Fluent NHibernate wiki

Fluent mapping

From Fluent NHibernate

Contents

- 1 ClassMap
- 2 Id
- 3 Properties
- 4 Relationships
  - 4.1 References / many-to-one
  - 4.2 HasMany / one-to-many
  - 4.3 HasManyToMany / many-to-many
  - 4.4 HasOne / one-to-one
  - 4.5 Any
- 5 Components
  - 5.1 ComponentMap<T>
- 6 Subclasses

Fluent mapping is the namesake mapping style that Fluent NHibernate uses. It's a fluent interface (http://martinfowler.com/bliki/FluentInterface.html) that allows you to map your entities completely in code, with all the compile-time safety and refactorability that brings.

The getting started guide has a good introduction to mapping with the fluent interface, and the detail provided here builds upon that introduction.

ClassMap

`ClassMap<T>` is the basis of all your mappings, you derive from this to map anything.

```
public class PersonMap : ClassMap<Person>
{
  public PersonMap()
  {
  }
}
```

You map your entities' properties inside the constructor.

> **Syntax note:** Every mapping inside a `ClassMap<T>` is built using lambda expressions, which allow us to reference the properties on your entities without sacrificing compile-time safety. The lambdas typically take the form of `x => x.Property`. The `x` on the left is the parameter declaration, which will implicitly be of the same type as the entity being mapped, while the `x.Property` is accessing a property on your entity (coincidently called "Property" in this case). You'll quickly get used to these lambdas, as they're used *everywhere* in Fluent NHibernate.

Once you've declared your `ClassMap<T>` you're going to need to map the properties on your entity. There are several methods available that map your properties in different ways, and each one of those is a chainable method (http://martinfowler.com/dslwip/MethodChaining.html) that you can use to customise the individual mapping.

Id

Every mapping requires an Id of some kind. The Id is mapped using the `Id` method, which takes a lambda expression that accesses the property on your entity that will be used for the Id. Depending on the return type of the property accessed in the lambda, Fluent NHibernate will make some assumptions about the kind of identifier you're using. For example, if your Id property is an `int`, an identity column is assumed. Similarly, if you use a `Guid` then a Guid Comb (http://nhforge.org/blogs/nhibernate/archive/2009/05/21/using-the-guid-comb-identifier-strategy.aspx) is assumed.

```
Id(x => x.Id);
```

Note that the property you supply to `Id` can have any name; it does *not* have to be called "Id," as shown here.

That's the most common scenario for mapping your Id. Customisations can be done by chaining methods off the `Id` call. For example, if the column to which the `Id` property was to be mapped were not called `Id`, we could use the `Column` method to specify the name. For explicitly specifying the identity generator, you could use the `GeneratedBy` property.

```
Id(x => x.Id)
  .Column("PersonId")
  .GeneratedBy.Assigned();
```

In this example we're specifying that the Id property is mapped to a PersonId column in the database, and it's using an assigned generator (http://www.nhforge.org/doc/nh/en/index.html#mapping-declaration-id-assigned) .

> Fluent NHibernate's interface is designed for discoverability. Everything you need should be easy to find "under" the declaring method using method chains.

Properties

Property mappings make up a large amount of any mapped domain. Fortunately, they're just as easy to create as identities, except we use the `Map` method.

```
Map(x => x.FirstName);
```

That's all you need for most situations. Fluent NHibernate knows what the return type of your property is, and assumes that the column it's mapping against will have the same name as the property itself. There are numerous customisations available through methods chained from the `Map` call. For example, if you're generating your schema you may want to specify whether the column itself is nullable, you can do that by using the `Nullable` method and (optionally) the `Not` operator property.

```
// nullable
Map(x => x.FirstName)
  .Nullable();

// not nullable
Map(x => x.FirstName)
  .Not.Nullable();
```

If you need to map private properties, you can explore your options here.

Relationships

Except in the most basic of scenarios, you'll need to map relationships among entities. These will typically take the form of many-to-one, one-to-many, and many-to-many relationships. For better or worse, we tend not to refer to these by their database design names: we aren't database administrators, after all. Instead, we refer to them as *References*, *HasMany*, and *HasManyToMany*, respectively. We'll go into each in more detail next.

References / many-to-one

References is for creating many-to-one relationships between two entities, and is applied on the "many side." You're *referencing* a single other entity, so you use the `References` method. HasMany is the "other side" of the References relationship, and gets applied on the "one side."

Let's map a relationship between a book and it's author.

```
public class Book
{
  public Author Author { get; set; }
}

public class Author
{
  public IList<Book> Books { get; set; }
}
```

In domain terms, we have an `Author` which may be associated with any number of `Books`, and `Books`, each of which can be associated with a single `Author`.

In database terms, we'd have a book table with a foreign key column referencing the primary key of an author table.

To create the references relationship in your `Book` mapping, add the following call in the `BookMap` constructor:

```
References(x => x.Author);
```

That's it, you've now created a references relationship between `Book` and `Author`. The foreign-key is assumed to be named `Author_id` in your book table, unless you have a different Convention in place or override the name by using the `Column` method.

As with all other fluent mappings, you can chain calls to customise the reference relationship. For example if you wanted to specify the cascade strategy you'd use the `Cascade` property.

```
References(x => x.Author)
  .Column("AuthorId")
  .Cascade.All();
```

HasMany / one-to-many

HasMany is probably the most common collection-based relationship you're going to use. HasMany is the "other side" of a References relationship, and gets applied on the "one side" (one author has many books). Now let's map the author side of the relationship we started above. We need to join into the books table returning a collection of any books associated with that author.

```
public class Author
{
  public IList<Book> Books { get; set; }
}
```

We use the `HasMany` method in the `AuthorMap` constructor to map this side of the relationship:

```
HasMany(x => x.Books);
```

As with References, the foreign-key defaults to `Author_id`, and you can override it with the `KeyColumn` method or change the default behaviour with a Convention.

There are a few different types of collections you can use, and they're all available under the `HasMany` call. You can read about them here.

HasManyToMany / many-to-many

HasManyToMany works exactly the same as HasMany, except the underlying database structure to which it maps is different.

```
HasManyToMany(x => x.Books);
```

There are a few different types of collections you can use, and they're all available under the HasManyToMany call. You can read about them here.

HasOne / one-to-one

HasOne is usually reserved for a special case. Generally, you'd use a References relationship in most situations (see: I think you mean a many-to-one (http://blog.jagregory.com/2009/01/27/i-think-you-mean-a-many-to-one-sir/)). If you really do want a one-to-one, then you can use the `HasOne` method.

```
HasOne(x => x.Cover);
```

Any

Any mappings are another special case, and you really only should use them if you know what you're doing. To quote the NHibernate documentation (http://www.nhforge.org/doc/nh/en/index.html#mapping-types-anymapping) :

> The `<any>` mapping element defines a polymorphic association to classes from multiple tables. This type of mapping always requires more than one column. The first column holds the type of the associated entity. The remaining columns hold the identifier. It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations. You should use this only in very special cases (eg. audit logs, user session data, etc).

There are three things you need to provide to be able to map using an Any:

1. A column that holds the type of the entity,
2. At *least* one column holding the identifier value, and
3. A type for the identifier itself.

You can specify these using the `EntityTypeColumn`, `EntityIdentifierColumn`, and `IdentityType` methods respectively.

```
ReferencesAny(x => x.Author)
  .EntityTypeColumn("Type")
  .EntityIdentifierColumn("Id")
  .IdentityType<int>();
```

Components

Components are a clever way of mapping a normalised data model into a more reasonable object model. You may have a customer table that has a series of address columns, ideally you'd want the address columns to be mapped into an Address object, rather than just being properties on a Customer; you can do that with components.

The `Component` method takes two parameters, rather than just one like the rest of the methods you've seen so far. The first parameter is a property accessor lambda, like all the other methods, and the second one is another lambda (quite often referred to as a nested-closure in these situations) that defines a new scope for defining the mappings of that particular sub-part (in this case the component).

```
Component(x => x.Address, m =>
{
  m.Map(x => x.Number);
  m.Map(x => x.Street);
  m.Map(x => x.PostCode);
});
```

As you can see, the first parameter references the `Address` property on our entity, which is the property that holds our component. The second property is where we define what makes up the component. The main difference in this lower scope is that you have to use the `m` instance to access the available methods; you don't have to call it `m`, but we are for brevity.

In this case we've mapped a component (stored in the Address property), that's made up of three properties (Number, Street, and PostCode); these properties are stored in the same table as the parent entity in a normalised fashion.

ComponentMap<T>

> Note: ComponentMap's are incompatible with the automapper. If you're using the automapper then your ComponentMap's will be ignored. See automapping components for guidance on using components with the automapper.

If you have a particular component that occurs regularly in your entities, you can abstract the mappings into a single `ComponentMap<T>` definition.

Using the example of an `Address` again, it wouldn't be surprising to find that multiple entities had addresses; perhaps `Person`, and `Company`.

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

public class Person
{
  public Address Address { get; set; }
}
```

```
public class Company
{
  public Address Address { get; set; }
}
```

Normally you would need to define an inline component mapping for each occurrence of `Address`, using the `Component(property, mapping)` method, even though they'd be identical.

```
public PersonMap()
{
  Component(x => x.Address, m =>
  {
    m.Map(x => x.Number);
    m.Map(x => x.Street);
    m.Map(x => x.City);
    m.Map(x => x.PostCode);
  });
}

public CompanyMap()
{
  Component(x => x.Address, m =>
  {
    m.Map(x => x.Number);
    m.Map(x => x.Street);
    m.Map(x => x.City);
    m.Map(x => x.PostCode);
  });
}
```

However, using the `ComponentMap<T>`, you can define your component mapping in a single location.

```
public class AddressMap : ComponentMap<Address>
{
  public AddressMap()
  {
    Map(x => x.Number);
    Map(x => x.Street);
    Map(x => x.City);
    Map(x => x.PostCode);
  }
}
```

With that created, you only need to signify that your properties are a component from your entity mappings by using the new `Component(property)` overload that takes a single parameter (it doesn't take a closure for defining the body like the other `Component` methods).

```
public PersonMap()
{
  Component(x => x.Address);
}

public CompanyMap()
{
  Component(x => x.Address);
}
```

Column names

If you have multiple instances of the same component in an entity, say `WorkAddress` and `HomeAddress`, you'll need to supply a prefix for the columns. This is because the column names are, by default, based on the properties within the component, so you'd end up with two Streets, etc...

To supply a prefix, chain a call to `ColumnPrefix` from your `Component(property)` call.

> **Note:** this only works for `ComponentMap<T>`-based components.

```
Component(x => x.WorkAddress)
  .ColumnPrefix("Work");
```

This would create WorkStreet, WorkCity, WorkPostCode columns.

Subclasses

> Note: SubclassMap's are incompatible with the automapper. If you're using the automapper, your SubclassMap's will be ignored. See automapping inheritance for guidance on dealing with subclasses with the automapper.

Subclasses work in a very similar way to ClassMaps, in that you create a derived class into which you put your mappings, but here you use `SubclassMap` instead of `ClassMap`.

There are two strategies for mapping inheritance hierarchies in Fluent NHibernate: table-per-class-hierarchy and table-per-subclass, the former being a subclass and the latter a joined-subclass. This means that you can define one table to hold all subclasses (with 1 or more columns used to identify the specific type for each row), or you can define separate tables for each subclass.

> **Note:** For multiple discriminator columns use a `DiscriminateSubClassesOnColumn ("").Formula([insert custom sql here])`

```
DiscriminateSubClassesOnColumn("").Formula("case when discriminatorColumnName = '' then 1 else 2 end")
```

Table-per-subclass is the default mapping for subclasses, so unless you say otherwise you'll have a separate table for each subclass. The parent mapping dictates what the subclass mapping strategy will be, by either specifying or not specifying a discriminator (discriminators are required for table-per-class-hierarchy).

We'll use the following two classes for examples:

```
public class Parent
{
  public int Id { get; set; }
  public string Name { get; set; }
}

public class Child : Parent
{
  public string AnotherProperty { get; set; }
}
```

If you wanted to map this as a table-per-subclass, you'd do it like this:

```
public class ParentMap : ClassMap<Parent>
{
  public ParentMap()
  {
    Id(x => x.Id);
    Map(x => x.Name);
  }
}

public class ChildMap : SubclassMap<Child>
{
  public ChildMap()
  {
    Map(x => x.AnotherProperty);
  }
}
```

That's all there is to it, `Parent` and `Child` are now mapped as subclasses. When Fluent NHibernate finds `ChildMap`, it knows that it's a subclass of `Parent`.

If you wanted to do a table-per-class-hierarchy strategy, then you just need to specify the discriminator column in your `ClassMap`.

```
public class ParentMap : ClassMap<Parent>
{
  public ParentMap()
  {
    Id(x => x.Id);
    Map(x => x.Name);

    DiscriminateSubClassesOnColumn("type");
  }
}

public class ChildMap : SubclassMap<Child>
{
  public ChildMap()
  {
    Map(x => x.AnotherProperty);
  }
}
```

The only difference is that we're now calling `DiscriminateSubclassesOnColumn` in `ParentMap` with a `"type"` parameter. This parameter specifies the name of the column in the table that identifies which subclass the data in a given row represents.

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

Page Discussion View source History
