceCreamDb;
```

2.2.3 Prep your database environment

Before we start we'll need to wire up NHibernate to your database. If you have not already done so, create a database. Let's assume you have SQLExpress running on your local machine and you've created a database called IceCreamDb.

Run the test `IceCreamYouScreamCorp.Tests.SharpArchTemplate.Data.NHibernateMaps.CanGenerateDatabase` which will generate a script for creating the database in the Database folder under the folder you templified.

Alternatively, run the `IceCreamYouScreamCorp.Tests.SharpArchTemplate.Data.NHibernateMaps.CanCreateDatabase` test which will create the tables for you. This test is set to run explicitly to avoid overwriting existing tables.

2.2.4 Create a Controller

Create a controller in your controllers folder named `ProductsController`.

NB: In this example we're injecting repositories into our controllers. This is not recommended best practice. Please see the Cookbook example for abstracting this to your Tasks layer. I think for the purposes of this example, it is much more clear to inject the Repository.

NB + 1: Sharp Architecture uses dependency injection, the "Don't call us, we'll call you," principle. Dependency injection is a bit out of the scope of the article. If this is new to you, I highly recommend looking at [Martin Fowler's article](#).

One of the great things about Sharp Architecture is that it has created a generic repository for you. Generally there's no need to worry about the NHibernate session, or creating a specific repository each time you need to talk to your database. As such, let's create a local field, `private readonly INHibernateRepository<Product> productRepository;` and inject it into our controller:

```
public ProductsController(INHibernateRepository<Product> productRepository)
{
    this.productRepository = productRepository;
}
```

2.2.5 Create ActionResult for our Crud Operations

Now we're all setup let's we'll need to do the following things:

- Return a list of all products
- Return a single product
- Create/Update a single product
- Delete a product

Returning all products on the Index ActionResult:

```
public ActionResult Index()
{
    var products = this.productRepository.GetAll();
    return View(products);
}
```

Return a single product and display it on an editable form:

```
[Transaction]
[HttpGet]
public ActionResult CreateOrUpdate(int id)
{
    var product = this.productRepository.Get(id);
    return View(product);
}
```

Post the result, return the object if it is invalid:

```
[Transaction]
[ValidateAntiForgeryToken]
[HttpPost]
public ActionResult CreateOrUpdate(Product product)
{
    if (ModelState.IsValid && product.IsValid())
    {
        this.productRepository.SaveOrUpdate(product);
        return this.RedirectToAction("Index");
    }

    return View(product);
}
```

Delete a product, making sure we are posting as we are changing data.

```
[Transaction]
[ValidateAntiForgeryToken]
[HttpPost]
public ActionResult Delete(int id)
{
    var product = this.productRepository.Get(id);

    if (product == null)
    {
        return HttpNotFound();
    }

    this.productRepository.Delete(product);
}
```

```
return this.RedirectToAction("Index");
}
```

2.2.6 Add the views

Now all we have to do is create our views for each action. Once this is complete, you can run the application to see it in action.

Index.cshtml:

```
@using IceCreamYouScreamCorp.Web.Mvc
@model IEnumerable<IceCreamYouScreamCorp.Domain.Product>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink((ProductsController c) => c.CreateOrUpdate(0), "Create New")
</p>
<table>
    <tr>
        <th>
            Name
        </th>
        <th></th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.ActionLink("Edit", "CreateOrUpdate", new { id=item.Id })
            </td>
            <td>
                @using (Html.BeginForm("Delete", "Products")) {
                    @Html.Hidden("id", item.Id)
                    <input type="submit" value="Delete" />
                    @Html.AntiForgeryToken()
                }
            </td>
        </tr>
    }
</table>
```

CreateOrUpdate.cshtml:

```
@model IceCreamYouScreamCorp.Domain.Product

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
```

```

<legend>Product</legend>

<div class="editor-label">
    @Html.LabelFor(model => model.Name)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)
</div>

    @Html.HiddenFor(model => model.Id)

    <p>
        <input type="submit" value="Save" />
    </p>
</fieldset>

    @Html.AntiForgeryToken()
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

Done!

Start the web project go to /Products to marvel at your creation.

We've achieved the basics of a CRUD operation, without touching on TDD or some other best practices, but this should get you going very quickly on using Sharp Architecture in your project.

Updating

3.1 Updating dependencies via nuget

In Visual Studio, open the Package Manager Console, and run:

```
Update-Package -Safe
```

This will update all packages to their latest compatible version. You can update a specific package by providing the package name.

If any of the assemblies that have been updated are strongly signed, you would need to add assembly binding redirects to your applications config files (including the test projects). Luckily this can be done easily with nuget, in the Package Manager Console select each project you want to add assembly binding redirects to and run:

```
foreach ($proj in get-project -all) {Add-BindingRedirect}
```

4.1 NHibernate

4.1.1 NHibernate HiLo Generator

Why use HiLo

TODO

Using HiLo Id generation

In your Infrastructure project, under NHibernateMaps:

```
public class PrimaryKeyConvention : IIdConvention
{
    public void Apply(FluentNHibernate.Conventions.Instances.IIdentityInstance instance)
    {
        instance.Column(instance.EntityType.Name + "Id");
        instance.UnsavedValue("0");
        instance.GeneratedBy.HiLo("1000");
    }
}
```

Create the following table:

```
CREATE TABLE [dbo].[hibernate_unique_key] (
    [next_hi] [int] NOT NULL
) ON [PRIMARY]
```

Populate the column with a number to seed and you're done.

4.1.2 Configuration cache

During an application's startup NHibernate can take significant time when configuring and validating it's mapping to a database. Caching the NHibernate configuration data can reduce initial startup time by storing the configuration to a file and avoiding the validation checks that run when a configuration is created from scratch.

SharpArch's NHibernateSession class now has a ConfigurationCache property that accepts an object reference that does the work of loading and saving the NHibernate configuration to a file. This new feature is based on an article by

Oren Eini (aka Ayende Rahien) Building a Desktop To-Do Application with NHibernate, and a big thanks to Sandor Driehuis who provided a lot of this code.

Cache Setup

To use the configuration cache simply set the `NHibernate.Session.ConfigurationCache` property to a new instance of the `NHibernate.ConfigurationFileCache` class, before calling the `NHibernate.Session.Init` method. For example:

```
private void InitializeNHibernateSession()
{
    NHibernate.Session.ConfigurationCache = new NHibernate.ConfigurationFileCache(
        new string[] { "Northwind.Domain" });
    NHibernate.Session.Init(
        webSessionStorage,
        new string[] { Server.MapPath("~/bin/Northwind.Infrastructure.dll") },
        new AutoPersistenceModelGenerator().Generate(),
        Server.MapPath("~/NHibernate.config"));
}
```

The constructor of the `NHibernate.ConfigurationFileCache` class takes a string array of file dependencies, typically this is the `Domain.dll` that contains the web application's domain model objects. The string array can contain just the assembly name (as above) or the file name (ex "Northwind.Domain.dll") or the full path to the file (ex "c:.Domain.dll").

Details

```
INHibernateConfigurationCache
```

Interface that defines two methods for loading and saving the configuration cache.

```
NHibernate.Cfg.Configuration LoadConfiguration(string configKey, string configPath, string[] mappingAssemblies)
void SaveConfiguration(string configKey, NHibernate.Cfg.Configuration config)
```

These methods are used by the `NHibernate.Session.AddConfiguration` method to load and save a configuration object to and from a cache file. This interface allows others to implement their own file caching mechanism if necessary.

```
NHibernateConfigurationCache
```

This class implements the interface and does the work of caching the configuration. Several methods are virtual so they can be overridden in a derived class, as may be necessary to store the cache file in a different location or to have different logic to invalidate the cache.

The constructor takes an string array of file dependencies which are used to test if the cached NHibernate configuration is current or not. If any of the dependent file's last modified time stamp is later than that of the cached configuration, then the cached file is discarded and a new configuration is created from scratch. This configuration is then serialized and saved to a file.

Note: NHibernate's XML config file and the mapping assemblies (ex "Northwind.Data.dll") are automatically included when testing if the cached configuration is current.

FileCache

The `SharpArch.Domainproject` now contains a `FileCache` class that can serialize and deserialize an object to a file in binary format. This class uses a generic type parameter to define the type being serialized and deserialized. This makes the `FileCache` class useful for any other object that you might want to serialize to a file.

ConfigurationCache property

This property holds a reference to an INHibernateConfigurationCache object, and if not null, will use the caching logic to load and save the NHibernate configuration. You must set the ConfigurationCache property before calling the NHibernateSession.Init method, and you cannot change the property reference after calling the Init method.

Configuration Serialization

To cache the configuration to a file, all objects contained within the NHibernate configuration must be serializable. All of the default data types included with NHibernate will serialize, but if you have any custom data types (i.e. classes that implement IUserType), they must also be marked with the [Serializable] attribute and, if necessary, implement ISerializable.

4.1.3 Multiple Databases

To add support for an additional database to your project:

Create an NHibernateForOtherDb.config file which contains the connection string to the new database

Within YourProject.Web.Mvc/Global.asax.cs, immediately under the NHibernateSession.Init() to initialize the first session factory, an additional call to NHibernateSession.AddConfiguration(). While the first NHibernate initialization assumes a default factory key, you'll need to provide an explicit factory key for the 2nd initialization. This factory key will be referred to by repositories which are tied to the new database as well; accordingly, I'd recommend making it a globally accessible string to make it easier to refer to in various locations. E.g.,

```
NHibernateSession.AddConfiguration(Northwind.Infrastructure.DataGlobals.OTHER_DB_FACTORY_KEY,
    new string[] { Server.MapPath("~/bin/Northwind.Data.dll") },
    new AutoPersistenceModelGenerator().Generate(),
    Server.MapPath("~/NHibernateForOtherDb.config"), null, null, null);

// In DataGlobals.cs:
public const string OTHER_DB_FACTORY_KEY = "nhibernate.other_db";
```

Create an Entity class which you intend to be tied to the new database; e.g.,

```
public class Village : Entity
{
    public virtual string Name { get; set; }
}
```

Within YourProject.Core/DataInterfaces, add a new repository interface for the Entity class and simply inherit from the base repository interface. Since you'll need to decorate the concrete repository implementation with the factory session key, you need to have the explicit interface to then have the associated concrete implementation; e.g.,

```
using SharpArch.Core.PersistenceSupport;

namespace Northwind.Domain.DataInterfaces
{
    public interface IVillageRepository : IRepository<Village> { }
}
```

Within YourProject.Data, add a new repository implementation for the repository interface just defined; e.g.,

```
using Northwind.Core.DataInterfaces;
using SharpArch.Data.NHibernate;
using Northwind.Core;
```

```
namespace Northwind.Infrastructure
{
    [SessionFactory(DataGlobals.OTHER_DB_FACTORY_KEY)]
    public class VillageRepository :
        Repository<Village>, IVillageRepository { }
}
```

Within YourProject.Web.Mvc.Controllers, add a new controller which will use the repository (or which will accept an application service which then uses the repository); e.g.,

```
using System.Web.Mvc;
using SharpArch.Web.NHibernate;
using Northwind.Core.DataInterfaces;
using SharpArch.Core;
using Northwind.Core;
using System.Collections.Generic;

namespace Northwind.Web.Mvc.Controllers
{
    public class VillagesController : Controller
    {
        public VillagesController(
            IVillageRepository villageRepository) {
            Check.Require(villageRepository != null,
                "villageRepository may not be null");
            this.villageRepository = villageRepository;
        }

        [Transaction(DataGlobals.OTHER_DB_FACTORY_KEY)]
        public ActionResult Index() {
            IList<Village> villages = villageRepository.GetAll();
            return View(villages);
        }

        private IVillageRepository villageRepository;
    }
}
```

Create an index page to list the villages given to it from the controller.

Note that the WCF support built into the SharpArch libraries does NOT support multiple databases at this time. WCF will always use the “default” database; the one which is configured without an explicit session factory key.

4.1.4 S#arp Architecture with NHibernate.Search

Note: Do not use this with NHibernate Configuration Cache:

```
NHibernateSession.ConfigurationCache = new NHibernateConfigurationFileCache();
```

NuGet Install

Install-Package NHibernate.Search

(This used to be a big deal, building from source, etc.)

Configuring and Building the Index

Maintaining an index in Lucene is easy. There's a configuration setting to keep any deletes, inserts or updates current in the index, and a way to build the index from an existing dataset from scratch. You usually need both unless you're starting on a project from scratch, but even so, it is nice to have a way to build an index in case the existing index gets corrupted.

This example will assume you've created a folder in the root of your MVC project, and have given the ASP.NET security guy full permissions.

Open up your root Web.Config, add the following right above :

Right below add the following:

```
<nhs-configuration xmlns='urn:nhs-configuration-1.0'>
  <search-factory>
    <property name="hibernate.search.default.indexBase">~\LuceneIndex</property>
  </search-factory>
</nhs-configuration>
```

Navigate to your NHibernate.Config, before add:

```
<listener class='NHibernate.Search.Event.FullTextIndexEventListener, NHibernate.Search' type='post-in
<listener class='NHibernate.Search.Event.FullTextIndexEventListener, NHibernate.Search' type='post-up
<listener class='NHibernate.Search.Event.FullTextIndexEventListener, NHibernate.Search' type='post-de
```

Add a Search Repository

In "Northwind.Infrastructure" add a repository called "ContractRepository.cs" (the example assumes you have an object you want to search over called contract):

```
public interface IContractRepository
{
    IList<Contract> Query(string q)
    void BuildSearchIndex();
}
```

Let's add a method to this repository that will create the initial index, if an index already exists, it will be deleted. We'll iterate through all the Suppliers to accomplish this:

```
public void BuildSearchIndex()
{
    FSDirectory entityDirectory = null;
    IndexWriter writer = null;

    var entityType = typeof(Contract);

    var indexDirectory = new DirectoryInfo(GetIndexDirectory());

    if (indexDirectory.Exists)
    {
        indexDirectory.Delete(true);
    }

    try
    {
        var dir = new DirectoryInfo(Path.Combine(indexDirectory.FullName, entityType.Name));
        entityDirectory = FSDirectory.Open(dir);
        writer = new IndexWriter(entityDirectory, new StandardAnalyzer(Lucene.Net.Util.Version.LUCENE
```

```

    }
    finally
    {
        if (entityDirectory != null)
        {
            entityDirectory.Close();
        }

        if (writer != null)
        {
            writer.Close();
        }
    }

    var fullTextSession = Search.CreateFullTextSession(this.Session);

    // Iterate through contracts and add them to Lucene's index
    foreach (var instance in Session.CreateCriteria(typeof(Contract)).List<Contract>())
    {
        fullTextSession.Index(instance);
    }
}

private static string GetIndexDirectory()
{
    INHSConfigCollection nhsConfigCollection = CfgHelper.LoadConfiguration();
    string property = nhsConfigCollection.DefaultConfiguration.Properties["hibernate.search.default..."];
    var fi = new FileInfo(property);
    return Path.Combine(AppDomain.CurrentDomain.BaseDirectory, fi.Name);
}

```

Finally, we'll add a method to query the index:

```

private static string GetIndexDirectory()
{
    INHSConfigCollection nhsConfigCollection = CfgHelper.LoadConfiguration();
    string property = nhsConfigCollection.DefaultConfiguration.Properties["hibernate.search.default..."];
    var fi = new FileInfo(property);
    return Path.Combine(AppDomain.CurrentDomain.BaseDirectory, fi.Name);
}

```

Add Search Controller

```

namespace Northwind.Web.Controllers
{
    private readonly IContractRepository contractRepository;

    public class SearchController : Controller
    {
        public LuceneSupplierController(IContractRepository contractRepository)
        {
            this.contractRepository = contractRepository;
        }

        public ActionResult BuildSearchIndex()
        {
            contractRepository.BuildSearchIndex();
        }
    }
}

```



```
        return RedirectToAction("Index", "Home");
    }

    public ActionResult Search(string query)
    {
        List<Contract> Contracts = contractRepository.Query(query).ToList();
        return View(Contracts);
    }
}
}
```

- Wire up a view to display the search results
- Navigate to localhost:portnumber/ContractController/BuildSearchIndex
- This will (quickly) build your index, it would be beneficial to pass status messages here
- You should see a Suppliers folder in the LuceneIndex folder of the project
- To verify the index, download Luke and point it to the LuceneIndex

Pre-Requisite Reading

I really recommend Hibernate Search in Action, you can really make queries do some neat things that aren't covered in this tutorial. It will, however, get you up and running quickly.

Additional resources

5.1 Additional Information

5.1.1 Blog posts

- [Sharp Architecture Contrib](#)
- [Sharp Architecture Cookbook](#)
- [\[\[NHibernate Configuration Cache\]\]](#)
- [Using MVCScaffolding with Sharp Architecture](#)

5.1.2 Multimedia

Videos? podcasts?

5.1.3 Must reads

TODO Links to pdfs, castle documentation, nhibernate documentation, ddd

5.1.4 Recommended tools and libraries

TODO dropkick, roundhouse, resharper, nhibernate profiler,