

NHibernate explained

Table of contents

Disclaimer	- 2 -
Additional resources.....	- 3 -
NHibernate by example.....	- 4 -
The scenario	- 5 -
Mapping the schema.....	- 6 -
Configuring the session factory.....	- 10 -
Performing basic crud operations.....	- 11 -
Querying data using ICriteria.....	- 13 -
ICriteria API short examples.....	- 13 -
Using query by example (QBE).....	- 15 -
Querying for exactly the needed data.....	- 17 -
Querying using IQuery.....	- 17 -
Performing queries	- 17 -
IQuery API short examples	- 18 -
Performance consideration.....	- 20 -
Mapping entities.....	- 20 -
Loading data using ISession.Get or ICriteria	- 21 -
Loading data using IQuery	- 21 -

License

Copyright © Dragos Nuta, 2005

This product is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

You should have received a copy of the GNU Lesser General Public License along with this product; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Disclaimer

DOCUMENTATION IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE DOCUMENTATION IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. THE ENTIRE RISK AS TO THE QUALITY, ACCURACY, AND PERFORMANCE OF THE DOCUMENTATION IS WITH YOU. SHOULD ANY DOCUMENTATION PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL WRITER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY DOCUMENTATION IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER IN TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL THE INITIAL WRITER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF DOCUMENTATION, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER DAMAGES OR LOSSES ARISING OUT OF OR RELATING TO THE USE OF THE DOCUMENTATION, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES.

Additional resources

Resources for NHibernate 1.x and Hibernate 2.1:

- “Hibernate in Action” written by Christian Bauer and Gavin King (members of the Hibernate team)
- <http://wiki.nhibernate.org/>
- www.nhibernate.org
- forum.hibernate.org
- <http://wiki.nhibernate.org/display/NH/Documentation>
- http://www.hibernate.org/hib_docs/reference/en/html/

NHibernate by example

This chapter is a practical example of using NHibernate in an application. Here we will cover mappings, basic create/update/delete operations and querying with both IQuery and ICriteria.

As a short description of the application, the application can manage a list of products together with their details (manufacturers, suppliers) and a simple sale system that can register sales of products for a date together with the quantity sold and its price.

The entities in this application are:

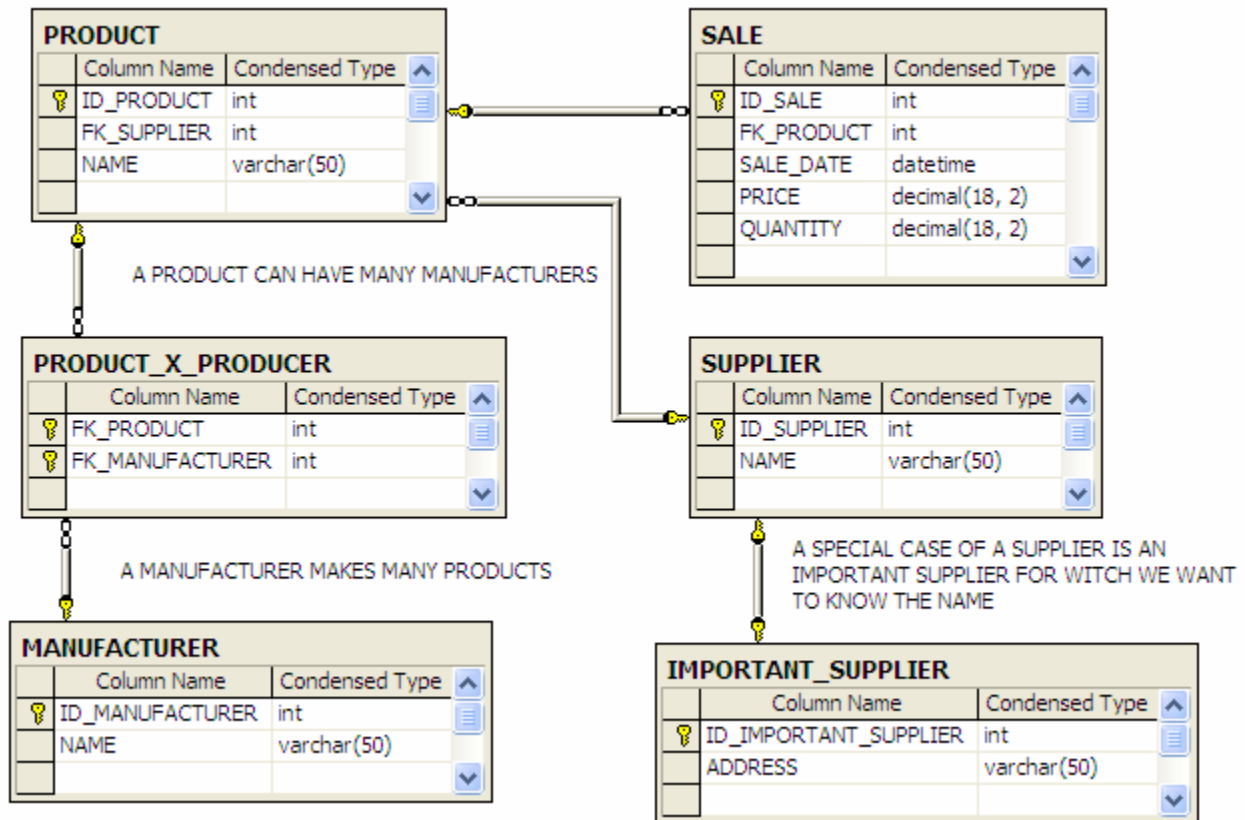
- product - each product can be manufactured by one or many manufacturers
- manufacturer - a manufacturer can make one or more products
- supplier - each product is bought from only one supplier
- there is a special kind of supplier for which we have to save the address, this is called an important supplier and that database schema represents such a supplier within another table where only the address is saved.
- sale - is defined by the product being sold, the quantity, the price and sale date.

Please note that the target of these samples is not to show how to build the best domain model out of this but to show as many kinds of approaches as possible, and then, build code to use these objects. In a real world example the mappings that we will use for this example may not suit the application needs. For example, on some associations we will not use lazy loading or with other we will use eager fetching (fetch=join), such decisions should be thoroughly analyzed before used like that as that affect the performance, the queries and others.

The scenario

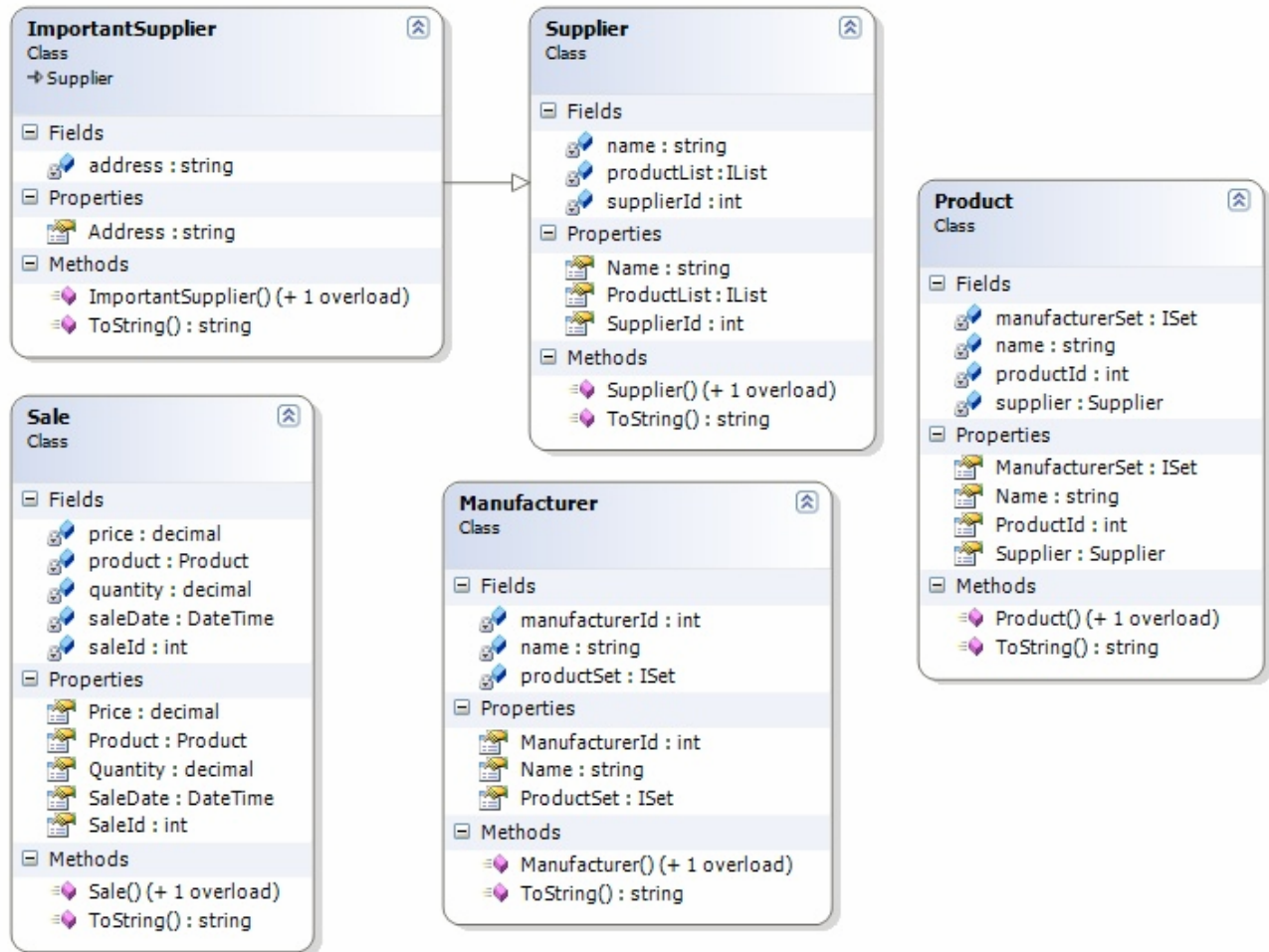
The database schema

The following schema represents the relational modeling of the previously described application.



The class diagram

There are some things to say about how this domain model is designed, one is that you will notice that each class has a specialized constructor with most of the properties. I consider this a good practice both in terms of code verbosity and because such constructors allow creating a reasonable well formed state of an entity. The purpose of overriding ToString method on each of the classes was to enable representation of an object as easy as possible for outputting data in the user interface. Of course that in a good design this should be localizable but it's not the case for this sample.



Mapping the schema

NOTE: In the following mappings all the properties have the types specified. This is not absolutely necessary because NHibernate detects the type using reflection but I consider it an improvement because it reduces the use of reflection. There are types that must be specifically set in the mappings, one of these are DateTime properties that can be: date, date and time, just time or timestamp, so here is the need to specify the type in the mapping. Also the type must be configured in the mapping when using custom types (such as the Nullable library that comes with NHibernate Contrib).

In this example that mappings will be stored in assemblies by adding them in the project as “Build Action=Embedded Resource”. You can also add the mappings at runtime to the configuration object by adding the assembly or assemblies containing mapping files. Other means of adding xml mappings to NHibernate will be presented later.

The following snippets show the mapping used for each of the entities.

Manufacturer.hbm.xml

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" auto-import="true" default-
access="nosetter.camelcase">
  <class name="OrderSystem.Objects.Manufacturer, OrderSystem.Objects" table="MANUFACTURER"
lazy="true" mutable="true">
    <id name="ManufacturerId" unsaved-value="0" column="ID_MANUFACTURER" type="System.Int32">
        <generator class="hilo"/>
    </id>
    <set name="ProductSet" table="PRODUCT_X_MANUFACTURER" cascade="none" lazy="true"
inverse="false">
        <key column="FK_MANUFACTURER" />
        <many-to-many class="OrderSystem.Objects.Product, OrderSystem.Objects"
column="FK_PRODUCT" />
    </set>
    <property name="Name" column="NAME" not-null="true" type="System.String" />
  </class>
</hibernate-mapping>
```

- The auto-import attribute set to true in the hibernate-mapping element tells NHibernate to treat entities as having unique names consequently in HQL queries only the name of the class can be specified.
- The class is mapped using assembly qualified name (type with namespace and assembly name)
- The persistent instance is lazy loaded (lazy=true) and it's mutable, meaning that insert/update on entities of this type is allowed and the session tracks changes in order to detect when an instance is dirty. If the type represents some built in system data that is not modified by the application, setting mutable=false can provide some optimization. Note that by default, NHibernate considers all classes to be mutable and also starting with v1.0.2 classes are lazy if not otherwise specified.
- The id is mapped using zero as unsaved value
- The many-to-many relation with the products is mapped using a set (Iesi.Collections.ISet). The set is configured as lazy and the inverse is set to false because this will be the updatable end while the other end of the relation will not be persistent directly.
- We configured the column in the many-to-many table that points back to the MANUFACTURER table using the *key* element. Also the content of the set is configured using the many-to-many element for which we specify the class that will be found at the other end (Product) and the column in the many-to-many table that points to the other end of the association.
- The name property uses the *property* element for mapping and here we specify the column name, the null/not null option and the type (not mandatory as stated before).

Product.hbm.xml

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" auto-import="true" default-
access="nosetter.camelcase">
  <class name="OrderSystem.Objects.Product, OrderSystem.Objects" table="PRODUCT" lazy="true"
mutable="true">
```

```

<id name="ProductId" unsaved-value="0" column="ID_PRODUCT" type="System.Int32">
  <generator class="hilo"/>
</id>
<set name="ManufacturerSet" table="PRODUCT_X_MANUFACTURER" cascade="none" lazy="true"
inverse="true">
  <key column="FK_PRODUCT" />
  <many-to-many class="OrderSystem.Objects.Manufacturer, OrderSystem.Objects"
column="FK_MANUFACTURER" />
</set>
<property name="Name" column="NAME" not-null="true" type="System.String" />
<many-to-one name="Supplier" column="FK_SUPPLIER" not-null="true" fetch="join" cascade="none" />
</class>
</hibernate-mapping>

```

- This end of the many-to-many mapping is set as inverse=true so changes to this endpoint will not be persisted. Note that cascading is applied for this association end also, the inverse=true only affects if SQL statements will be generated for this association end point.
- The reference to the Supplier that sales this item is marks as fetch=join this way, when a Product instance is loaded the Supplier will also be loaded using a outer join statement if not otherwise specified (in the next part there will be examples that show how to change the fetch behavior using ICriteria and IQuery).

Supplier.hbm.xml

```

<?xml version="1.0" encoding="utf-8"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" auto-import="true" default-
access="nosetter.camelcase">
  <class name="OrderSystem.Objects.Supplier, OrderSystem.Objects" table="SUPPLIER" lazy="true"
mutable="true">
    <id name="SupplierId" unsaved-value="0" column="ID_SUPPLIER" type="System.Int32">
      <generator class="hilo"/>
    </id>
    <bag name="ProductList" table="PRODUCT" cascade="none" fetch="join" inverse="true">
      <key column="FK_SUPPLIER" />
      <one-to-many class="OrderSystem.Objects.Product, OrderSystem.Objects" />
    </bag>
    <property name="Name" column="NAME" not-null="true" type="System.String" />
  </class>
</hibernate-mapping>

```

- Here we use a bag to hold a list of instances of the Product type. The list is loaded using an outer join and is set as inverse=true so the end-point of the relation will not be updatable. A bag is a simple collection represented by an IList in the domain model that does not ensure the items in the collection are distinct. A reason for using bags in mappings is if we need an indexer in the collection. Another type that is also an IList in the domain model is list, the list being a special kind of mapping for which we can also specify what column will represent the indexer value.

ImportantSupplier.hbm.xml

```

<?xml version="1.0" encoding="utf-8"?>

```



```

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" auto-import="true" default-
access="nosetter.camelcase">
  <joined-subclass name="OrderSystem.Objects.ImportantSupplier, OrderSystem.Objects"
extends="OrderSystem.Objects.Supplier, OrderSystem.Objects" table="IMPORTANT_SUPPLIER"
lazy="true">
    <key column="ID_IMPORTANT_SUPPLIER" />
    <property name="Address" column="ADDRESS" not-null="true" type="System.String" />
  </joined-subclass>
</hibernate-mapping>

```

- The object ImportantSupplier is mapped as a class inheriting from Supplier class. This is stated using the “extends” attribute from the joined-class element.
- This class is also marked as lazy=true. A condition for joined-subclass is that the base class and the derived one must have the same lazy state. In this case must be noted that we cannot perform a type-check on a proxy of the base class. For example if product has a lazy reference to supplier and the supplier is an ImportantSupplier the following assertion will fail:

```

bool isImportant = typeof(product.Supplier) == typeof(ImportantSupplier);
// product.Supplier will be a NHibernate Supplier proxy, inherited from Supplier
// in order to be sure of the type either we perform an eager load on the Supplier property
// or check like this
isImportant = session.Load( typeof(Supplier), product.Supplier.SupplierId ) is ImportantSupplier;

```

- We configure the key column that points to the base class, this column will be used to join the SUPPLIER and the IMPORTANT_SUPPLIER tables.
- The ImportantSupplier class will inherit all the attributes of the Supplier base class.
- In addition the the Supplier class, the ImportantSupplier defines the Address property of type string.

Sale.hbm.xml

```

<?xml version="1.0" encoding="utf-8"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" auto-import="true" default-
access="nosetter.camelcase">
  <class name="OrderSystem.Objects.Sale, OrderSystem.Objects" table="SALE" lazy="true"
mutable="true">
    <id name="SaleId" unsaved-value="0" column="ID_SALE" type="System.Int32">
      <generator class="hilo"/>
    </id>
    <property name="Price" column="PRICE" not-null="true" type="System.Decimal" />
    <many-to-one name="Product" column="FK_PRODUCT" not-null="true" fetch="select" cascade="none"
/>
    <property name="Quantity" column="QUANTITY" not-null="true" type="System.Decimal" />
    <property name="SaleDate" column="SALE_DATE" not-null="true" type="System.DateTime" />
  </class>
</hibernate-mapping>

```

- This mapping has no new concepts different from the previous mappings.

Although not provided by this example, the one-to-one mapping is similar to the many-to-one mapping with only some small changes, meaning that for the one-to-one mapping we have a

“constrained” attribute (boolean value) that if set to true will allow NHibernate to perform some loading optimizations. This attribute tells NHibernate if the relation always has an instance at the other end, this way there is no need to check at load time if the property is null or not. Otherwise, NHibernate has to check if there is something at the other end performing a separate select or a join because null cannot be wrapped within a proxy.

The classes (business entities)

The classes have no exotic form of some sort. These are all POCO (plain old C# objects) with no special base class whatsoever. Only the ImportantSupplier is inherited from the Supplier class because it was mapped that way.

What should be noted here is that all properties that can be exposed as proxies are marked as virtual (eg. Product.Supplier or Product.ManufacturerSet). If the virtual is omitted you will not be able to use them as lazy instances. Also in general, all collections should be marked as virtual because NHibernate always replaces collections with a proxy I order to detect changes to such collections.

The set collections are mapped as Iesi.Collections.ISet, the bag collections are mapped as System.Collections.IList). In the constructor the collections must be initialized (to System.Collections.ArrayList or Iesi.Collections.ListSet) in order to allow adding elements to newly created instances.

Configuring the session factory

```
class NHibernateSessionFactory
{
    private static NHibernate.Cfg.Configuration cfg = null;
    private static NHibernate.ISessionFactory sessionFactory = null;

    static NHibernateSessionFactory()
    {
        cfg = new NHibernate.Cfg.Configuration();

        // set provider & driver properties
        cfg.Properties.Add(
            "hibernate.connection.provider",
            "NHibernate.Connection.DriverConnectionProvider");
        cfg.Properties.Add(
            "hibernate.connection.driver_class",
            "NHibernate.Driver.SqlClientDriver");
        cfg.Properties.Add(
            "hibernate.dialect",
            "NHibernate.Dialect.MsSql2000Dialect");
        cfg.Properties.Add(
            "hibernate.max_fetch_depth",
            "-1"); // allows for unlimited outer joins (recommeded value is maximum 4
        cfg.Properties.Add(
            "hibernate.connection.connection_string",
            "Server=myserver;initial catalog=OrderSystem;User Id=sa;Password=");

        // here we add all the needed assemblies that contain mappings or objects
    }
}
```

```
cfg.AddAssembly(Assembly.LoadFrom("OrderSystem.Objects.dll));
```

```
//cfg.AddDirectory( new System.IO.DirectoryInfo(@"c:\myproject\mappings") );  
//cfg.AddFile(@"c:\myproject\mappings\someclass.hmb.xml");  
//cfg.AddInputStream(new System.IO.StringReader("..xml goes here.."));  
  
sessionFactory = cfg.BuildSessionFactory();  
}  
  
public static NHibernate.ISession OpenSession()  
{  
    return sessionFactory.OpenSession();  
}  
}
```

This code is an example of a utility class creating the NHibernate configuration and session factory at startup. The code configures NHibernate to work on a Microsoft SQL Server database using the specified connection string and also adds all the objects and mappings defined in the current assembly.

Some ways for adding mappings to the configuration are presented. Also keep in mind that all mappings can be modified in memory if necessary before building the session factory just after loading the mappings.

Performing basic crud operations

- Adding data using cascades and the “transitive persistence using cascades” concept.

```
public int SaveObjectAndReturnSupplier(bool makeImportant, int prodCount)  
{  
    ISession session = NHibernateSessionFactory.OpenSession();  
    ITransaction transaction = session.BeginTransaction();  
    string timeStamp = DateTime.Now.ToString("yyyyMMdd-HH:mm:ss");  
  
    // create important supplier  
    Supplier s = makeImportant ?  
        new ImportantSupplier("supplier " + timeStamp, "n/a") :  
        new Supplier("supplier " + timeStamp);  
    // create manufacturer  
    Manufacturer m = new Manufacturer("manufacturer " + timeStamp);  
    // create products and link to supplier  
    for (int i = 0; i < prodCount; i++)  
    {  
        Product p = new Product();  
        p.Name = "product " + timeStamp;  
        p.Supplier = s; s.ProductList.Add(p); // sync both ways  
        m.ProductSet.Add(p); p.ManufacturerSet.Add(m);  
    }  
    try  
    {  
        Console.WriteLine("Saving changes...");  
    }
```

```
// all changes will be persisted because of cascades set to save-update or all
```

```
    session.Save(s);  
    transaction.Commit();  
    Console.WriteLine("Save succeeded");  
}  
catch { transaction.Rollback(); Console.WriteLine("Save failed"); throw; }  
finally { session.Close(); }  
return s.SupplierId;  
}
```

- Example for listing information saved with previous example.

```
public void ListSupplierProductsAndManufacturers(int supplierId)  
{  
    Console.WriteLine("Loading results for supplier " + supplierId.ToString());  
    ISession session = NHibernateSessionFactory.OpenSession();  
    Supplier s = (Supplier)session.Get(typeof(Supplier), supplierId);  
  
    Console.WriteLine(s.Name + " is ");  
    Console.WriteLine(s is ImportantSupplier ? "important" : "normal");  
    Console.WriteLine("Products...");  
    foreach (Product p in s.ProductList)  
    {  
        Console.WriteLine("\t" + p.Name);  
        foreach (Manufacturer m in p.ManufacturerSet)  
            Console.WriteLine("\t\t" + m.Name);  
    }  
    session.Close();  
}
```

- Example for adding sales for each product sold by a supplier.

```
public void AddSaleForEachProduct(int supplierId)  
{  
    ISession session = NHibernateSessionFactory.OpenSession();  
    Supplier s = (Supplier)session.Load(typeof(Supplier), supplierId);  
    Console.WriteLine("Adding sales...");  
    foreach (Product p in s.ProductList)  
    {  
        Sale sale = new Sale(new Random().Next(), p, 1, DateTime.Now);  
        session.Save(sale);  
    }  
    Console.WriteLine("Saving sales");  
    session.Flush();  
}
```

- Calling the above functions

```
CrudTest test = new CrudTest();  
int suppld;  
suppld = test.SaveObjectAndReturnSupplier(true, 4);  
test.ListSupplierProductsAndManufacturers(suppld);
```

```
suppld = test.SaveObjectAndReturnSupplier(false, 2);
test.ListSupplierProductsAndManufacturers(suppld);
test.AddSaleForEachProduct(suppld);
```

- Output on the console after using the above routines should be this:

```
Saving changes...
Save succeeded
Loading results for supplier 1998849
supplier 20060116-174034 is important
Products...
    product 20060116-174034
        manufacturer 20060116-174034
    product 20060116-174034
        manufacturer 20060116-174034
    product 20060116-174034
        manufacturer 20060116-174034
    product 20060116-174034
        manufacturer 20060116-174034
Saving changes...
Save succeeded
Loading results for supplier 1998850
supplier 20060116-174035 is normal
Products...
    product 20060116-174035
        manufacturer 20060116-174035
    product 20060116-174035
        manufacturer 20060116-174035
Adding sales...
Saving sales
```

Querying data using ICriteria

ICriteria API short examples

The ICriteria is a powerful API that allows querying data but the following things describe the concept behind ICriteria:

- It allows querying of a single entity, using it you will always get a list of objects of a single persistent type.
- It allows adding conditional expression (for the where part of the query) in an object oriented manner using expression classes that are applied to the query. These expressions support basic comparison (equal, not equal, greater, less, greater or equal, less or equal, in, null, not null, like), logical operators (conjunction, disjunction, not) and even plain SQL conditions.
- It can perform eager fetching or lazy fetching of associations, thus overriding the default behavior defined in the mappings.
- It allows creating conditional expressions on associations using CreateAlias or CreateCriteria but both of these methods imply an inner join.
- It has a limitation when using SetFetchMode and CreateAlias/CreateCriteria on an association path because an inner join is used to connect two end points of an association and so in a master-

detail association only the masters that have details will be retrieved. This limitation can be worked around using SQL expressions.

Below there are some examples of tasks that can be performed with an ICriteria:

```
ISession session = NHibernateSessionFactory.OpenSession();
ICriteria criteria; IList result;

criteria = session.CreateCriteria(typeof(Sale));

// adding where conditions
criteria.Add(Expression.Or(
    Expression.Gt("Price", 40),
    Expression.Between("Quantity", 0, 5)
));

// adding where condition on associated classes using aliases
criteria.CreateAlias("Product", "p")
    .Add(Expression.Like("p.Name", "product%"))
    .SetFetchMode("Product.Supplier", FetchMode.Select);

// the result is => sales with price > 40 and qty between 0 and 5 where product name like 'product%'
result = criteria.List();
```

The generated query is something like:

```
SELECT this.ID_SALE as ID_SALE1_, this.FK_PRODUCT as FK_PRODUCT1_, this.PRICE as PRICE1_,
this.QUANTITY as QUANTITY1_, this.SALE_DATE as SALE_DATE1_, p.ID_PRODUCT as
ID_PRODUCT0_, p.FK_SUPPLIER as FK_SUPPL3_0_, p.NAME as NAME0_
FROM SALE this
    inner join PRODUCT p on this.FK_PRODUCT=p.ID_PRODUCT
WHERE
    (this.PRICE > @p0 or this.QUANTITY between @p1 and @p2)
    and p.NAME like @p3
```

```
criteria = session.CreateCriteria(typeof(Sale));

// adding where condition on associated classes using sub-criterias
ICriteria subCriteria = criteria
    .CreateCriteria("Product")
    .CreateCriteria("Supplier")
    .Add(Expression.Like("Name", "supplier%"));

// force sale.Product.Supplier.ProductList to be left un-initialized
criteria.SetFetchMode("Product.Supplier.ProductList", FetchMode.Select);

// sorting result set
criteria.AddOrder(NHibernate.Expression.Order.Asc("SaleDate"));
```

```
//paging
```

```
int currentPage = 0, pageSize = 5;
criteria.SetFirstResult(currentPage * pageSize).SetMaxResults(pageSize);

// locking results for upgrade
criteria.SetLockMode(LockMode.Upgrade);

// caching the entire result-set in the second level cache
criteria.SetCacheable(true).SetCacheRegion("app_special_sales");

// sales with sale.Product.Supplier.Name like 'supplier%' without retrieving entire ProductList of a supplier
// ordering by SaleDate. Retrieve 5 results starting from record 0 and lock the sales for upgrade
result = criteria.List();
```

The generated query is something like:

```
SELECT this.ID_SALE as ID_SALE1_, this.FK_PRODUCT as FK_PRODUCT1_, this.PRICE as PRICE1_,
this.QUANTITY as QUANTITY1_, this.SALE_DATE as SALE_DATE1_, p.ID_PRODUCT as
ID_PRODUCT0_, p.FK_SUPPLIER as FK_SUPPL3_0_, p.NAME as NAME0_
FROM SALE this
    inner join PRODUCT p on this.FK_PRODUCT=p.ID_PRODUCT
WHERE
    (this.PRICE > @p0 or this.QUANTITY between @p1 and @p2)
and p.NAME like @p3
```

Using query by example (QBE)

```
ISession session = NHibernateSessionFactory.OpenSession();

ICriteria criteria; IList result = null;
Product example = new Product();
example.Name = "product 20060116-174035"; // 2 results should match at this point
NHibernate.Expression.Example queryEx = Example.Create(example).IgnoreCase();

// retrieves all products, because of fetch=join with the supplier the supplier will be fetched
// in the same select, and also because supplier has the list of products mapped as
// fetch=join, for each supplier all products will be retrieved
criteria = session.CreateCriteria(typeof(Product)).Add(queryEx);
result = criteria.List(); // count in this case is 4 (2 prod with 1 supp each, each supplier has 2 prod)

// #1: using result transformer
criteria = session.CreateCriteria(typeof(Product)).Add(queryEx)
    .SetResultTransformer(CriteriaUtil.DistinctRootEntity);
result = criteria.List(); // 2 results

// #2: disabling join fetching if we don't need the supplier
criteria = session.CreateCriteria(typeof(Product)).Add(queryEx)
    .SetFetchMode("Supplier", FetchMode.Select);
result = criteria.List(); // 2 results

// #3: disabling fetching on product.Supplier.Products
criteria = session.CreateCriteria(typeof(Product)).Add(queryEx)
```

```
.SetFetchMode("Supplier.ProductList", FetchMode.Select);
```

```
result = criteria.List();
```

Before considering this example I restate that this should be taken as is, changing mappings would prevent lots of the problems but what I want is to show is how NHibernate works. Also consider that fetching using join is also controlled by the `max_fetch_depth` parameter of the configuration. At some point in a query you may see that you have unexpected behavior but it may be the result of your configuration in most cases.

Using query-by-example you can show a query the form of records you want to obtain and let NHibernate create the full parameterized query. In this example we are searching products having the name "product 20060116-174035".

The example class also allows filtering string properties using LIKE, ignore case, exclude ad-hoc properties from search and much more. Look at `NHibernate.Expressions.Example` for more details. For the current mappings, when NHibernate creates the query it takes into consideration the `fetch=join` association and includes them in the select by default. This way, for the current test set, there are two products with this name, each of them has one supplier, each supplier having in turn two products. This will cause the first query to return four results, meaning each product twice.

The approximate SQL generated is:

```
FROM PRODUCT this
left outer join SUPPLIER supplier1_
left outer join IMPORTANT_SUPPLIER [supplier1__1_]
left outer join PRODUCT productlis2_ on supplier1_.ID_SUPPLIER=productlis2_.FK_SUPPLIER
```

The first solution query shows how to obtain the correct result count using a query transformer with distinct root entity that will generate the same query as above; it will only be transformed after execution. Keep in mind that this is a solution only if we really want to have the supplier and all the products that the supplier sells. Otherwise this is just another ill query that for a large database will take forever to run for such a simple task.

The second solution excludes the supplier from the query this way obtaining an SQL like:

```
FROM PRODUCT this
```

This is a solution if we don't need the information regarding the supplier because the supplier will be loaded as a proxy. If we need the supplier and take this approach we will get to the N+1 select problem because a query will be generated each time we first access a supplier proxy.

The third solution is similar to the above but takes into consideration the case where we need the supplier data but we really don't need the whole list of products that each supplier sells. The query generated will be:

```
FROM PRODUCT this
left outer join SUPPLIER supplier1_
left outer join IMPORTANT_SUPPLIER [supplier1__1_]

```


Querying for exactly the needed data

Problem: get all sales to put in a grid, for each sale we need the product name to display it also.

```
ISession session = NHibernateSessionFactory.OpenSession();
ICriteria criteria = null; IList result = null; string buffer = null;

// approach #1: when querying products we will hit the N+1 select problem
criteria = session.CreateCriteria(typeof(Sale));
result = criteria.List();
// because product.Supplier is mapped as fetch=join the select issued
// for each product will join with the supplier table
foreach (Sale s in result) buffer = s.Product.Name;

// approach #2: get all sales with the product (no N+1 problem, also don't need supplier so don't get it)
criteria = session.CreateCriteria(typeof(Sale))
    .SetFetchMode("Product", FetchMode.Join)
    .SetFetchMode("Product.Supplier", FetchMode.Select);
result = criteria.List();
foreach (Sale s in result) buffer = s.Product.Name;
```

In the first approach we queried for all sales in the system. The foreach statement is supposed to force retrieval of product name. If you watch the SELECT statements issued by this we will get one select for all sales and one select for each product of each sale. Of course NHibernate will not retrieve twice the same product so for example if we have 3 sales of 2 products, 3 selects will be issued: one for the sales and two for the products.

```
...FROM sale
...FROM product WHERE id = 1
...FROM product WHERE id = 2
```

In the second approach we instruct NHibernate to include the product information in the criteria. The second line instructs NHibernate to perform a select on the supplier information for each product only when the supplier of the product is accessed (FetchMode.Select). If we would have removed the second fetch mode line, the select issued by the query would be of the following form:

```
... FROM sale INNER JOIN product INNER JOIN supplier INNER JOIN important_supplier
```

Because we don't need the supplier information we have managed to instruct NHibernate exactly the needed data.

```
...FROM sale INNER JOIN product
```

We could have mapped the classes appropriately but for the sake of this discussion let's suppose that we had a very good reason for the current mappings.

Querying using IQuery

Performing queries

HQL is similar in syntax with SQL but with some considerable differences:

- Allows OOP like full association traversal (eg. Order.OrderDetail.Product.Name="my product") but with some exceptions (traversing to a collection is allowed only with special functions *elements* or *indices*).
- Is case sensitive.
- Performs polymorphic queries (querying for Cat retrieve DomesticCat-s or WildCats-s if Cat is the abstract base class).
- Provides more search mechanisms.
- Can interpret constants and enumerations in the where clause (not in the select part).
- The join syntax is slightly different as with SQL - the join is made given an association path from the object to a member object or collection because NHibernate already knows the columns that define the association.

It also borrows a lot from SQL:

- Has "select...from...where...group by...having...order by" syntax.
- Performs all type of joins (left outer, right outer, inner, full and cross join).
- Supports aggregate functions and all SQL expressions (eg. arithmetic operators, in, between exists, all).
- No parameter or one parameter functions (through query substitutions).

IQuery API short examples

The following code snippets show common tasks that can be accomplished using IQuery.

As all previous examples this is the setup of the code samples:

```
ISession session = NHibernateSessionFactory.OpenSession();
IQuery query;
IList result = null;
```

An important feature of HQL is that it allows referencing properties using association paths, in this example from the sale we got the product name using s.Product.Name. Also the sample shows how to add a simple like condition. The IQuery has multiple methods for setting parameters (named or unnamed), some with specific type or for collections for entities etc.

The result of such a query is an IList containing strings as elements. If the select list contained more than one element (lets say we selecte the s.Date property also), the result would have been an IList, each element beeing in turn also an IList. This last list contains the selected row (a string and a date). As a rule, NHibernate returns IList of objects when only one element is selected otherwise returns IList of IList's.

```
// traverse an association from many part to one part with pagination
query = session.CreateQuery("select s.Product.Name from Sale s where s.Product.Name like :filterName");
query.SetString("filterName", "%");
query.SetFirstResult(5).SetMaxResults(10);
result = query.List();
```

Association traversal is not allowed when the association path contains a reference to a collection. For traversing this kind of associations, an explicit join mode must be specified (outer, inner etc.).

```
// traversing associated collections
```

```

query = session.CreateQuery("select s.ProductList.Name from Supplier s");
result = query.List(); // exception because association path contains a collection
query = session.CreateQuery("select p.Name from Supplier s inner join s.ProductList p");
result = query.List();

```

This code sample shows usage of the special HQL function *elements*. This function allows retrieval of all elements of a collection. Another special HQL function that applies to collections is *size* that allows retrieving the element count of a collection. A call to *size* will be translated into a subselect counting the collection elements.

```

// get all elements of a collection
query = session.CreateQuery(
    "select elements(m.ProductSet) from Manufacturer m where m.ManufacturerId = :id")
    .SetInt32("id", 2064385);
result = query.List();

// using size to check collection count in HQL, retrieves sales of products with only one supplier
query = session.CreateQuery("from Sale s where size(s.Product.ManufacturerSet) = 1");
result = query.List();

```

Another feature of NHibernate is how it handles the IN sql clause. You can map the content of an IN clause to a named paramter and initialize it with a collection. Be aware that setting the value of such a paramter to an empty collection will throw an exception on the database because empty IN sets are not accepted on any RDBMS

```

// get all manufactureres whose products were sold
query = session.CreateQuery(
    @"select distinct m.Name from Sale s inner join s.Product.ManufacturerSet m
    where s.Date in (:allowedDates) order by m.Name");
query.SetParameterList("allowedDate",
    new DateTime[] { new DateTime(2000, 01, 01), DateTime.Now.Date });
result = query.List();

```

All previous examples executed the query using the List method that returns an IList of objects that meet the seach criteria. Another option is to run the query using the UniqueResult. This is not equivalent to SQL TOP 1 statement because it retrieves the full result set and checks that it contains one or more instances of the same value or otherwise throws an exception. UniqueResult only applies to queries returning having a single element in the select list (entity or value, doesn't matter).

```

// retrieving scalar value using UniqueResult
query = session.CreateQuery("select max( s.Date ) from Sale s");
object uniqueResult = query.UniqueResult();

```

The eager fetching presented with the ICriteria query method is also present in HQL this time the syntax is *inner join fetch* to an association path. As a result of the folowing code snippet, the product will have the ManufacturerSet collection already initialized after running the query.

```

// eager fetching associations
query = session.CreateQuery("from Product p inner join fetch p.ManufacturerSet");
result = query.List();

```

All examples of HQL up to this one use query in-code statements. You can also store statements in NHibernate configuration file, key them and then retrieve them by key. For this you must add a line like `<query name="suppliers">from Supplier</query>` to one of your mapping files.

```
// using named queries
// <query name="suppliers">from Supplier</query>
query = session.GetNamedQuery("suppliers");
result = query.List();
```

The code snippet below shows two new concepts. First concept is instructing the query to retrieve a list of objects of a custom type instead of the common `IList` mode using the *select new* statement. Classes that are used with this types of queries must be also declared in one of your mapping files using the syntax `<import class="OrderSystem.Objects.ReportObjects.SaleDetails, OrderSystem.Objects"/>`. After declaring the class in the mapping you can use it in such queries.

Another new concept the snippet below is executing the query and obtaining an *IEnumerable* result.

```
// using select new style queries
query = session.CreateQuery(
    "select new SaleDetails(s.Product.Name, s.SaleDate, s.Price, s.Quantity) from Sale s");
IEnumerator enumerator = query.Enumerable().GetEnumerator();
while (enumerator.MoveNext())
    { object item = enumerator.Current; }
```

Performance consideration

As any good thing, NHibernate comes with its drawbacks, and often there is the danger that all the great features (transitive persistence, dirty-tracking, lazy loading) that it has to offer turn in dangerous, un-optimized, bug-prone code. You can't talk about "the architecture" that you should use when using NHibernate because it depends on each situation, but you can surely talk about some guide-lines.

General rule is keeping the sessions as short as possible. Reasons for this are that sessions imply connections to database, transactions and memory occupied by objects associated with the session.

Mapping entities

Almost always it's a good thing to start with everything being lazy (classes, collections) and then, by analyzing requirements, decide what should be eagerly fetched and what not. The mapping examples I showed in this examples were not provided as guidelines for mapping but merely as examples to work with when performing queries and a presentation of the variety of things that NHibernate provides.

As it was presented in the previous code samples, eager fetching can be either a blessing or a curse if not all aspects of the application are taken into consideration. For example the association between supplier and product in the above examples was setup as `fetch=join`, this way, transparently, each time we retrieved a supplier (by session load/get or by `ICriteria`) the entire list was loaded. This thing also happened each time we instructed a criteria to retrieve a supplier association eagerly. This could lead to both unexpected behavior (the results count is larger than expected because of the cartesian product) or not optimized queries that, at some point, will take forever to execute.

Another thing is that start-up time can be improved if we provide as much information as possible so that Reflection use is minimized. Of course the configuration is usually loaded only once but with ASP.NET, if the process is restarted because of encountered problems or occupied memory, the configuration object will be lost and will be again recreated.

Loading data using ISession.Get or ICriteria

When loading instances they will be tracked down by the session for its entire life-cycle. So when loading large sets of data through these methods you have two large performance issues:

- Computing effort for checking dirty state for objects.
- Memory occupied by all those instances in the session pool.

The first point can be tweaked either by evicting unnecessary objects from the session, or marking the classes as mutable="false" in the mappings. Further, evicting the objects from the session will cause another round-trip to the database if the entity will be requested again later (considering that it is much cheaper to keep instances in memory than to hit the DB). Marking classes as mutable="false" is not always possible, only hard-coded data that never changes can be defined this way because not insert/updates/delete will be possible on an instance of such a class.

Another thing to tweak for ICriteria is the fetch mode for association. You can force retrieval of entire objects or force skipping retrieval for others using ICriteria.SetFetchMode. Be careful to the logic of such actions, because, for example, setting join fetch on Product.Sale without setting fetch join to Product will produce no result if product is not also fetched eagerly. And also SetFetchMode does not work on an association where CreateAlias or CreateCriteria is used.

Also, consider that the cost of select by id is smaller than retrieving by using join, but on the other hand, executing N selects by id is more expensive than using one big select with join. When loading a Product for example, you can load the Supplier using a separate select but if you also want to access the sales is better to use join fetching on the original load or use a separate query to load all sales at once rather than allowing NHibernate to fetch each sale in turn.

So in terms of database optimization, you should try to load exactly the amount of data needed as compact as possible but remember that join fetching is limited by the setting of the *max_fetch_depth* configuration variable.

Loading data using IQuery

IQuery has the advantage that it can retrieve just values so there is nothing to manage state for, and also, if needed you can cache results using the same syntax as with the ICriteria (SetCacheable, SetCacheRegion) and so you can minimize round-trips to database when the results are needed more often. Another advantage of using IQuery to select just values instead of entities is that there is no danger that at some point, someone accesses some un-initialized collections that perform a lot of select statements in background.

When using IQuery to load object graphs the same consideration for ICriteria must be taken into account.

With regard to database optimization, it is a good practice to create queries that have a constant form. For example, we have to problem if retrieving all sales filtered by the name and/or by the price:

```

ISession session = NHibernateSessionFactory.OpenSession();

int priceFilter = 0; string nameFilter = null;
IQuery query = null; IList result = null;

// solution #1
ArrayList filters = new ArrayList();
string statement = "select s.Product.Name, s.SaleDate, s.Price, s.Quantity from Sale s";
if (priceFilter != -1) filters.Add("s.Price > :priceFilter");
if (nameFilter != null) filters.Add("s.Product.Name like :nameFilter");
for (int i = 0; i < filters.Count; i++)
{
    if (i == 0)
        statement += " where ";
    else
        statement += " and ";
    statement += (string)filters[i];
}
query = session.CreateQuery(statement);
if (priceFilter != -1) query.SetDecimal("priceFilter", priceFilter);
if (nameFilter != null) query.SetString("nameFilter", nameFilter);
result = query.List();

// solution #2
query = session.CreateQuery(
    @"select s.Product.Name, s.SaleDate, s.Price, s.Quantity from Sale s where
    (:priceFilter = -1 or s.Price > :priceFilter)
    and (:nameFilter is null or s.Product.Name = :nameFilter)");
query.SetDecimal("priceFilter", priceFilter);
query.SetString("nameFilter", nameFilter);
result = query.List();

```

The second solution may benefit if the database caches the execution plan and reuses it.