Log in / create account

Fluent NHibernate wiki

Getting started

From Fluent NHibernate

Contents

Introduction

Fluent NHibernate in a Nutshell

Fluent NHibernate offers an alternative to NHibernate's standard XML mapping files. Rather than writing XML documents (.hbm.xml files), Fluent NHibernate lets you write mappings in strongly typed C# code. This allows for easy refactoring, improved readability and more concise code.

Fluent NHibernate also has several other tools, including:

- Auto mappings - where mappings are inferred from the design of your entities
- Persistence specification testing - round-trip testing for your entities, without ever having to write a line of CRUD
- Full application configuration with our Fluent configuration API
- Database configuration - fluently configure your database in code

Fluent NHibernate is external to the NHibernate Core (http://nhforge.org/media/p/4.aspx) , but is fully compatible with NHibernate version 2.1, and is experimentally compatible with NHibernate trunk.

Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

Background

NHibernate (http://nhforge.org/) is an Object Relational Mapping (http://en.wikipedia.org/wiki/Object-relational_mapping) framework, which (as ORM states) maps between relational data and objects. It defines its mappings in an XML format called HBM, each class has a corresponding HBM XML file that maps it to a particular structure in the database. It's these mapping files that Fluent NHibernate (http://fluentnhibernate.org) provides a replacement for.

Why replace HBM.XML? While the separation of code and XML is nice, it can lead to several undesirable situations.

- Due to XML not being evaluated by the compiler, you can rename properties in your classes that aren't updated in your mappings; in this situation, you wouldn't find out about the breakage until the mappings are parsed at runtime.
- XML is verbose; NHibernate has gradually reduced the mandatory XML elements, but you still can't escape the verbosity of XML.
- Repetitive mappings - NHibernate HBM mappings can become quite verbose if you find yourself specifying the same rules over again. For example if you need to ensure all `string` properties mustn't be nullable and should have a length of 1000, and all `int`s must have a default value of -1.

How does Fluent NHibernate counter these issues? It does so by moving your mappings into actual code, so they're compiled along with the rest of your application; rename refactorings will alter your mappings just like they should, and the compiler will fail on any typos. As for the repetition, Fluent NHibernate has a conventional configuration system, where you can specify patterns for overriding naming conventions and many other things; you set how things should be named once, then Fluent NHibernate does the rest.

Simple example

Here's a simple example so you know what you're getting into.

Traditional HBM XML mapping

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
  namespace="QuickStart" assembly="QuickStart">

  <class name="Cat" table="Cat">
    <id name="Id">
      <generator class="identity" />
    </id>

    <property name="Name">
      <column name="Name" length="16" not-null="true" />
    </property>
    <property name="Sex" />
    <many-to-one name="Mate" />
    <bag name="Kittens">
      <key column="mother_id" />
        <one-to-many class="Cat" />
      </bag>
  </class>
</hibernate-mapping>
```

Fluent NHibernate equivalent

```
public class CatMap : ClassMap<Cat>
{
  public CatMap()
  {
    Id(x => x.Id);
    Map(x => x.Name)
      .Length(16)
      .Not.Nullable();
    Map(x => x.Sex);
    References(x => x.Mate);
    HasMany(x => x.Kittens);
  }
}
```

Installation

Binaries

We produce binaries of our source-tree and compiled assemblies every time anyone commits. These are recommended if you're unable to use Git or Subversion

> Git and Subversion repository information and downloads list is available on the main downloads page (http://fluentnhibernate.org/downloads) , and will be updated whenever new releases are available.

Getting the source

Our source-control is Git (http://git-scm.com/) using Github (http://github.com) , and you have two options available. If you plan on modifying Fluent NHibernate and contributing back to us, then you're best forking our repository and sending us a pull request. You can find out more about that on the github guides site: forking (http://github.com/guides/fork-a-project-and-submit-your-modifications) and pull requests (http://github.com/guides/pull-requests) . The other option is a direct clone of our repository, but that will leave you in an awkward state if you ever do plan to contribute (but it's fine if you don't).

Our repository is located at: http://github.com/jagregory/fluent-nhibernate

Again, we recommend forking our repository; if you don't want to do that, you can do a direct clone:

```
git clone git://github.com/jagregory/fluent-nhibernate.git
```

Once you've got a copy on your local machine, there are two ways you can do a build.

- *If you have Ruby (http://www.ruby-lang.org) installed:* The first time you build, you should run `InstallGems.bat` from the Fluent NHibernate root directory; this makes sure your machine has all the gems (http://www.rubygems.org/) required to build successfully. Once that's completed, and for all subsequent builds, you need to run `Build.bat`; this batch file runs our Rake (http://rake.rubyforge.org/) script which builds the standard NHibernate 2.0 version and outputs it in the `build` directory. For more options for building, see the details of our rake script.
- *Without Ruby:* Open up the `src\FluentNHibernate.sln` solution in Visual Studio and do a build, you can optionally run the tests too.

Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

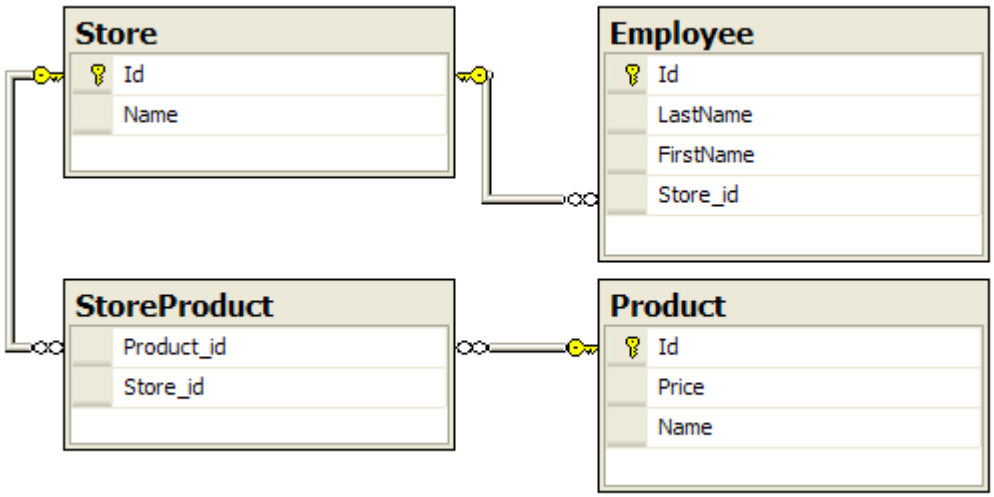
Now that you've got Fluent NHibernate built, you just need to reference the `FluentNHibernate.dll` assembly in your project.

Your first project

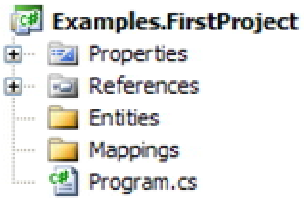
> All the source can be found in the main Fluent NHibernate solution, in the Example.FirstProject (http://github.com/jagregory/fluent-nhibernate/tree/master/src/Examples.FirstProject) project.

You need to have Fluent NHibernate already downloaded and compiled to follow this guide, if you haven't done that yet then please refer to the installation section above.

For this example we're going to be mapping a simple domain for a retail company. The company has a couple of stores, each with products in (some products are in both stores, some are exclusive), and each with employees. In database terms, that's a `Store`, `Employee`, and `Product` table, with a many-to-many table between `Store` and `Product`.



First, create a console application and reference the `FluentNHibernate.dll` you built earlier, and whichever version of `NHibernate.dll` you built against (if you're unsure, just use the one that's output with `FluentNHibernate.dll`); also, because for this example we're going to be using a SQLite (http://www.sqlite.org/) database, you'll need the System.Data.SQLite (http://sourceforge.net/projects/sqlite-dotnet2) library which is distributed with Fluent NHibernate.



Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

You can see the project structure that I used to the left. The `Entities` folder is for your actual domain objects, while the `Mappings` folder is where we're going to put your fluent mapping classes.

For the rest of this guide I'm going to assume you've used the same structure.

Entities

Now that we've got our project setup, let's start by creating our entities. We've got three tables we need to map (we're ignoring the many-to-many join table right now), so that's one entity per table. Create the following classes in your `Entities` folder.

```
public class Employee
{
  public virtual int Id { get; private set; }
  public virtual string FirstName { get; set; }
  public virtual string LastName { get; set; }
  public virtual Store Store { get; set; }
}
```

Entities/Employee.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Entities/Employee.cs)

Our `Employee` entity has an `Id`, the person's name (split over `FirstName` and `LastName`), and finally a reference to the `Store` that they work in.

There's two things that may stand out to you if you're unfamiliar with NHibernate. Firstly, the `Id` property has a private setter, this is because it's only NHibernate that should be setting the value of that `Id`. Secondly, all the properties are marked `virtual`; this is because NHibernate creates "proxies" of your entities at run time to allow for lazy loading, and for it to do that it needs to be able to override the properties.

```
public class Product
{
  public virtual int Id { get; private set; }
  public virtual string Name { get; set; }
  public virtual double Price { get; set; }
  public virtual IList<Store> StoresStockedIn { get; private set; }

  public Product()
  {
    StoresStockedIn = new List<Store>();
  }
}
```

Entities/Product.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Entities/Product.cs)

`Product` has an `Id`, it's `Name`, `Price`, and a collection of `Stores` that stock it.

```
public class Store
{
  public virtual int Id { get; private set; }
  public virtual string Name { get; set; }
  public virtual IList<Product> Products { get; set; }
  public virtual IList<Employee> Staff { get; set; }
```

Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

```
  public Store()
  {
    Products = new List<Product>();
    Staff = new List<Employee>();
  }

  public virtual void AddProduct(Product product)
  {
    product.StoresStockedIn.Add(this);
    Products.Add(product);
  }

  public virtual void AddEmployee(Employee employee)
  {
    employee.Store = this;
    Staff.Add(employee);
  }
}
```

Entities/Store.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Entities/Store.cs)

Finally, we've got our `Store` entity. This has an `Id` and a `Name`, along with a collection of `Products` that are stocked in it, as well as a collection of `Employees` (in the `Staff` list) that work there. This entity has a little bit of logic in it to make our code simpler, that's the `AddProduct` and `AddEmployee` methods; these methods are used to add items into the collections, and setup the other side of the relationships.

> If you're an NHibernate veteran then you'll recognise this; however, if it's all new to you then let me explain: in a relationship where both sides are mapped, NHibernate needs you to set both sides before it will save correctly. So as to not have this extra code everywhere, it's been reduced to these extra methods in `Store`.

Mappings

Now that we've got our entities created, it's time to map them using Fluent NHibernate. We'll start with the simplest class, which is `Employee`. All the following mappings should be created inside the `Mappings` folder.

To map an entity, you have to create a dedicated mapping class (typically this follows the naming convention of \EntityNameMap), so we'll create an `EmployeeMap` class; these mapping classes have to derive from `ClassMap<T>` where `T` is your entity.

```
public class EmployeeMap : ClassMap<Employee>
{

}
```

Mappings/EmployeeMap.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Mappings/EmployeeMap.cs)

The mappings themselves are done inside the constructor for `EmployeeMap`, so we need to add a constructor and write the mappings.

Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

```
public class EmployeeMap : ClassMap<Employee>
{
  public EmployeeMap()
  {
    Id(x => x.Id);
  }
}
```

Mappings/EmployeeMap.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Mappings/EmployeeMap.cs)

To start with we've mapped the `Id` column, and told Fluent NHibernate that it's actually an identifier. The `x` in this example is an instance of `Employee` that Fluent NHibernate uses to retrieve the property details from, so all you're really doing here is telling it which property you want your Id to be. Fluent NHibernate will see that your `Id` property has a type of `int`, and it'll automatically decide that it should be mapped as an auto-incrementing identity in the database - handy!

> For NHibernate users, that means it automatically creates the `generator` element as `identity`.

Let's map the rest of `Employee`.

```
public class EmployeeMap : ClassMap<Employee>
{
  public EmployeeMap()
  {
    Id(x => x.Id);
    Map(x => x.FirstName);
    Map(x => x.LastName);
    References(x => x.Store);
  }
}
```

Mappings/EmployeeMap.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Mappings/EmployeeMap.cs)

There are a couple of new methods we've used there, `Map` and `References`; `Map` creates a mapping for a simple property, while `References` creates a many-to-one relationship between two entities. In this case we've mapped `FirstName` and `LastName` as simple properties, and created a many-to-one to `Store` (many `Employees` to one `Store`) through the `Store` property.

> NHibernate users: `Map` is equivalent to the `property` element and `References` to `many-to-one`.

Let's carry on by mapping the `Store`.

```
public class StoreMap : ClassMap<Store>
{
  public StoreMap()
  {
    Id(x => x.Id);
    Map(x => x.Name);
    HasMany(x => x.Staff)
      .Inverse()
      .Cascade.All();
    HasManyToMany(x => x.Products)
```


Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

```
    .Cascade.All()
    .Table("StoreProduct");
  }
}
```

Mappings/StoreMap.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Mappings/StoreMap.cs)

Again, there's a couple of new calls here. If you remember back to `Employee`, we created a many-to-one relationship with `Store`, well now that we're mapping `Store` we can create the other side of that relationship. So `HasMany` is creating a one-to-many relationship with `Employee` (one `Store` to many `Employees`), which is the other side of the `Employee.Store` relationship. The other new method is `HasManyToMany`, which creates a many-to-many relationship with `Product`.

You've also just got your first taste of the fluent interface Fluent NHibernate provides. The `HasMany` method has a second call directly from it's return type (`Inverse()`), and `HasManyToMany` has `Cascade.All()` and `Table`; this is called method chaining, and it's used to create a more natural language in your configuration.

1. `Inverse` on `HasMany` is an NHibernate term, and it means that the other end of the relationship is responsible for saving.
2. `Cascade.All` on `HasManyToMany` tells NHibernate to cascade events down to the entities in the collection (so when you save the `Store`, all the `Products` are saved too).
3. `Table` sets the many-to-many join table name.

   The `Table` call is currently only required if you're doing a bidirectional many-to-many, because Fluent NHibernate currently can't guess what the name should be; for all other associations it isn't required.

   For NHibernaters: `HasMany` maps to a `bag` by default, and has a `one-to-many` element inside; `HasManyToMany` is the same, but with a `many-to-many` element.

Finally, let's map the `Product`.

```
public class ProductMap : ClassMap<Product>
{
  public ProductMap()
  {
    Id(x => x.Id);
    Map(x => x.Name);
    Map(x => x.Price);
    HasManyToMany(x => x.StoresStockedIn)
      .Cascade.All()
      .Inverse()
      .Table("StoreProduct");
  }
}
```

Mappings/ProductMap.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Mappings/ProductMap.cs)

That's the `Product` mapped; in this case we've used only methods that we've already encountered. The `HasManyToMany` is setting up the other side of the bidirectional many-to-many relationship with `Store`.

Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

Application

In this section we'll initialise some data and output it to the console.

```
static void Main()
{
  var sessionFactory = CreateSessionFactory();

  using (var session = sessionFactory.OpenSession())
  {
    using (var transaction = session.BeginTransaction())
    {
      // create a couple of Stores each with some Products and Employees
      var barginBasin = new Store { Name = "Bargin Basin" };
      var superMart = new Store { Name = "SuperMart" };

      var potatoes = new Product { Name = "Potatoes", Price = 3.60 };
      var fish = new Product { Name = "Fish", Price = 4.49 };
      var milk = new Product { Name = "Milk", Price = 0.79 };
      var bread = new Product { Name = "Bread", Price = 1.29 };
      var cheese = new Product { Name = "Cheese", Price = 2.10 };
      var waffles = new Product { Name = "Waffles", Price = 2.41 };

      var daisy = new Employee { FirstName = "Daisy", LastName = "Harrison" };
      var jack = new Employee { FirstName = "Jack", LastName = "Torrance" };
      var sue = new Employee { FirstName = "Sue", LastName = "Walkters" };
      var bill = new Employee { FirstName = "Bill", LastName = "Taft" };
      var joan = new Employee { FirstName = "Joan", LastName = "Pope" };

      // add products to the stores, there's some crossover in the products in each
      // store, because the store-product relationship is many-to-many
      AddProductsToStore(barginBasin, potatoes, fish, milk, bread, cheese);
      AddProductsToStore(superMart, bread, cheese, waffles);

      // add employees to the stores, this relationship is a one-to-many, so one
      // employee can only work at one store at a time
      AddEmployeesToStore(barginBasin, daisy, jack, sue);
      AddEmployeesToStore(superMart, bill, joan);

      // save both stores, this saves everything else via cascading
      session.SaveOrUpdate(barginBasin);
      session.SaveOrUpdate(superMart);

      transaction.Commit();
    }

    // retreive all stores and display them
    using (session.BeginTransaction())
    {
      var stores = session.CreateCriteria(typeof(Store))
        .List<Store>();

      foreach (var store in stores)
      {
        WriteStorePretty(store);
      }
    }

    Console.ReadKey();
  }
}

public static void AddProductsToStore(Store store, params Product[] products)
{
  foreach (var product in products)
  {
    store.AddProduct(product);
  }
}

public static void AddEmployeesToStore(Store store, params Employee[] employees)
{
```

```
  foreach (var employee in employees)
  {
    store.AddEmployee(employee);
  }
}
```

Program.cs (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Program.cs)

> For brevity, I've left out the definition of `WriteStorePretty` which simply calls `Console.Write` for the various relationships on a `Store` (but you can see it in the full code (http://github.com/jagregory/fluent-nhibernate/blob/master/src/Examples.FirstProject/Program.cs)).

This is the `Main` method from your `Program.cs`. It's a bit lengthy, but what we're doing is creating a couple of `Store` instances, then adds some `Employees` and `Products` to them, then saves; finally, it re-queries them from the database and writes them out to the console.

You won't be able to run this yet, because there's one thing left to do. We need to implement the `CreateSessionFactory` method; that's where our configuration goes to tie NHibernate and Fluent NHibernate together.

Configuration

Let's implement the `CreateSessionFactory` method.

```
private static ISessionFactory CreateSessionFactory()
{

}
```

That's the method signature sorted, you'll note it's returning an NHibernate `ISessionFactory`. Now we're going to use the Fluent NHibernate `Fluently.Configure` API to configure our application. You can see more examples on this in the Fluent configuration wiki page.

```
private static ISessionFactory CreateSessionFactory()
{
  return Fluently.Configure()
    .BuildSessionFactory();
}
```

That's not quite right yet, we're creating a `SessionFactory`, but we haven't configured anything yet; so let's configure our database.

```
private static ISessionFactory CreateSessionFactory()
{
  return Fluently.Configure()
    .Database(
      SQLiteConfiguration.Standard
        .UsingFile("firstProject.db")
    )
    .BuildSessionFactory();
}
```

There we go, we've specified that we're using a file-based SQLite database. You can learn more about the database configuration API in the Database configuration wiki page.

Just one more thing to go, we need to supply NHibernate with the mappings we've created. To do that, we add a call to `Mappings` in our configuration.

```
private static ISessionFactory CreateSessionFactory()
{
  return Fluently.Configure()
    .Database(
      SQLiteConfiguration.Standard
        .UsingFile("firstProject.db")
    )
    .Mappings(m =>
      m.FluentMappings.AddFromAssemblyOf<Program>())
    .BuildSessionFactory();
}
```

That's it; that's your application configured!

You should now be able to run the application and see the results of your query output to the console window (provided that you've actually created the SQLite database schema; otherwise, see the Schema generation section below).

```
Bargin Basin
  Products:
    Potatoes
    Fish
    Milk
    Bread
    Cheese
  Staff:
    Daisy Harrison
    Jack Torrance
    Sue Walkters

SuperMart
  Products:
    Bread
    Cheese
    Waffles
  Staff:
    Bill Taft
    Joan Pope
```

There you go; that's your first Fluent NHibernate project created and running!

Schema generation

If you haven't manually created the schema for this application, then it will fail on the first time you run it. There's something you can do about that, but it needs to be done directly against the NHibernate `Configuration` object; we can do that using the `ExposeConfiguration` method. Combine that call with a method to generate the schema, then you're able to create your schema at runtime.

```
private static ISessionFactory CreateSessionFactory()
{
  return Fluently.Configure()
    .Database(
```

Create PDF files without this message by purchasing novaPDF printer (http://www.novapdf.com)

```
      SQLiteConfiguration.Standard
        .UsingFile("firstProject.db")
    )
    .Mappings(m =>
      m.FluentMappings.AddFromAssemblyOf<Program>())
    .ExposeConfiguration(BuildSchema)
    .BuildSessionFactory();
}

private static void BuildSchema(Configuration config)
{
  // delete the existing db on each run
  if (File.Exists(DbFile))
    File.Delete(DbFile);

  // this NHibernate tool takes a configuration (with mapping info in)
  // and exports a database schema from it
  new SchemaExport(config)
    .Create(false, true);
}
```

You can read more about this in the Fluent configuration wiki page.

Retrieved from "http://wiki.fluentnhibernate.org/Getting_started"

Page Discussion View source History

Creative commons licensed. MediaWiki

Search

Xml validation tool
Automate Tests & Validate for Secure XML WS.
www.parasoft.com

Ads by Google