Understanding the NHibernate Type System

10.08.2009

 | 31949 views |

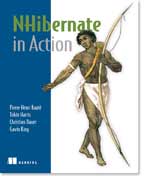
[inShare](javascript:void(0);)1

* 1  
  [inShare](javascript:void(0);)
* [submit to reddit](http://www.reddit.com/r/programming/submit)

This article is taken from [NHibernate in Action](http://www.manning.com/kuate/) from Manning Publications. This article delves into the NHibernate type system. For the table of contents, the Author Forum, and other resources, go to <http://www.manning.com/kuate/>.

It is being reproduced here by permission from [Manning Publications](http://www.manning.com/). Manning ebooks are sold exclusively through Manning. Visit the book's page for more information.

Softbound print: February 2009 | 400 pages   
ISBN: 1932394923

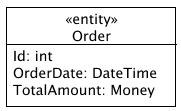
[](http://www.manning.com/kuate/)

**Use code "dzone30" to get 30% off any version of this book.**

Entities are the coarse-grained classes in a system. You usually define the features of a system in terms of the entities involved: “the user places a bid for an item” is a typical feature definition that mentions three entities - user, bid and item. In contrast, value types are the much more fine grained classes in a system, such as strings, numbers, dates and monetary amounts. These fine grained classes can be used in many places and serve many purposes; the value type string can store email address, usernames and many other things. Strings are simple value types, but it is possible (but less common) to create value types that are more complex. For example, a value type could contain several fields, like an Address.

So how do we differentiate between value types and entities? From a more formal standpoint, we an say an entity is any class whose instances have their own persistent identity, and a value type is a class who’s instances do not. The entity instances may therefore be in any of the three persistent lifecycle states: transient, detached, or persistent. However, we don’t consider these lifecycle states to apply to the simpler value type instances. Furthermore, because entities have their own lifecycle, the Save() and Delete() methods of the NHibernate ISession interface will apply to them, but never to value type instances. To illustrate, lets consider Figure A.

Figure A – An order entity with TotalAmount value type

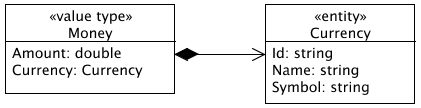


TotalAmount is an instance of value type Money. Because value types are completely bound to that their owning entities, TotalAmount is only saved when the Order is saved.

**Associations and Value Types**

As we said, not all value types are simple. It’s possible for value types to also define associations. For example, our Money value type could have a property called Currency that is an association to a Currency entity as shown in figure 6.1.2

Figure B – The Money value type with association to a Currency entity.



If your value types have associations, they must always point to entities. The reason is that, if those associations could point from entities to value types, a value type could potentially belong to several entities, which isn’t desirable. This is one of the great things about value types; if you update a value type instance, you know that it only affects the entity that owns it. For example, changing the TotalAmount of one Order simply cannot accidentally affect others. So far we’ve talked about value types and entities from an object oriented perspective. To build a more complete picture, we shall now take a look at how the relational model sees value types and entities, and how NHibernate bridges the gap.

**Bridging from objects to database**

You may be aware that a database architect would see the world of value types and entities slightly differently to this object oriented view of things. In the database, tables represent the entities, and columns represent the values. Even join tables and lookup tables are entities. So, if all tables represent entities in the database, does that mean we have to map all tables to entities in our .NET domain model? What about those value types we wanted in our model? NHibernate provides constructs for dealing with this. For example, a many-to-many association mapping hides the intermediate association table from the application, so we don’t end up with an unwanted entity in our domain model. Similarly, a collection of value typed strings behaves like a value type from the point of view of the .NET domain model even though it’s mapped to its own table in the database.

These features have their uses and can often simplify your C# code. However, over time we have become suspicious of them; these “hidden” entities often end up needing exposure in our applications as business requirements evolve. The many-to-many association table, for example, often has additional columns added as the application matures, so the relationship itself becomes an entity. You might not go far wrong if you make every database-level entity be exposed to the application as an entity class. For example, we’d be inclined to model the many-to-many association as two one-to-many associations to an intervening entity class. We’ll leave the final decision to you, and return to the topic of many-to-many entity associations later in this chapter.

**Mapping types**

So far we’ve discussed the differences between value types and entities, as seen from the object oriented and relational database perspectives. We know that mapping entities is quite straight forward – entity classes are simply always mapped to database tables using <class>, <subclass>, and <joined-subclass> mapping elements.

Value types need something more, which is where mapping types enter the picture. Consider this mapping of the CaveatEmptor User and email address:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.<property

2.name="Email"

3.column="EMAIL"

4.type="String"/>

In ORM, you have to worry about both .NET types and SQL data types. In the example above imagine that the Email field is a .NET string, and EMAIL column is an SQL varchar. We want to tell NHibernate know how to carry out this conversion, which is where NHibernate mapping types come in. In this case, we’ve specified the mapping type "String", which we know is appropriate for this particular conversion.

The String mapping type isn’t the only one built into NHibernate; NHibernate comes with various mapping types that define default persistence strategies for primitive .NET types and certain classes, such as like DateTime.

**Built-in mapping types**

NHibernate’s built-in mapping types usually reflect the name of the .NET type they map. Sometimes you’ll have a choice of mapping types available to map a particular .NET type to the database. However, the built-in mapping types aren’t designed to perform arbitrary conversions, such as mapping a VARCHAR field value to a .NET Int32 property value. If you want this kind of functionality, you will have to define your own custom value types. We will get to that topic a little later in this chapter.

We’ll now discuss the basic types; date and time, objects, large objects, and various other built-in mapping types and show you what .NET and System.Data.DbType data types they handle. DbTypes are used to infer the data provider types (hence SQL data types).

**.NET primitive mapping types**

The basic mapping types in table A map .NET primitive types to appropriate DbTypes.

Table A Primitive types

|  |  |  |
| --- | --- | --- |
| **Mapping Type** | **.NET Type** | **System.Data.DbType** |
| Int16 | System.Int16 | DbType.Int16 |
| Int32 | System.Int32 | DbType.Int32 |
| Int64 | System.Int64 | DbType.Int64 |
| Single | System.Single | DbType.Single |
| Double | System.Double | DbType.Double |
| Decimal | System.Decimal | DbType.Decimal |
| Byte | System.Byte | DbType.Byte |
| Char | System.Char | DbType.StringFixedLength - 1 character |
| AnsiChar | System.Char | DbType.AnsiStringFixedLength - 1 character |
| Boolean | System.Boolean | DbType.Boolean |
| Guid | System.Guid | DbType.Guid |
| PersistentEnum | System.Enum (an enumeration) | The DbType for the underlying value |
| TrueFalse | System.Boolean | DbType.AnsiStringFixedLength - either 'T' or 'F' |
| YesNo | System.Boolean | DbType.AnsiStringFixedLength - either 'Y' or 'N' |

You’ve probably noticed that your database doesn’t support some of the DbTypes listed in table A. However, ADO.NET provides a partial abstraction of vendor-specific SQL data types, allowing NHibernate to work with ANSI-standard types when executing data manipulation language (DML). For database-specific DDL generation, NHibernate translates from the ANSI-standard type to an appropriate vendor-specific type, using the built-in support for specific SQL dialects. (You usually don’t have to worry about SQL data types if you’re using NHibernate for data access and data schema definition.)

NHibernate supports a number of mapping types coming from Hibernate for compatibility (useful for those coming over from Hibernate or using Hibernate tools to generate hbm.xml files). Table B lists the additional names of NHibernate mapping types.

Table B Additional names of NHibernate mapping types

|  |  |
| --- | --- |
| **Mapping type** | **Additional name** |
| Binary | binary |
| Boolean | boolean |
| Byte | byte |
| Character | character |
| CultureInfo | locale |
| DateTime | datetime |
| Decimal | big\_decimal |
| Double | double |
| Guid | guid |
| Int16 | short |
| Int32 | int |
| Int32 | integer |
| Int64 | long |
| Single | float |
| String | string |
| TrueFalse | true\_false |
| Type | class |
| YesNo | yes\_no |

From this table, you can see that writing type="integer" or type="int" is identical to type="Int32". Note that this table contains many mapping types that will be discussed in the following sections.

**Date/time mapping types**

Table C lists NHibernate types associated with dates, times, and timestamps. In your domain model, you may choose to represent date and time data using either System.DateTime or System.TimeSpan. As they have different purposes, the choice should be easy.

Table C Date and time typesExcerptOpenSourceSOAch5-6.doc

|  |  |  |
| --- | --- | --- |
| **Mapping Type** | **.NET Type** | **System.Data.DbType** |
| DateTime | System.DateTime | DbType.DateTime - ignores the milliseconds |
| Ticks | System.DateTime | DbType.Int64 |
| TimeSpan | System.TimeSpan | DbType.Int64 |
| Timestamp | System.DateTime | DbType.DateTime - as specific as database supports |

**Object mapping types**

All .NET types in tables A and C are value types (i.e. derived from System.ValueType). This means that  
they can’t be null; unless you use the .NET 2.0 Nullable<T> structure or the Nullables add-in, as  
discussed in the next section. Table D lists NHibernate types for handling .NET types derived from  
System.Object (which can store null values).

Table D Nullable object typesExcerptOpenSourceSOAch5-6.doc

|  |  |  |
| --- | --- | --- |
| **Mapping Type** | **.NET Type** | **System.Data.DbType** |
| String | System.String | DbType.String |
| AnsiString | System.String | DbType.AnsiString |

This table is completed by tables E and F which also contain nullable mapping types.

**Large object mapping types**

Table E lists NHibernate types for handling binary data and large objects. Note that none of these types may be used as the type of an identifier property.

Table E Binary and large object typesExcerptOpenSourceSOAch5-6.doc

|  |  |  |
| --- | --- | --- |
| **Mapping Type** | **.NET Type** | **System.Data.DbType** |
| Binary | System.Byte[] | DbType.Binary |
| BinaryBlob | System.Byte[] | DbType.Binary |
| StringBlob | System.String | DbType.String |
| Serializable | Any System.Object marked with SerializableAttribute | DbType.Binary |

BinaryBlob and StringClob are mainly supported by SQL Server. They can have a very large size and are fully loaded in memory. This can be a performance killer if used to store very large objects. So use this feature carefully. Note that you must set the NHibernate property "prepare\_sql" to "true" to enable this feature.

You can find up-to-date design patterns and tips for large object usage on the NHibernate website.

**Various CLR mapping types**

Table F lists NHibernate types for various other types of the CLR that may be represented as DbType.Strings in the database.

Table F Other CLR-related typesExcerptOpenSourceSOAch5-6.doc

|  |  |  |
| --- | --- | --- |
| **Mapping Type** | **.NET Type** | **System.Data.DbType** |
| CultureInfo | System.Globalization.CultureInfo | DbType.String - 5 chars for culture |
| Type | System.Type | DbType.String holding Assembly Qualified Name |

Certainly, <property> isn’t the only NHibernate mapping element that has a type attribute.

**Using mapping types**

All of the basic mapping types may appear almost anywhere in the NHibernate mapping document, on normal property, identifier property, and other mapping elements.

The <id>, <property>, <version>, <discriminator>, <index>, and <element> elements all define an attribute named type. (There are certain limitations on which mapping basic types may function as an identifier or discriminator type, however.)

You can see how useful the built-in mapping types are in this mapping for the BillingDetails class:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.<class name="BillingDetails"

02.table="BILLING\_DETAILS"

03.lazy="false"

04.discriminator-value="0">

05.

06.<id name="Id" type="Int32" column="BILLING\_DETAILS\_ID">

07.<generator class="native"/>

08.</id>

09.

10.<discriminator type="Char" column="TYPE"/>

11.

12.<property name="Number" type="String"/>

13....

14.</class>

The BillingDetails class is mapped as an entity. Its discriminator, id, and Number properties are value typed, and we use the built-in NHibernate mapping types to specify the conversion strategy.

It’s often not necessary to explicitly specify a built-in mapping type in the XML mapping document. For instance, if you have a property of .NET type System.String, NHibernate will discover this using reflection and select String by default. We can easily simplify the previous mapping example:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.<class name="BillingDetails"

02.table="BILLING\_DETAILS"

03.lazy="false"

04.discriminator-value="0">

05.

06.<id name="Id" column="BILLING\_DETAILS\_ID">

07.<generator class="native"/>

08.</id>

09.

10.<discriminator type="Char" column="TYPE"/>

11.

12.<property name="Number"/>

13.....

14.</class>

For each of the built-in mapping types, a constant is defined by the class NHibernate. NHibernateUtil. For example, NHibernate.String represents the String mapping type. These constants are useful for query parameter binding, as discussed in more detail in chapter 8:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.session.CreateQuery("from Item i where i.Description like :desc")

2..SetParameter("desc", desc, NHibernate.String)

3..List();

These constants are also useful for programmatic manipulation of the NHibernate mapping metamodel, as discussed in chapter 3.

Of course, NHibernate isn’t limited to the built-in mapping types; you can create your own custom mapping types for handling certain scenarios. We’ll take a look this next, and explain how the mapping type system is a central to NHibernates flexibility.

**Creating custom mapping types**

Object-oriented languages like C# make it easy to define new types by writing new classes. Indeed, this is a fundamental part of the definition of object orientation. If you were limited to the predefined built-in NHibernate mapping types when declaring properties of persistent classes, you’d lose much of C#’s expressiveness. Furthermore, your domain model implementation would be tightly coupled to the physical data model, since new type conversions would be impossible.

In order to avoid that, NHibernate provides a very powerful feature called custom mapping types. NHibernate provides two user-friendly interfaces that applications may use when defining new mapping types, the first being NHibernate.UserTypes.IUserType. IUserType is suitable for most simple cases and even for some more complex problems. Let’s use it in a simple scenario.

Our Bid class defines an Amount property and our Item class defines an InitialPrice property, both monetary values. So far, we’ve only used a simple System.Double to represent the value, mapped with Double to a single DbType.Double column.

Suppose we wanted to support multiple currencies in our auction application and that we had to refactor the existing domain model for this change. One way to implement this change would be to add new properties to Bid and Item: AmountCurrency and InitialPriceCurrency. We would then map these new properties to additional VARCHAR columns with the built-in String mapping type. Imagine if we had currency stored in 100 places, this would be lots of changes. We hope you never use this approach!

**Creating an implementation of IUserType**

Instead, we should create a MonetaryAmount class that encapsulates both currency and amount. This is a class of the domain model and doesn’t have any dependency on NHibernate interfaces:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.[Serializable]

02.public class MonetaryAmount

03.{

04.private readonly double value;

05.private readonly string currency;

06.

07.public MonetaryAmount(double value, string currency)

08.{

09.this.value = value;

10.this.currency = currency;

11.}

12.

13.public double Value { get { return value; } }

14.public string Currency { get { return currency; } }

15.

16.public override bool Equals(object obj) { ... }

17.public override int GetHashCode() { ... }

18.}

We’ve also made life simpler by making MonetaryAmount an immutable class, meaning it can’t be changed after it’s instantiated. We would have to implement Equals() and GetHashCode() to complete the class - but there is nothing special to consider here aside that they must be consistent, and GetHashCode() should return mostly unique numbers.

We will use this new MonetaryAmount to replace the Double, as defined on the InitialPrice property for Item. We would benefit by using this new class in other places, such as the Bid.Amount.

The next challenge is in mapping our new MonetaryAmount properties to the database. Suppose we’re working with a legacy database that contains all monetary amounts in USD. Our new class means our application code is no longer restricted to a single currency, but it will take time to get the changes done by the database team. Until this happens, we’d like to store just the Amount property of MonetaryAmount to the database. Because we can’t store the currency yet, we’ll convert all Amounts to USD before we save them, and from USD when we load them.

The first step to handling this is to tell NHibernate how to handle our Monetarymount type. To do this, we create a MonetaryAmountUserType class that implements the NHibernate interface IUserType. Our custom mapping type is shown in listing A.

Listing A Custom mapping type for monetary amounts in USD

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.using System;

02.using System.Data;

03.using NHibernate.UserTypes;

04.

05.public class MonetaryAmountUserType : IUserType {

06.

07.private static readonly NHibernate.SqlTypes.SqlType[] SQL\_TYPES =

08.{ NHibernateUtil.Double.SqlType };

09.

10.public NHibernate.SqlTypes.SqlType[] SqlTypes { |1

11.get { return SQL\_TYPES; }

12.}

13.public Type ReturnedType { get { return typeof(MonetaryAmount); } } |2

14.

15.public new bool Equals( object x, object y ) { |3

16.if ( object.ReferenceEquals(x,y) ) return true;

17.if (x == null || y == null) return false;

18.return x.Equals(y);

19.}

20.public object DeepCopy(object value) { return value; } |4

21.

22.public bool IsMutable { get { return false; } } |5

23.

24.public object NullSafeGet(IDataReader dr, string[] names, object owner){ |6

25.object obj = NHibernateUtil.Double.NullSafeGet(dr, names[0]);

26.if ( obj==null ) return null;

27.double valueInUSD = (double) obj;

28.return new MonetaryAmount(valueInUSD, "USD");

29.}

30.

31.public void NullSafeSet(IDbCommand cmd, object obj, int index) { |7

32.if (obj == null) {

33.((IDataParameter)cmd.Parameters[index]).Value = DBNull.Value;

34.} else {

35.MonetaryAmount anyCurrency = (MonetaryAmount)obj;

36.MonetaryAmount amountInUSD = MonetaryAmount.Convert( anyCurrency,"USD" );

37.((IDataParameter)cmd.Parameters[index]).Value = amountInUSD.Value;

38.}

39.}

40.

41.public static MonetaryAmount Convert( MonetaryAmount m, stringtargetCurrency)

42.{

43.... |8

44.}

45.}

The SqlTypes property tells NHibernate what SQL column types to use for DDL schema generation, as seen in #1. The types are subclasses of NHibernate.SqlTypes.SqlType. Notice that this property returns an array of types. An implementation of IUserType may map a single property to multiple columns, but our legacy data model only has a single Double.

In #2, we can see that ReturnedType tells NHibernate what .NET type is mapped by this IUserType.

The IUserType is responsible for dirty-checking property values (#3). The Equals() method compares the current property value to a previous snapshot and determines whether the property is dirty and must by saved to the database.

The IUserType is also partially responsible for creating the snapshot in the first place, as shown in #4. Since MonetaryAmount is an immutable class, the DeepCopy() method returns its argument. In the case of a mutable type, it would need to return a copy of the argument to be used as the snapshot value. This method is also called when an instance of the type is written to or read from the second level cache.

NHibernate can make some minor performance optimizations for immutable types. The IsMutable (#5) property tells NHibernate that this type is immutable.

The NullSafeGet() method shown near #6 retrieves the property value from the ADO.NET IDataReader. You can also access the owner of the component if you need it for the conversion. All database values are in USD, so you have to convert the MonetaryAmount returned by this method before you show it to the user.

In #7, the NullSafeSet() method writes the property value to the ADO.NET IDbCommand. This method takes whatever currency is set and converts it to a simple Double USD value before saving.

Note that, for briefness, we haven’t provided a Convert function as shown in #8. If we were to implement it, it would have code that converts between various currencies.

Mapping the InitialPrice property of Item can be done as follows:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.<property name="InitialPrice"

2.column="INITIAL\_PRICE"

3.type="NHibernate.Auction.CustomTypes.MonetaryAmountUserType, NHibernate.Auction"/>

This is the simplest kind of transformation that an implementation of IUserType could perform. It takes a Value Type class and maps it to a single database column. Much more sophisticated things are possible; a custom mapping type could perform validation, it could read and write data to and from an Active Directory, or it could even retrieve persistent objects from a different NHibernate ISession for a different database. You’re limited mainly by your imagination and performance concerns!

In a perfect world, we’d prefer to represent both the amount and currency of our monetary amounts in the database, so we’re not limited to storing just USD. We could still use an IUserType for this, but it’s limited; If an IUserType is mapped with more than one property, we can’t use them our HQL or Criteria queries. The NHibernate query engine wouldn’t know anything about the individual properties of MonetaryAmount. You still access the properties in your C# code (MonetaryAmount is just a regular class of the domain model, after all), but not in NHibernate queries.

To allow for a custom value type with multiple properties that can be accessed in queries, we should use the ICompositeUserType interface. This interface exposes the properties of our MonetaryAmount to NHibernate.

**Creating an implementation of ICompositeUserType**

To demonstrate the flexibility of custom mapping types, we won’t have to change our MonetaryAmount domain model class at all—we change only the custom mapping type, as shown in listing B.

Listing B Custom mapping type for monetary amounts in new database schemas

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.using System;

02.using System.Data;

03.using NHibernate.UserTypes;

04.

05.public class MonetaryAmountCompositeUserType : ICompositeUserType {

06.

07.public Type ReturnedClass { get { return typeof(MonetaryAmount); } }

08.public new bool Equals( object x, object y ) {

09.if ( object.ReferenceEquals(x,y) ) return true;

10.if (x == null || y == null) return false;

11.return x.Equals(y);

12.}

13.public object DeepCopy(object value) { return value; }

14.public bool IsMutable { get { return false; } }

15.

16.public object NullSafeGet(IDataReader dr, string[] names, NHibernate.Engine.ISessionImplementor session, object owner) {

17.object obj0 = NHibernateUtil.Double.NullSafeGet(dr, names[0]);

18.object obj1 = NHibernateUtil.String.NullSafeGet(dr, names[1]);

19.if ( obj0==null || obj1==null ) return null;

20.double value = (double) obj0;

21.string currency = (string) obj1;

22.return new MonetaryAmount(value, currency);

23.}

24.

25.public void NullSafeSet(IDbCommand cmd, object obj, int index, NHibernate.Engine.ISessionImplementor session) {

26.if (obj == null) {

27.((IDataParameter)cmd.Parameters[index]).Value = DBNull.Value;

28.((IDataParameter)cmd.Parameters[index+1]).Value = DBNull.Value;

29.} else {

30.MonetaryAmount amount = (MonetaryAmount)obj;

31.((IDataParameter)cmd.Parameters[index]).Value = amount.Value;

32.((IDataParameter)cmd.Parameters[index+1]).Value = amount.Currency;

33.}

34.}

35.

36.public string[] PropertyNames { |1

37.get { return new string[] { "Value", "Currency" }; }

38.}

39.public NHibernate.Type.IType[] PropertyTypes { |2

40.get { return new NHibernate.Type.IType[] {

41.NHibernateUtil.Double, NHibernateUtil.String }; }

42.}

43.public object GetPropertyValue(object component, int property) { |3

44.MonetaryAmount amount = (MonetaryAmount) component;

45.if (property == 0)

46.return amount.Value;

47.else

48.return amount.Currency;

49.}

50.public void SetPropertyValue(object comp, int property, object value) { |4

51.throw new Exception("Immutable!");

52.}

53.public object Assemble(object cached, |5

54.NHibernate.Engine.ISessionImplementor session, object owner) {

55.return cached;

56.}

57.public object Disassemble(object value, |6

58.NHibernate.Engine.ISessionImplementor session) {

59.return value;

60.}

61.}

#1 shows how an implementation of ICompositeUserType has its own properties, defined by PropertyNames.

Similarly, the properties each have their own type, as defined by PropertyTypes (#2).

The GetPropertyValue() method, shown in #3, returns the value of an individual property of the MonetaryAmount.

Since MonetaryAmount is immutable, we can’t set property values individually (see #4) This isn’t a problem because this method is optional anyway.

In #5, the Assemble() method is called when an instance of the type is read from the second-level cache.

The Disassemble() method is called when an instance of the type is written to the second-level cache, as shown in #6.

The order of properties must be the same in the PropertyNames, PropertyTypes, and GetPropertyValues() methods. The InitialPrice property now maps to two columns, so we declare both in the mapping file. The first column stores the value; the second stores the currency of the MonetaryAmount. Note that the order of columns must match the order of properties in your type implementation:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.<property name="InitialPrice"type="NHibernate.Auction.CustomTypes.MonetaryAmountCompositeUserType, NHibernate.Auction">

2.<column name="INITIAL\_PRICE"/>

3.<column name="INITIAL\_PRICE\_CURRENCY"/>

4.</property>

In a query, we can now refer to the Amount and Currency properties of the custom type, even though they don’t appear anywhere in the mapping document as individual properties:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.from Item i

2.where i.InitialPrice.Value &gt; 100.0

3.and i.InitialPrice.Currency = 'XAF'

In this example we’ve expanded the buffer between the .NET object model and the SQL database schema with our custom composite type. Both representations can now handle changes more robustly.

If implementing custom types seems complex, relax; you rarely need to use a custom mapping type. An alternative way to represent the MonetaryAmount class is to use a component mapping, as in section 4.4.2, “Using components.” The decision to use a custom mapping type is often a matter of taste.

There are few more interfaces that can be used to implement custom types; they are introduced in  
the next section.

**Other interfaces to create custom mapping types**

You may find that the interfaces IUserType and ICompositeUserType do not allow you to easily add  
more features to your custom types. In this case, you will need to use some of the other interfaces  
which are in the NHibernate.UserTypes namespace:

The IParameterizedType interface allows you to supply parameters to your custom type in the  
mapping file. This interface has a unique method: SetParameterValues(IDictionary parameters)  
that will be called at the initialization of your type. Here is an example of mapping providing a  
parameter:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.<property name="Price">

2.<type name="NHibernate.Auction.CustomTypes.MonetaryAmountUserType">

3.<param name="DefaultCurrency">Euro</param>

4.</type>

5.</property>

This mapping tells the custom type to use Euro as currency if it isn’t specified.

The IEnhancedUserType interface makes it possible to implement a custom type that can be marshalled to and from its string representation. This functionality allows this type to be used as identifier or discriminator type. To create a type that can be used as version, you must implement the IUserVersionType interface.

The INullableUserType interface allows you to interpret non-null values in a property as null in the database. When using dynamic-insert or dynamic-update, fields identified as null will not be inserted or updated. This information may also be used when generating the where clause of the SQL command when optimistic locking is enabled.

The last interface is different from the previous because it is meant to implement user defined collection types: IUserCollectionType. For more details, take a look at the implementation NHibernate.Test.UserCollection.MyListType in the source code of NHibernate.

Now, let’s look at an extremely important application of custom mapping types. Nullable types are found in almost all enterprise applications.

**Using Nullable types**

With .NET 1.1, primitive types can not be null; but this is no longer the case in .NET 2.0. Let’s say that we want to add a DismissDate to the class User. As long as a user is active, its DismissDate should be null. But the System.DateTime struct can not be null. And we don’t want to use some "magic" value to represent the null state. With .NET 2.0 (and 3.5 of course), you can simply write:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.public class User

02.{

03....

04.private DateTime? dismissDate;

05.public DateTime? DismissDate

06.{

07.get { return dismissDate; }

08.set { dismissDate = value; }

09.}

10....

11.}

We omit other properties and methods because we focus on the nullable property. And no change is required in the mapping.

If you work with .NET 1.1, the Nullables add-in (in the NHibernateContrib package for versions prior to NHibernate 1.2.0) contains a number of custom mapping types which allow primitive types to be null. For our previous case, we can use the Nullables.NullableDateTime class:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

01.using Nullables;

02.

03.[Class]

04.public class User {

05....

06.private NullableDateTime dismissDate;

07.

08.[Property]

09.public NullableDateTime DismissDate

10.{

11.get { return dismissDate; }

12.set { dismissDate = value; }

13.}

14....

15.}

The mapping is quite straightforward:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.<class name="Example.Person, Example">

2....

3.<property name="DateOfBirth"type="Nullables.NHibernate.NullableDateTimeType, Nullables.NHibernate" />

4.</class>

It is important to note that, in the mapping, the type of DismissDate must be Nullables.NHibernate.NullableDateTimeType (from the file Nullables.NHibernate.dll). This type is a wrapper used to translate Nullables types from/to database types. But if when using the NHibernate.Mapping.Attributes library, this operation is automatic, that’s why we just had to put the attribute [Property].

The NullableDateTime type behaves exactly like System.DateTime; there are even implicit operators to easily interact with it. The Nullables library contains nullable types for most .NET primitive types supported by NHibernate. You can find more details in NHibernate documentation.

**Using enumerated types**

An enumeration (enum) is a special form of value type, which inherits from System.Enum and supplies alternate names for the values of an underlying primitive type.

For example, the Comment class defines a Rating. If you recall, in our CaveatEmptor application, users are able to give other comments about other users. Instead of using a simple int property for the rating, we create an enumeration:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.public enum Rating {

2.Excellent,

3.Ok,

4.Low

5.}

We then use this type for the Rating property of our Comment class. In the database, ratings would be represented as the type of the underlying value. In this case (and by default), it is Int32. And that’s all we have to do. We may specify type="Rating" in our mapping, but it is optional; NHibernate can use reflection to find this.

One problem you might run into is using enumerations in NHibernate queries. Consider the following query in HQL that retrieves all comments rated “Low”:

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.IQuery q = session.CreateQuery("from Comment c where c.Rating = Rating.Low");

This query doesn’t work, because NHibernate doesn’t know what to do with Rating.Low and will try to use it as a literal. We have to use a bind parameter and set the rating value for the comparison dynamically (which is what we need for other reasons most of the time):

[view source](http://dotnet.dzone.com/articles/understanding-nhibernate-type#viewSource)

[print](http://dotnet.dzone.com/articles/understanding-nhibernate-type#printSource)[?](http://dotnet.dzone.com/articles/understanding-nhibernate-type#about)

1.IQuery q = session.CreateQuery("from Comment c where c.Rating = :rating");

2.q.SetParameter("rating", Rating.Low, NHibernateUtil.Enum(typeof(Rating));

The last line in this example uses the static helper method NHibernateUtil.Enum() to define the NHibernate Type, a simple way to tell NHibernate about our enumeration mapping and how to deal with the Rating.Low value.

We’ve now discussed all kinds of NHibernate mapping types: built-in mapping types, user-defined custom types, and even components. They’re all considered value types, because they map objects of value type (not entities) to the database. With a good understanding of what value types are, and how they are mapped, you can now move on to the more complex issue of collections of value typed instances.

Legacy

**Article Resources:**

application/pdf icon

[NHibernate Article 1.pdf](http://dotnet.dzone.com/sites/all/files/NHibernate%20Article%201.pdf)

Published at DZone with permission of its author, [Alvin Ashcraft](http://dotnet.dzone.com/users/alashcraft).

*(Note: Opinions expressed in this article and its replies are the opinions of their respective authors and not those of DZone, Inc.)*

Tags:

* [NHibernate](http://dotnet.dzone.com/category/tags/nhibernate)

* [.NET & Windows](http://dotnet.dzone.com/category/dzone-taxonomy/net-windows)

* [Tools & Methods](http://dotnet.dzone.com/category/dzone-taxonomy/tools-methods)

**Comments**

[http://cdn.dzone.com/sites/all/themes/dzone/images/avatars/default_48.gif](http://dotnet.dzone.com/users/amolr212)

A Developer replied on Mon, 2011/08/15 - 5:07am

I tried to create a custom type for one of the DB2 Time data type. following are the classes: using System; using System.Collections.Generic; using System.Linq; using System.Text; using System.Data; using NHibernate; using NHibernate.UserTypes; namespace IJM.LT.EnfinRASHavens.Domain { [Serializable] public class DB2Time { private readonly string time; public DB2Time(string time) { this.time = time; } public string Time { get { return time; } } public override bool Equals(object obj) { return base.Equals(obj); } public override int GetHashCode() { return base.GetHashCode(); } } } ///teh custome type class using System; using System.Data; using NHibernate; using NHibernate.UserTypes; namespace IJM.LT.EnfinRASHavens.Domain { public class DB2TimeType : IUserType { private static readonly NHibernate.SqlTypes.SqlType[] SQL\_TYPES = {NHibernateUtil.String.SqlType}; public NHibernate.SqlTypes.SqlType[] SqlTypes { get { return SQL\_TYPES; } } public Type ReturnedType { get { return typeof(DB2Time); } } public new bool Equals(object x, object y) { if (object.ReferenceEquals(x, y)) return true; if (x == null || y == null) return false; return x.Equals(y); } public object DeepCopy(object value) { return value; } public bool IsMutable { get { return false; } } public object NullSafeGet(IDataReader dr, string[] names, object owner){ object obj = NHibernateUtil.String.NullSafeGet(dr, names[0]); if ( obj==null ) return null; string timeInDB2 = (string) obj; return new DB2Time(timeInDB2); } public void NullSafeSet(IDbCommand cmd, object obj, int index) { if (obj == null) { ((IDataParameter)cmd.Parameters[index]).Value = DBNull.Value; } else { DB2Time anyTime = (DB2Time)obj; //MonetaryAmount amountInUSD = MonetaryAmount.Convert( anyCurrency, "USD" ); ((IDataParameter)cmd.Parameters[index]).Value = anyTime.Time; } } public object Replace(object original, object target, object owner) { return original; } public object Assemble(object cached, object owner) { return cached; } public object Disassemble(object value) { return value; } public int GetHashCode(object x) { return x == null ? typeof(string).GetHashCode() + 473 : x.GetHashCode(); } // public static MonetaryAmount Convert(MonetaryAmount m, string targetCurrency) //{ // ... |8 //} } } while fetching the data, I got the following error: Unable to cast object of type 'IJM.LT.EnfinRASHavens.Domain.DB2Time' to type 'IJM.LT.EnfinRASHavens.Domain.DB2TimeType'. I guess, I am getting this error on NullSafeGet where I am returning return new DB2Time(timeInDB2); Please provide a solution to this problem as I am really stuck with this.

* [Login](http://dotnet.dzone.com/user/login?destination=node%2F13121%23comment-form) or [register](http://dotnet.dzone.com/user/register?destination=node%2F13121%23comment-form) to post comments