S#arp Architecture Documentation

Release 2.1rc

S#arp Architecture team

Jun 28, 2017

Contents

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.

CHAPTER 1

General

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.

Acknowledgments

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 S#arp Architecture's current form including and in no paticular 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.

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.Infrastruction/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 { ; }
    }
}
```

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.

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*

Starting with version 4 Sharp Architecture does not contains support for Tasks. Use MediatR or similiar library instead.

Additional occupants

- EventHandlers
- CommandHandlers
- Commands

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: These can live in the Presentation or in the Tasks project, depending on whether your tasks layer will be returning ViewModels which is not usually the case as ViewModels are tied to a specific view.
- Query Objects: These can live in the Presentation or in the Tasks project, depending on whether you need the queries in your tasks layer.
- Controllers
- Views

Infrastructure

The *Infrastructure Layer* setups up third party data sources, along with items such as NHibernate maps. You can extend the repository implementation with additional methods to perform specific queries, but it is recommended to write your own Query Objects as shown in the cookbook.

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
- Events that are emitted by your domain

CHAPTER 2

Getting started

Installation

Using NuGet

ASP.NET MVC

Install following packages:

- SharpArch.Web.Mvc
- SharpArch.Web.Mvc.Castle if Castle.Windsor is your container of choice.

See TradisBank.Web.Mvc/CastleWindsor/ and TradisBank.Web.Mvc/Global.asax.cs for more details on how configure and initialize IoC.

ASP.NET WebAPI

- SharpArch.Web.Http
- SharpArch.Web.Http.Castle if Castle.Windsor is your container of choice.

Deploy Sharp Architecture with Templify

Templify is not supported anymore.

> 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 required, currently there are templates for NHibernate and RavenDB.
>
> 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.

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

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; }
    }
}
```

Configure NHibernate.config

You can skip this step if you are using RavenDB persistance.

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;Integrated Security=SSPI;</property>
```

Prep your database environment

You can skip this step if you are using RavenDB persistance.

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.CanGenerateDatabaseSchema` 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 explictly to avoid overwriting existing tables.

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;
}
```

Create ActionResults 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");
}
```

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>

        <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.

CHAPTER 3

Updating

Updating

Notes

These are general notes on updating, for specfic notes on updating check the instructions for the version you are updating to below.

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

```
Update-Package -Safe
```

Will only update packages where Major and Minor versions match what you have installed, which should work for patch updates (i.e. from 2.0.0 to 2.0.4) but will not update minor or major releases.

If any of the assemblies that have been updated are strongly named (most are unfortunatley), 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 run:

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

2.0 to 2.1

You can pick and choose which of those packages you want updates, if you just want to update NHibernate, then just run the command with SharpArhc.NHibernate

```
Update-Package SharpArch.Web.Mvc.Castle
Update-Package SharpArch.NHibernate
Update-Package SharpArch.Testing.NUnit
Update-Package SharpArch.Domain
```

If you get problems with Iesi.Collections version mismatch, this is because versions of NHibernate < 3.3.1 didn't specify the highest version of Iesi.Collections, causing problems now that there is a major version change released. To

fix this, add allowedVersions=(,4.0) as an attirbute to the Iesi.Collections element to any packages.config file that has it.

CHAPTER 4

Features

NHibernate

NHibernate Transaction attribute

TransactionAttribute implements Unit of work pattern for ASP.NET MVC and ASP.NET WebAPI controller.

Please refer to TardisBank sample for details.

TODO: update documentation.

NHibernate HiLo Generator

Why use HiLo

HiLo idnentity generatos allows NHibernate to generate identity values to map child objects to their parent without having to hit the database as opposed to using the native identity generation causes nhibernate to hit the database on every save, which affects performance and running of batch statements.

For more information on generator behaviours refer to this blog post by Fabio Maulo.

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.

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 provides interface INHibernateConfigurationCache and NHibernateConfigurationFileCache class which caches configuration in a file under system TEMP folder.

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 Drieenhuizen who provided a lot of this code.

Cache Setup

To use the configuration cache provide cache implementation to UseConfigurationCache() method of the NHibernate-SessionFactoryBuilder class:

For example:

```
ISessionFactory sessionFactory = new NHibernateSessionFactoryBuilder()
  .AddMappingAssemblies(new[] { HostingEnvironment.MapPath(@"~/bin/Suteki.TardisBank.
↪Infrastructure.dll") })
  .UseAutoPersistenceModel(new AutoPersistenceModelGenerator().Generate())
  .UseConfigFile(HostingEnvironment.MapPath("~/NHibernate.config"))
  .UseConfigurationCache(new NHibernateConfigurationFileCache())
  .BuildSessionFactory();
```

Details

```
INHibernateConfigurationCache
```

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

```
NHibernate.Cfg.Configuration LoadConfiguration([NotNull] string configKey, string␣
↪configPath,
      [NotNull] IEnumerable<string> mappingAssemblies);

void SaveConfiguration([NotNull] string configKey, [NotNull] NHibernate.Cfg.
↪Configuration config);
```

These methods are used by the NHibernateSession.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.

```
NHibernateConfigurationFileCache
```

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 optional 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.Domain project 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.

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.

Multiple Databases

Version 4

Note: Due to refactoring of NHibernate session management S#arch v4.0 does not support multiple databases.

This feature will be added in 4.1.

Version 3

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 refere to in various locations. E.g.,

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

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.,

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

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.,

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.

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-insert'/>
<listener class='NHibernate.Search.Event.FullTextIndexEventListener, NHibernate.Search
↪' type='post-update'/>
<listener class='NHibernate.Search.Event.FullTextIndexEventListener, NHibernate.Search
↪' type='post-delete'/>
```

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_29), true, IndexWriter.MaxFieldLength.UNLIMITED);
    }
    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.indexBase"];
    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.indexBase"];
    var fi = new FileInfo(property);
    return Path.Combine(AppDomain.CurrentDomain.BaseDirectory, fi.Name);
}
```

Add Search Controller

```
    namespace Northwind.Web.Controllers
    {
            private readonly IContractRepositorycontractRepository;

            public class SearchController : Controller
            {
                    public␣
↪LuceneSupplierController(IContractRepositorycontractRepository)
                    {
                            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.

RavenDB

RavenDB support was added in version 2.1, for sample usage checkout the SharpArch.TardisBank solution which is a fork of Mike Hadlow's Suteki.TardisBank that was converted to use S#arp Architecture with NHibernate and then converted to work with S#arp Architecture's RavenDB.

The repostiory shows usage of various features of S#arp Architecture, and is a work in progress for a good sample project.

You can start with RavenDB by following the installation steps and using the RavenDB templify template from S#arp Architecture releases.

The RavenDB feature provides generic repositories for use with RavenDB and a UnitOfWork attribute, you can learn more about the UnitOfWorkAttribute by following the link below.

UnitOfWork attribute

With the RavenDB client, changes are not persisted until SaveChanges is called on the RavenDB session. With S#arp Architecture's implementaion of the RavenDB repository, the changes are not persisted on each call, in order to allow for multiple changes to happen per request with all of them being atomic, so that a later error within the request would not leave you application in an inconsitant state.

The UnitOfWorkAttribute should be added to all controller actions where changes are being made to RavenDB, otherwise the changes would not be persisted (unless you get hold of the session and call save changed youself).

RavenDB provides does provide DTC transactions, but the UnitOfWork attribute only uses RavenDB's standard transactions based on the SaveChanges call.

If you want more information about transactions in RavenDB, refer to RavenDB transaction support page.

CHAPTER 5

Additional resources

Additional Information

Asking Questions

Join the google group and ask questions there or post your questions to stackoverflow with the tag 'sharp-architecture' and any other tags for projects your question might be related to (e.g. fluent-nhibernate, nhibernate, castle-windsor)

Related Projects

- Sharp Architecture Contrib
- Sharp Architecture Cookbook
- Castle Windsor
- NHibernate
- FluentNHibernate

Blog posts

- Enhanced Query Objects With S#arp Architecture

Multimedia

- Introduction to S#arp Architecture
- Another look at Sharp Architecture: Validation, Design Decisions and Automapping

Must reads

- Domain Driven Design Quickly free pdf
- Domain Driven Design
- Clean code
- Enterprise Integration Patterns
- Refactoring
- Test Driven Development
- Continuos Delivery
- And many more...

Recommended tools and libraries

- **Deployment and configuration management**
  - OpsCode Chef
  - Puppet
  - dropkick
  - Octopus Deploy
  - Rounhouse DB Migrations
- **Profilers and productivity**
  - Resharper
  - Ants Performance Profiler
  - Ants Memory Profiler
  - NHibernate Profiler

Contributing

- Contribution guidlines on S#arp Architecture github