Log in / create account

Fluent NHibernate wiki

Auto mapping

From Fluent NHibernate

Contents

Fluent NHibernate has a concept called Auto Mapping, which is a mechanism for automatically mapping all your entities based on a set of conventions.

Auto mapping utilises the principle of convention over configuration (http://en.wikipedia.org/wiki/Convention_over_Configuration) . Using this principle, the auto mapper inspects your entities and makes assumptions of what particular properties should be. Perhaps you have a property with the name of Id and type of `int`, the auto mapping will decide that this is an auto-incrementing primary key.

By using the auto mappings, you can map your entire domain with very little code, and certainly no XML. There are still scenarios where it may not be suitable to use the auto mapping, at which point it would be more appropriate to use the ClassMap based fluent mappings; however, for most greenfield applications (and quite a few brownfield ones too) auto mapping will be more than capable.

Getting started

> There's a full example project available in the source download, and in the source code: Examples.FirstAutomappedProject (http://github.com/jagregory/fluent-nhibernate/tree/master/src/Examples.FirstAutomappedProject/) .

Lets go through how to map a simple domain using the Fluent NHibernate AutoMapper.

Given the following clichéd store domain:

```
public class Product
{
  public virtual int Id { get; private set; }
  public virtual string Name { get; set; }
  public virtual decimal Price { get; set; }
}

public class Shelf
{
  public virtual int Id { get; private set; }
  public virtual IList<Product> Products { get; private set; }

  public Shelf()
  {
    Products = new List<Product>();
  }
}
```

We've got a product, with an auto-incrementing primary key called `Id`, and `Name` and `Price` properties. The store has some shelves it fills with products, so there's a Shelf entity, which has an Id, and a list of the Product's on it; the `Product` collection is a one-to-many or HasMany relationship to the Product.

> I'm going to make the assumption here that you have *an existing NHibernate infrastructure*, if you don't then it's best you consult a general NHibernate walkthrough before continuing.

We're going to be using the `AutoMap` class to do our mapping, which you can use in combination with the fluent configuration API. To begin with we should take a look at the static `AutoMap.AssemblyOf<T>` method; this method takes a generic type parameter from which we determine which assembly to look in for mappable entities.

```
AutoMap.AssemblyOf<Product>();
```

That's it, *you've mapped your domain*. There are a few customisations you'll want to make before putting this into production though.

Lets go through what's actually happening here. The `AutoMap.AssemblyOf<Product>` call creates an instance of an `AutoPersistenceModel` that's tied to the assembly that Product is declared. No mappings are actually generated until we push them into NHibernate, this gives us opportunity to customise them before-hand, which we'll go into later.

We're going to utilise the fluent configuration API to create our `SessionFactory`.

```
var sessionFactory = Fluently.Configure()
  .Database(/* database config */)
  .Mappings(m =>
    m.AutoMappings
      .Add(AutoMap.AssemblyOf<Product>()))
  .BuildSessionFactory();
```

You can see that our `AutoMap` is sat in the middle of the configuration, inside the `Mappings` call; this is our way of telling Fluent NHibernate that we're using the auto mapper; there's nothing stopping you from passing in an instance of `AutoPersistenceModel` that you've configured elsewhere, we're just doing it in-line for brevity.

Configuration

We're now capable of getting NHibernate to accept our auto mapped entities, there's just one more thing we need to deal with. The auto mapper doesn't know which classes are your entities, and which ones are *everything else*. The setup we're using above simply maps every class in your assembly as an entity, which isn't going to be very useful.

What we need to do is provide the Automapper with instructions on how to treat your domain. We do this by creating an implementation of the `IAutomappingConfiguration` interface. There are 12+ methods in this interface which control how the automapper works. Fear not though, you don't have to implement them all. The easiest thing you can do is derive from the `DefaultAutomappingConfiguration` class; this class is pre-configured with all the default options for the automapper, all you need to do is override anything that you want to work differently to the defaults.

In our case, we need to restrict the types that the automapper will work with; to do that, we'll create our configuration object and override the `ShouldMap(Type)` method. We'll change the implementation to restrict types that are mapped to just the ones in our entities namespace.

```
public class StoreConfiguration : DefaultAutomappingConfiguration
{
  public override bool ShouldMap(Type type)
  {
    return type.Namespace == "Storefront.Entities";
  }
}
```

To utilise our new configuration, we just need to pass an instance of this configuration to our AutoMap setup.

```
var cfg = new StoreConfiguration();
var sessionFactory = Fluently.Configure()
  .Database(/* database config */)
  .Mappings(m =>
    m.AutoMappings.Add(
      AutoMap.AssemblyOf<Product>(cfg))
  .BuildSessionFactory();
```

> There's nothing stopping you from instantiating your configuration elsewhere, perhaps in an IoC container and injecting it at runtime.

That's all that you should need. It's all a lot easier than writing out mappings, isn't it?

Identities

The automapper is opinionated, it expects your classes to be designed in a particular manner; if they're not, then it won't be able to automap them without a little assistance. The automapper expects your identities to be named `Id`, and if they aren't it won't find them.

You can modify the way the automapper discovers identities by overriding the `IsId` method in your automapping configuration. This method will be called for all the members in your entities that have already been accepted by the `ShouldMap(Member)` criteria; whichever member you return true for will be mapped as the identity.

```
public override bool IsId(Member member)
{
  return member.Name == member.DeclaringType.Name + "Id";
}
```

That example would match any ids that are named after their entity, such as `CustomerId`.

Components

Sometimes you need components in your domain model, here's how to map them automatically.

Lets imagine this database structure:

```
TABLE Person (
  Id int PRIMARY KEY,
  Name varchar(200),
  AddressNumber int,
  AddressStreet varchar(100),
  AddressPostCode varchar(8)
)
```

We want to map that to the following model:

```
public class Person
{
  public virtual int Id { get; private set; }
  public virtual string Name { get; set; }
  public virtual Address Address { get; set; }
}

public class Address
{
  public int Number { get; set; }
  public string Street { get; set; }
  public string PostCode { get; set; }
}
```

With this design Address is actually a component, which isn't a full entity, more of a way of providing a clean model to a normalised database structure. Lets start with what we left off with:

```
var cfg = new StoreConfiguration();
AutoMap.AssemblyOf<Person>(cfg);
```

We've now got the auto mappings integrated with NHibernate, so we need to instruct the auto mapper how to identify components. Inside our configuration, similar to how we overrode the `ShouldMap(Type)`, we need to override the `IsComponent` method; this method will be called for each type that has already been accepted by `ShouldMap(Type)`, and whichever types you return true for will be mapped as components.

```
public override bool IsComponent(Type type)
{
  return type == typeof(Address);
}
```

> It's fairly common that you'll have more than one component type, so you'll need to chain together ORs in your `IsComponent` method, or check on something more generic like the namespace.

The Address should now be automatically mapped as a component; the auto mapper will pick up the three properties and map them as properties on the component.

The default column names for the component's properties are a combination of the name of the property in the entity and the name of the property in the component type. For example, if the Person class had a HomeAddress property, it would be mapped to the columns HomeAddressStreet, HomeAddressPostCode, etc.

If you need to customise the column naming of your component's properties you can override the `GetComponentColumnPrefix` method:

```
public override string GetComponentColumnPrefix(Member member)
{
  return member.PropertyType.Name + "_";
}
```

The convention now specifies that columns should be named `TypeName_PropertyName`, so `Address.Street` is now mapped to `Address_Street`.

Conventions

The mappings produced by Fluent NHibernate are based on a pre-defined set of assumptions about your domain, this allows us to greatly reduce the amount of code you're required to write; sometimes however, the conventions we supply are not to your liking, perhaps you're a control-freak and want to have full say over your design, or more likely you're working against an existing database that has it's own set of standards.

You'd still like to use the auto mapper, but can't because it maps your entities all wrong. Luckily for you we've thought about that, the auto mapper can utilise the conventions just like fluent mappings.

Using the following entities:

```
public class Product
{
  public int Id { get; private set; }
  public virtual string Name { get; set; }
  public virtual decimal Price { get; set; }
}

public class Shelf
{
  public virtual int Id { get; private set; }
  public virtual IList<Product> Products { get; private set; }

  public Shelf()
  {
    Products = new List<Product>();
  }
}
```

The standard conventions, it'd map to a database schema like this:

```
TABLE Product (
  Id int identity PRIMARY KEY,
  Name varchar(100),
  Price decimal,
  Shelf_id int FOREIGN KEY
)

TABLE Shelf (
  Id int identity PRIMARY KEY
)
```

Nothing too complicated there, but that might not be the schema you expect. Lets pretend (or not!) that you name the primary key after the table it's in, so our Product identity should be called `ProductId`; also, you like your foreign key's to be explicitly named `_FK`, and your strings are always a bit longer than 100.

Remember this fellow?

```
var cfg = new StoreConfiguration();
AutoMap.AssemblyOf<Product>(cfg);
```

Lets update it to include some convention overrides. We'll start with the Id name. The conventions we're about to implement are better explained in the conventions page.

```
AutoMap.AssemblyOf<Product>(cfg)
  .Conventions.Add<PrimaryKeyConvention>();
```

We've added a convention to the convention discovery mechanism, now let's implement it.

```
public class PrimaryKeyConvention
  : IIdConvention
{
  public void Apply(IIdentityInstance instance)
  {
    instance.Column(instance.EntityType.Name + "Id");
  }
}
```

Our `PrimaryKeyConvention` gets applied to all Ids and sets their column name based on the entity that contains the Id property. Our primary key's will now be generated as TypeNameId; which means our schema now looks like this:

```
TABLE Product (
  ProductId int identity PRIMARY KEY,
  Name varchar(100),
  Price decimal,
  Shelf_id int FOREIGN KEY
)

TABLE Shelf (
  ShelfId int identity PRIMARY KEY
)
```

As you can see, our primary key's now have our desired naming convention. Lets do the other two together, as they're so simple; we'll override the foreign-key naming, and change the default length for strings.

```
.Conventions.Setup(c =>
{
  c.Add<PrimaryKeyConvention>();
  c.Add<CustomForeignKeyConvention>();
  c.Add<DefaultStringLengthConvention>();
});

public class CustomForeignKeyConvention
  : ForeignKeyConvention
{
  protected override string GetKeyName(PropertyInfo property, Type type)
  {
    if (property == null)
      return type.Name + "_FK";

    return property.Name + "_FK";
  }
}

public class DefaultStringLengthConvention
  : IPropertyConvention
{
  public void Apply(IPropertyInstance instance)
  {
    instance.Length(250);
  }
}
```

That's all there is to it, when combined with the other conventions you can customise the mappings quite heavily while only adding a few lines to your auto mapping.

This is our final schema:

```
TABLE Product (
  ProductId int identity PRIMARY KEY,
  Name varchar(250),
  Price decimal,
  Shelf_FK int FOREIGN KEY
)
```

```
TABLE Shelf (
  ShelfId int identity PRIMARY KEY
)
```

Altering entities

Sometimes it's necessary to make slight changes to a specific entity, without wishing to affect anything else; you can do that with the with `Override<T>` method.

```
.Override<Shelf>(map =>
{
  map.HasMany(x => x.Products)
    .Cascade.All();
});
```

The `Override` method takes a generic parameter that's the entity you want to customise. The parameter is an expression that allows you to alter the underlying mapping that is generated by the auto mapper. You can do just about anything in this call that you can do in the fluent mappings.

In the example above we're setting `Cascade.All` on the HasMany of Products for the entity Shelf.

You can do this for as many types as you need in your domain; however, baring in mind readability, it may sometimes be more appropriate to use an override or map entities explicitly using the fluent mappings if you find yourself overriding a lot of conventions.

Overrides

Using too many calls to `Override` can quickly clutter up your auto mapping setup; an alternative is to use an `IAutoMappingOverride<T>`, which is an interface you can implement to override the mappings of a particular auto-mapped class.

```
public class PersonMappingOverride
  : IAutoMappingOverride<Person>
{
  public void Override(AutoMapping<Person> mapping)
  {
  }
}
```

This example overrides the auto-mapping of a Person entity. Within the `Override` method you can perform any actions on the mapping that you can in the fluent mappings.

To use overrides, you need to instruct your AutoMap instance to use them. Typically this would be done in the context of a fluent configuration setup, but I'll just illustrate with the AutoMap on it's own.

```
AutoMap.AssemblyOf<Person>(cfg)
  .UseOverridesFromAssemblyOf<PersonMappingOverride>();
```

It's the `UserOverridesFromAssemblyOf<T>` call that instructs the AutoPersistenceModel to read any overrides that reside the assembly that contains `T`.

Ignoring properties

When using the auto mapper sometimes you may want to ignore certain properties on your entity. If it's only for one entity, then you can override the entity or specify an alteration each which allow you to call an `IgnoreProperty` method.

For example:

```
.Override<Shelf>(map =>
{
  map.IgnoreProperty(x => x.YourProperty);
});
```

If you need to ignore a property regardless of what type it's present on, you can use the `OverrideAll` method, which exposes a similar interface.

```
.OverrideAll(map =>
{
  map.IgnoreProperty("YourProperty");
});
```

The difference in this case is that because we don't know the entity at the time of writing our ignore call, we have to specify the name with a string; this is not ideal, but there aren't really any alternatives in this situation.

There are a couple of overloads available when using `OverrideAll`.

A single property:

```
.OverrideAll(map =>
{
  map.IgnoreProperty("YourProperty");
});
```

Multiple properties:

```
.OverrideAll(map =>
{
  map.IgnoreProperties("YourProperty", "AnotherProperty");
});
```

Properties by predicate:

```
.OverrideAll(map =>
{
  map.IgnoreProperties(x => x.Name.Contains("Something"));
});
```

That last one is powerful. You can do whatever you like in that predicate against the `PropertyInfo` instance supplied. You could check if the name contains a particular string, like above, or you could check if the property has a particular attribute on.

Inheritance

There are two main things that you'd want to do with inherited classes, either ignore the base class all together, or map them using an inheritance strategy. I'm going to start with the former, then move on to the latter.

Ignoring base-types

This scenario is where you may have a base class in your domain that you use to simplify your entities, you've moved common properties into it so you don't have to recreate them on every entity; typically this would be the Id and perhaps some audit information. So lets start with a model that has a base class we'd like to ignore.

```
namespace Entities
{
  public abstract class Entity
  {
    public virtual int Id { get; set; }
  }

  public class Person : Entity
  {
    public virtual string FirstName { get; set; }
    public virtual string LastName { get; set; }
  }

  public class Animal : Entity
  {
    public virtual string Species { get; set; }
  }
}
```

Relatively simple model here, we've got an Entity base class that defines the Id, then the Person and Animal entities. We have no desire to have Entity mapped by NHibernate, so we need a way to tell the auto mapper to ignore it.

For those individuals from traditional XML mapping land, this is what we're going to be recreating:

```
<class name="Person">
  <id name="Id">
    <generator class="identity" />
  </id>

  <property name="FirstName" />
  <property name="LastName" />
</class>

<class name="Animal">
  <id name="Id">
    <generator class="identity" />
  </id>
  <property name="Species" />
</class>
```

We'll start with this automapping setup:

```
AutoMap.AssemblyOf<Entity>(cfg);
```

If we were to run this now, we wouldn't get the mapping we desire. Fluent NHibernate would see Entity as an actual entity and map it with Animal and Person as subclasses; this is not what we desire, so we need to modify our auto mapping configuration to reflect that.

You can ignore base types by simply excluding them from your `ShouldMap(Type)` method, that's sometimes the cleanest option; however, if you want to be a bit more explicit you can use the `IgnoreBase` method.

After `AutoMap.AssemblyOf<Entity>()` we need to alter the conventions that the auto mapper is using so it can identify our base-class.

```
AutoMap.AssemblyOf<Entity>(cfg)
  .IgnoreBase<Entity>();
```

We've added the `IgnoreBase<Entity>` call which simply instructs the automapper to ignore the Entity class; you can chain this call as many times as needed.

With this change, we now get our desired mapping. Entity is ignored as far is Fluent NHibernate is concerned, and all the properties (Id in our case) are treated as if they were on the specific subclasses.

Base-type as an inheritance strategy

If you want to have your base-class included in NHibernate, then just don't include the IgnoreBase above! Easy.

Configuring the subclassing strategy

Fluent NHibernate defaults to using table-per-subclass strategy for automapping inheritance hierarchies. If this is not what you want, then you can modify your configuration to specify with types use a discriminator (and therefore are table-per-hierarchy).

If you always want table-per-hierarchy subclassing, then you just need to override the `IsDiscriminated` configuration method to always return `true`

```
public override bool IsDiscriminated(Type type)
{
  return true;
}
```

If you wanted to customise it on a per-class basis, you'd set it up like so:

```
public override bool IsDiscriminated(Type type)
{
  return type.In(typeof(ClassOne), typeof(ClassTwo));
}
```

That will make any subclasses of `ClassOne` and `ClassTwo` share their parent table, rather than being in their own tables. All other entities will have their own tables for subclasses.

Subclassing FAQs

Abstract base-classes

You'll notice that our Entity class is `abstract`. This is good practice, but for the record, it is *not* mandatory. If you're experiencing problems, it's unlikely to be this.

In case you're wondering, making the class `abstract` is like saying "I'll never create this directly, instead I will create derived classes such as Customer and Order (which inherit from Entity)."

The default behavior is to consider abstract classes as layer supertypes (http://martinfowler.com/eaaCatalog/layerSupertype.html) and effectively unmapped, you may want to change this for specific scenarios. The easiest way to do this is to use `IncludeBase<T>`, where T is your entity.

```
AutoMap.AssemblyOf<Entity>(cfg)
  .IncludeBase<AbstractBaseClass>();
```

This *forces* the automapper to include that base-type, regardless of it's previous assumptions about it.

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

Page Discussion View source History

Search