

Tales from the Evil Empire

Bertrand Le Roy's blog

A total n00b's guide to migrating from a custom data layer to Nhibernate: so many choices



One of the great things about Nhibernate is its vibrant community and ecosystem. So many people are using it or building other libraries on top of it that you can be pretty sure that there is always a reasonable solution to any problem you might face. Or several.

This means of course that there are lots of choices you can make about how you use Nhibernate. While this is essentially a good thing, too many choices can be intimidating when starting to use a technology.

In this post, I'll present some of the first choices you are going to have to make for your Nhibernate code. I'll explain the choices that our team made for our own migration and an attempt at explaining why you would make different choices depending on the particular context of your application.

1. Database first or object first?

That's an easy one.

If you already have a database, it's database first.

If you already have both, all you have to do is build the mappings.

If you already have an object model, it depends how much you care about the shape of your database. Nhibernate has reasonable defaults so you might be able to get away with object first and reasonably fuzzy mappings (or even a [generated mapping](#)). The obvious advantage of that approach is that you'll be able to easily move database engines without having to rewrite anything but your web.config.

If you need more control over the shape of the database, you might be able to achieve just that by making your mappings more explicit but still keep generation of the database from Nhibernate.

If that's still not enough control, just don't generate the database, and instead manually maintain database, objects and mappings.

In our own project, we had the database already, but we want to abstract ourselves of a particular engine eventually, so we are moving from a database first approach to an object first approach. Extensions to the application will have to work independently of the database engine (because the app itself is), so it will have to be object first for them.

2. To lazy-load or not to lazy-load?

Lazy loading is the practice of delaying the loading of an object until the moment when it is actually used for the first time. If you have a collection of blog post objects for example, and those objects each have a property that is their collection of comments, you won't want those comments to be queried from the database unless you are actually going to use them.

Lazy loading is obviously a good thing, but you also must know that it can get you in trouble. So it can be a dangerous tool, but its usefulness far outweighs the danger.

I've also mentioned in the [previous post](#) that lazy loading requires the use of dynamic proxies and that these in turn require that you run in higher trust than medium. Well, several commenters pointed out that this isn't exactly true.

Lazy loading of collection properties does not require the generation of dynamic proxies at all because Nhibernate can set the value of those properties directly to a list type that already implements lazy loading. It is only in the case of lazy loading of properties or 1-1 associations that the proxy generation

will be necessary. In other words, if your database schema only has 1-N and N-N associations or if you don't mind eager loading on your 1-1's, NHibernate can run in medium trust with no problem. This is the approach we are currently using in our application. We will revise it if necessary.

The first way to achieve medium trust lazy-loading is to generate the proxies at build-time instead of dynamically at runtime. Caveat with this approach is that the generated proxies will be specialized to a dialect, which might or might not be a problem. You may be able to mitigate that problem by building dialect-specific versions for all databases that you target, but this is definitely less convenient than the entirely dynamic approach.

The second approach is to fix the code of the proxy factory that you are using to work in medium trust, which is entirely achievable but clearly not for n00bs. Apparently patches are on their way here and this could make the whole problem go away in the near future.

And the last and lamest approach is to disable lazy loading entirely.

3. What dynamic proxy factory to use?

Once you've decided that you're going to use lazy loading and are going to need dynamic proxies, you still have to decide which proxy factory you're going to use. Three choices are available out of the box: Castle, LinFu and Spring.

If you want to work with Castle.ActiveRecord or Castle.Windsor, Castle is the obvious choice.

If you want to work with Spring, Spring is obviously what you should use.

If you are not using Castle or Spring, or have no idea what you want, LinFu can be a reasonable default.

In our application, because we don't need dynamic proxies for the moment, we haven't had to choose.

3. XML, attributes or fluent mappings?

XML

XML mappings are the default but some people dislike XML's verbosity. There is also a fair amount of repetition and magic strings involved. Refactoring won't propagate automatically to the mapping files. Advantages include that it's the most mature approach and that it doesn't require any additional dependency.

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

  <class name="Product" table="Products">
    <id name="SKU">
      <generator class="uuid.hex" />
    </id>
    <property name="ProductName"/>
    <property name="BasePrice" />
  </class>

</hibernate-mapping>
```

Attributes

It is possible to use attributes through Castle to specify the mappings instead. This is very easy, involves minimal repetition and a less magic strings than XML. On the other hand, doing the mapping on the objects themselves means less separation of concerns as the objects and the mappings are in the same place. It also means buying into the whole active record approach and giving up POCOs.

```
[ActiveRecord("Products")]
public class Product : ActiveRecordBase {
  [PrimaryKey(PrimaryKeyType.UuidHex)]
  public string SKU { get; set; }
  [Property]
  public string ProductName { get; set; }
  [Property]
  public double Price { get; set; }
}
```

Fluent

Finally, the favorite approach *du jour* is Fluent NHibernate. That approach still keeps mappings nicely

separated but has several other advantages.

First, it's code, which opens up a lot of dynamic scenarios.

Second, it's strongly-typed. That means that refactoring will work, that you will get compile-time checks and IntelliSense.

Third, it uses Lambdas and a fluent interface, and those are so cute... On the other hand, if you find those alien, it might not be such a win. On the other other hand, it's a good occasion to learn something new and cool.

```
public class ProductMap : ClassMap<Product> {
    public ProductMap() {
        Id(p => p.SKU).GeneratedBy.UuidHex("B");
        Map(p => p.ProductName);
        Map(p => p.Price);
    }
}
```

In our application, we have chosen XML mappings for now to minimize the number of dependencies and because choosing the most mature approach also means learning is more incremental. We might revise that decision if and when the disadvantages become too much of a problem and as we learn more.

4. HQL, criteria, lambdas, ActiveRecord or Linq?

HQL

HQL is the Hibernate Query Language and is the default way of querying in NHibernate. It works and is mature but it is a text format, which means no static verification, refactoring or IntelliSense.

```
"select p from Product as p where p.ProductName like 'boot%'"
```

There are a few alternatives to HQL though. Let's first talk about native SQL. It is possible to send native SQL from NHibernate but you should really only do that if you need to use a specific native feature of your database. While the possibility exists, it is not a relevant choice here.

CreateCriteria

The NHibernate session object has a CreateCriteria method that exposes a fluent querying interface. You can use it with expressions, but this is deprecated, or you can use it with restrictions, which are very similar but more current.

```
session.CreateCriteria<Product>()
    .Add(Restrictions.Like("ProductName", "boot%"));
```

This API tends to be relatively verbose and is not completely strongly typed but it may be more appropriate than HQL for simple queries.

Lambdas

The nhlambdaextensions project provides a strongly-typed alternative to the restrictions described above through the use of Lambda expressions. This is a strong choice that comes close to Linq but it does require yet another dependency.

```
session.CreateCriteria<Product>()
    .Add<Product>(p =>
        p.ProductName.StartsWith("boot",
            StringComparison.CurrentCultureIgnoreCase));
```

ActiveRecord

If you chose to use Castle ActiveRecord, you can still query using HQL, but you also get basic repository-like operations directly on your data classes out of the box.

```
FindAll(typeof(Product), Restrictions.Like("ProductName", "boot%"));
```

Linq

Finally, you can use an implementation of Linq to query NHibernate. This is Linq, for NHibernate. In other words, pure, distilled WIN: strong typing, compile-time checks, etc. The problem is that it is still a work in progress (and an additional dependency).

```
from product in session.Linq<Product>()  
  where product.StartsWith("boot",  
    StringComparison.CurrentCultureIgnoreCase)  
  select product;
```

Linq to NHibernate can be obtained from [NHibernate Contrib](#) or as part of [Fluent NHibernate](#).

For the moment, our application uses CreateCriteria with restrictions for simple queries, and HQL in places. Again, the motivation here is to minimize the dependencies.

Conclusion

I hope this post will be helpful for NHibernate beginners such as myself to make the right first choices.

Next time, we'll really start digging into our transition from our old custom data layer.

Posted: [Aug 20 2009, 02:05 PM](#) by [Bertrand Le Roy](#) | with [1 comment\(s\)](#)
Filed under: [ASP.NET](#), [NHibernate](#)

Comments

Joe said:

Thanks for these posts! Keep it up.

August 20, 2009 10:04 PM

[Terms of Use](#)