nHydrate 5

User's Guide

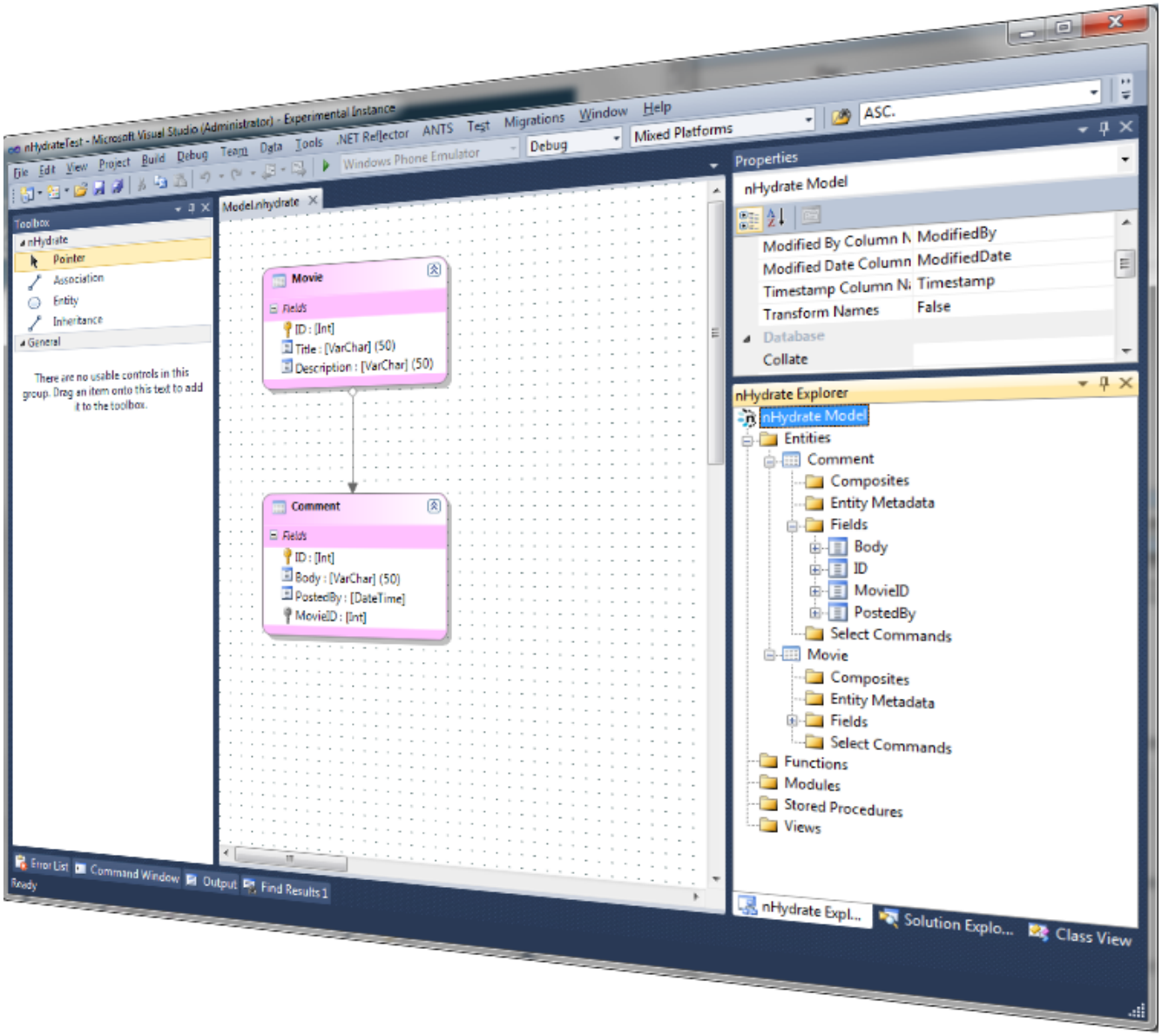


Table of Contents

Introduction

About nHydrate

nHydrate is a domain modelling and object to relational mapping framework for the .NET Framework. Simply put, it allows you to design the business entities around which your system will be formed and handles the retrieval and persistence of those entities allowing you to concentrate on developing the solution at hand.

nHydrate has been designed around the idea of a domain model and the philosophy is centred on the following guiding principles:

- Convention over configuration.
- Support idiomatic .NET domain models: validation, data binding, change notification etc.
- Highly usable API and low barrier to entry.
- Small, lightweight and fast.

Objects and Databases

When you analyse a business domain, you are creating a conceptual model of that domain. You identify the entities in that domain, the state and behaviour of those entities, and their relationships. However, at some point, that conceptual model has to be translated into a concrete software implementation.

In fact, in almost all practical business applications, it has to be translated into (at least) two concrete software implementations: one implementation in terms of programming entities (objects), and one in terms of a relational database. This is where things start getting tedious and potentially complex, because the object and relational worlds use quite different representations. At best, the code to query the database, load objects and save them again is laborious and repetitive.

This is where object-relational mapping comes in. An object-relational mapper, or ORM, takes care of the mechanical details of translating between the worlds of programmatic objects and relational data. The ORM figures out how to load and save objects, using either explicit instructions such as an XML configuration file, or its own heuristics, or a combination of the two. This lets you, the programmer, focus on writing your business logic and application functionality against the domain model (in its object representation), without having to worry about the details of the relational representation.

nHydrate as an object-relational mapper

nHydrate as an object-relational mapper uses a model top define a data system base for organizing objects. The objects map back to specific tables, stored procedures, views, or functions in the database, but you do not need to write any connection code for this. The whole system is Entity Framework based and rides on top of SQL Server. Instead of being all things to all people and

being completely generic, nHydrate strives to be optimized and fast running exclusively on Microsoft technologies.

Key Features

- **True Model Driven Architecture (MDA).** You can manage your whole architectural framework from the modeler. The generated code allows you manage your database and all underlying framework code. You literally can concentrate on the business rules of your application not the framework.
- **Powerful code generation framework.** The generated framework has just about everything you will ever need for your application's API. There is just about no reason to go outside the framework for anything. You can version your database; perform database updates or creations; access a database in numerous ways; save data; and perform aggregate functions.
- **Written with performance in mind.** The generated code allows you to query data with one line of code in most situations. Fields can be optimized for fast database selects. All collections have overloaded static methods that allow complex queries to be constructed in one line. Persistence can also be called in one line. In addition, you can also update multiple database records with one line. Through in as well, aggregate data functions (count, sum, avg, min, max, and distinct) can be called in one line.
- **Numerous Building Blocks.** The framework allows you to model and generate Entities, Typed Lists, Typed Views, and Stored Procedures.
- **Entity Inheritance.** Using inheritance is easy. Simply set the parent table of an entity. It is that simple. If the validation criteria is met, the generator will create a seemly object inheritance hierarchy. When referencing a child object, all parent properties, methods and custom code are available. This is because inherited objects are really derived in code from the base objects.
- **Support for all relation types.** You can model and generate 1:1, 1:N, and N:M relations in the model. There is no need to write any custom for select by defined database relationships.
- **Dependency Walking.** There is no need to write any custom code to walk relationships. Each entity with a 1:N relation will reference the foreign table with a list of objects that will be retrieved automatically when requested. This is completely transparent to you. In addition, each 1:1 relation will have a single related foreign object. Finally each N:N relation will have list of foreign objects in both directions. The retrievals are done in the background or can be cached to load all at once.
- **Full support for complex database constructions.** Full support for primary keys, foreign key constraints, unique constraints, defaults, identity fields, GUID fields, sequences, code facades, views, stored procedures, relations with self, multi-field primary keys, multi-field foreign keys, multi-field unique constraints, objectified m:n relations, and much more.
- **Support for modified relational schemas.** Version a database and keep track of your model changes between deployments. You can create an upgrade track between versions allows you to upgrade any lower version to a higher version. The database installer is generated from your model and contains all information to upgrade a database or create it from scratch. It can run in Microsoft InstallUtil tool or plug it directly into a larger installation application.

- **Typed Lists.** You can define typed entity that generates an enumeration to be used in code. This allows you to write code that is easy to read instead of using “magic numbers” for properties.
- **Graph-aware Save Logic.** You can load and manipulate data in any order. There is no need to write any custom code to define save logic. All subdomains have a Persist method that knows the order in which to save data to your database. You can load, add, delete, and modify data in any way. You can even dependency walk from any loaded collection to any other object or collection (loaded or not loaded) to get new data at any time. Persistence is automatic and transparent.
- **Persistence of 'dirty' (changed) data only.** Only 'dirty' entities are persisted to the database. The entire parent-child hierarchy is saved but only the modified and added entities are persisted to store.
- **All objects are disconnected from the database.** All objects are disconnected from the database and from every other object. You can use the entity objects, typed list objects and typed view objects without having a connection to the database; no database connection is kept open after you have fetched an entity. All loading and persistence is atomic.
- **Server-side paging.** Full paging support is available for collections. Paging is performed on the database server, so only necessary data is brought to the client and loaded into memory.
- **In-memory filtering and sorting using LINQ.** Use the new LINQ syntax to sort or filter data in memory.
- **Auditing Support.** Two types of auditing are supported. The first is built-in fields for tables that allows you to track the person that added or modified a record and the date. The second is a shadow table that stores all add, edit, and delete operations on a table. This is a history table that can be queried to determine any state of a record in a table.
- **Validation support.** There is a validation framework, which makes it very easy to add validation code to the generated classes and intercept actions on the entities. Each property has events that are raised during and after modifications. There are also hooks in to the Persistence mechanisms. The framework also supports client-side null validation and length validation before you save changes.
- **Singularization and Pluralization of Entity Names.** Singularization of entities and pluralization of lists is built-in.
- **Exclude fields from entity fetches.** You can define components on tables which are subsets of tables. You can manipulate these entities just like tables, even persisting them. This is quiet useful for legacy systems or large tables with blob or image fields.
- **Unit of Work and multi-versioning of entities or sets.** You inheritably define a unit of by using the framework. All CRUD actions are built-up and saved at one time in one transaction. This is a unit of work and all action will succeed or fail as an atomic transaction.
- **Advanced lazy loading/load on demand.** Walk any relationship in the model with no custom code at all. This is built-in and you get it out of the box. Simply call a related object or collection of an entity and if it is not already in the subdomain it will be loaded automatically. This is completely transparent to you.
- **Advanced prefetch functionalty of related entities.** Lazy loading is the easiest way to get data, but you can also define the entities to be loaded and do so in a transaction. There are

many select commands that are generated based on keys and relationships. Simply add the desired order of load to a list of select commands and load them all at one time.

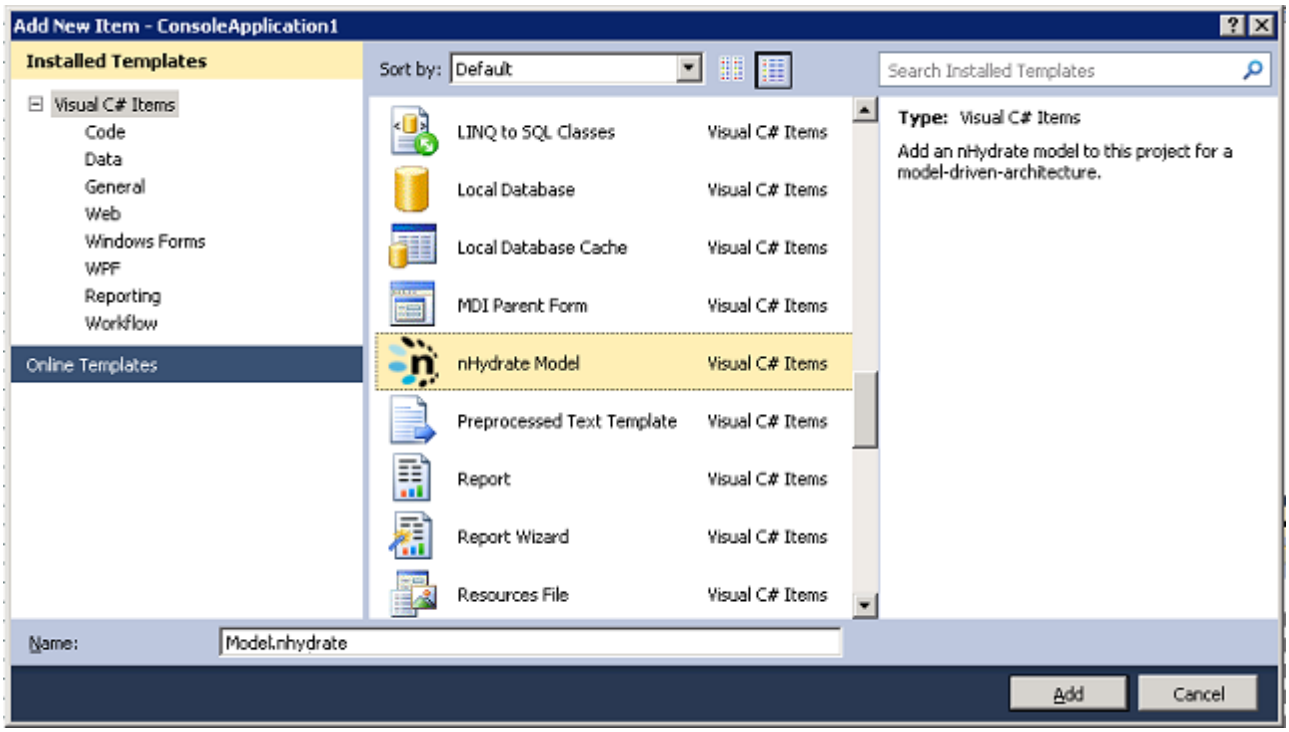
- **Load objects with LINQ syntax.** You can load any collection with a strongly-typed LINQ syntax. If your model changes, you will get a compile time error. You will not ship a product to break in the field anymore. There is never a reason to retrieve an entity property by string value like a dataset.
- **Patterns based generated code.** All generated code uses various well-known patterns to form a working layer and to deliver its functionality. Patterns used are the Observer, Active Record, Repository, Visitor, Composite, and Interpreter and Data Transfer patterns.
- **Code added to the generated classes is preserved.** All classes are generated as partial classes with a gen-once file and a gen-always file. You can add any code you to the gen-once file and it is never overwritten. This code is merged with the generated class file to create one class. There is no need to write code only in specified regions like most code generators. You can write any free form code you wish.
- **Partial classes.** Generated code is built with partial classes, which makes it easy to extend the generated code, through the partial class mechanism in .NET 2.0+.
- **Generics and nullable types supported.** If a field is defined as nullable in the database, it is nullable in code as well.
- **Fully object oriented, typed query mechanism.**
    1. Table joins seen as object relations and used to create object and list references on an entity
    2. Compile time checked filter construction, with easy to formulate constructs like (Customer.Order.Product.Name == "Widget")
    3. Support for relations (1:1, 1:n, n:m)
    4. Predicate specification with static entity collection methods.
    5. 100% typed dynamic filter construction using Predicates
    6. Support for sorting, filtering and resultset limitations (number of rows returned)
    7. Support for strongly-typed aggregate querying, i.e. when you aggregate an integer value you get back an integer value. You are never returned an “object” type.
    8. Join tables on multiple relationships by giving each a role name. The specified name is in the generated code and is used to distinguish which relationship is being walked.
    9. Fully integrated inheritance. A derived object has all of its properties and those of its parent.

- There is no need for a developer to know the entire inheritance hierarchy. You can work with an object as if it were stand-alone. There is no need to know which property comes from which entity in the inheritance tree.
- **Support for aggregate functionality.** Call aggregates with one line of code, even with complex logic. For example you can call the count aggregate like so “CustomerCollection.GetCount(x=>x.CustomerId < 100)”. This is strongly-typed and any change in the reference fields will be validated by the compiler.

- **Concurrency Control Mechanism.** Concurrency is built-in with a timestamp field. If a database row is changed by another user while in memory, the concurrency will fail and an exception will be raised. You will never get the “lost update problem” again.
- **Dynamic SQL Support.** If there is some custom action or query that you need that is not provided by the framework, you can include it in the framework with custom view and custom stored procedures. This allows strongly-typed objects to be generated based on your custom specification and parameters.
- **Visual Studio Integration.** After looking at many other code generators what is your biggest gripe. Chances are it is no VS.NET integration. With complete integration into the .NET environment, your classes are generated right into the Project Explorer. There is no need to manually include files in your project. After your generation is complete you can compile, that simple.
- **Start Working Immediately.** There is no setup time for days and weeks. Simple reverse engineer your database into a model (or create one from scratch). Setup your metadata, naming schema, etc and generate. You can start building code immediately. There are no cumbersome XML configuration files to write by hand!

Creating Models with the Visual Designer

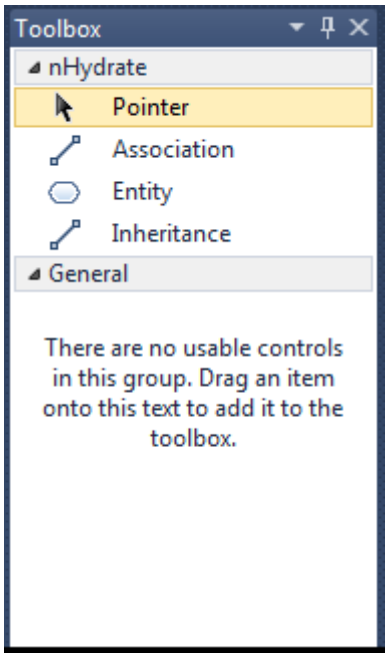
The nHydrate visual designer integrates into Visual Studio 2010. To create a model with the visual designer, add a nHydrate Model item to your Visual Studio project. The easiest way to do this is to right-click the project in Solution Explorer and choose Add > New Item, then choose nHydrate Model in the Add Item dialog.



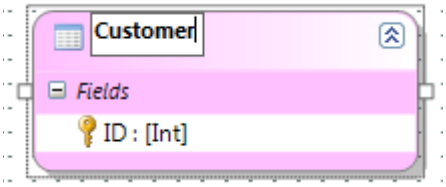
The designer is initially blank . You can either create entities in the designer using the Toolbox, or create
entities from your existing database tables.

Creating Entities in the Designer

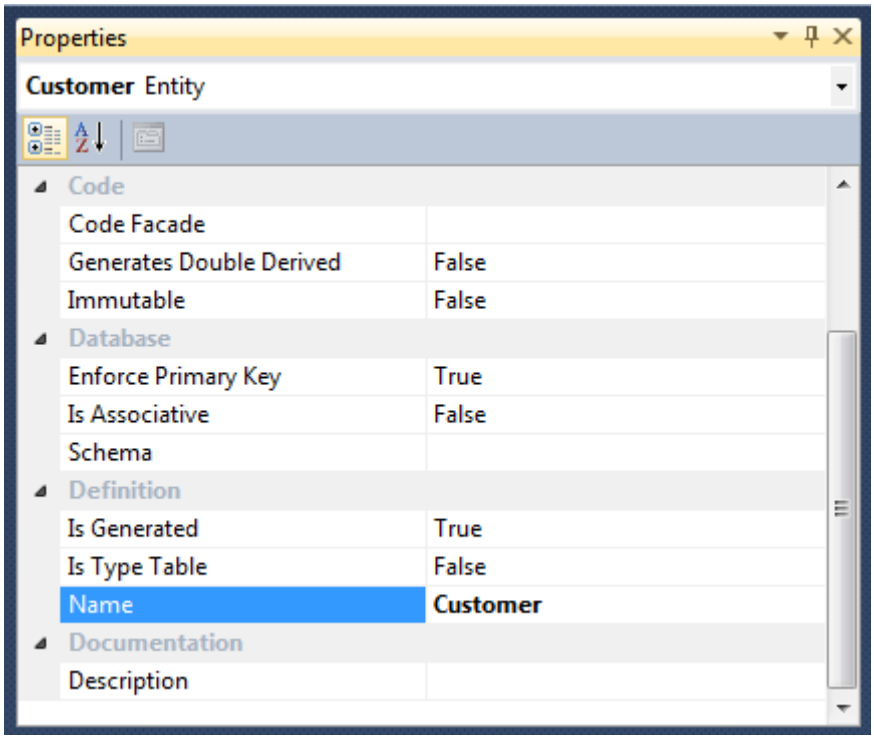
To create an entity in the designer, open the Visual Studio Toolbox and select the nHydrate tab and drag an Entity object onto the design surface.



Initially, the new entity will be called Entity1. To change this, just type the new name while the entity is selected.

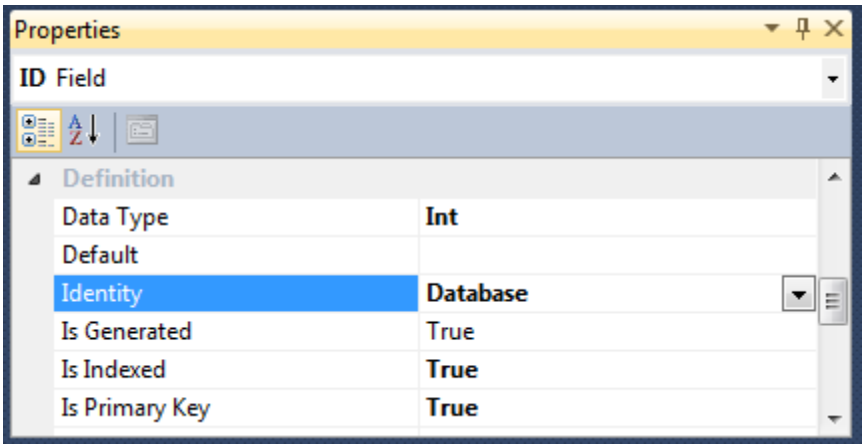


The Visual Studio Properties window shows more options for configuring the entity. For example, you can also edit the entity name through the Name box in the Properties window.



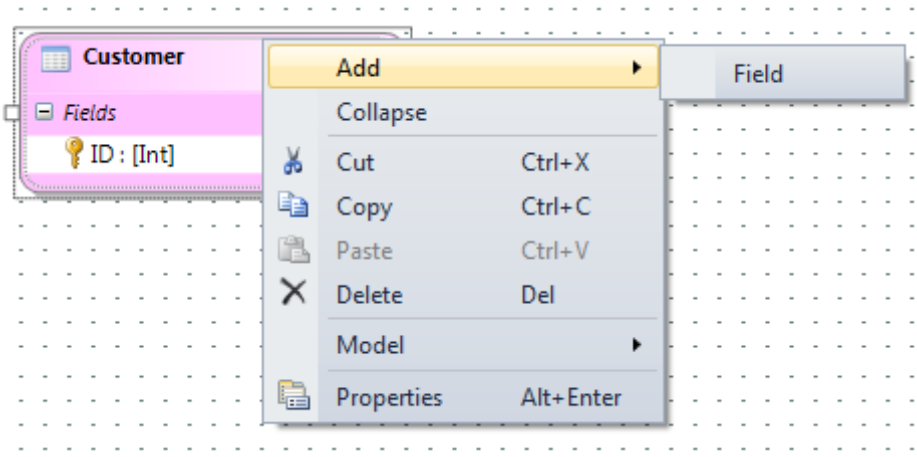
Many of the settings in the Properties window are things you'll only need to think about as you start building up your model. For example, you'll use the Auditing options if you need to add the built-in auditing to the entity. This book covers the various options under the relevant chapters. You can also get an idea of what each option does by looking at the description area at the bottom of the Properties window.

One option that's important for all models is the Identity option. Every entity in nHydrate must have a primary key, a unique identifier that allows nHydrate to tell it apart from other entities of the same type. The default identity type is Int32 – the .NET Int32 type, equivalent to the C# int type. If you expect to create a huge number of entities, you'll probably want to change this to Int64 (C# long). Some users prefer GUID Ids to numeric Ids, so you can also choose the Guid identity type.

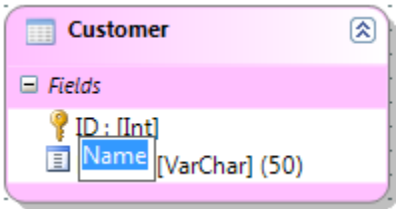


Notice in that the ID field is marked as a primary key which means it is non-nullable. It is an integer and a database identity. This will cause the database field to have these attributes as well the code. The code will have a read-only property as its unique identifier, since the values are managed by the database.

To add properties to the entity, right-click it and choose Add Field.

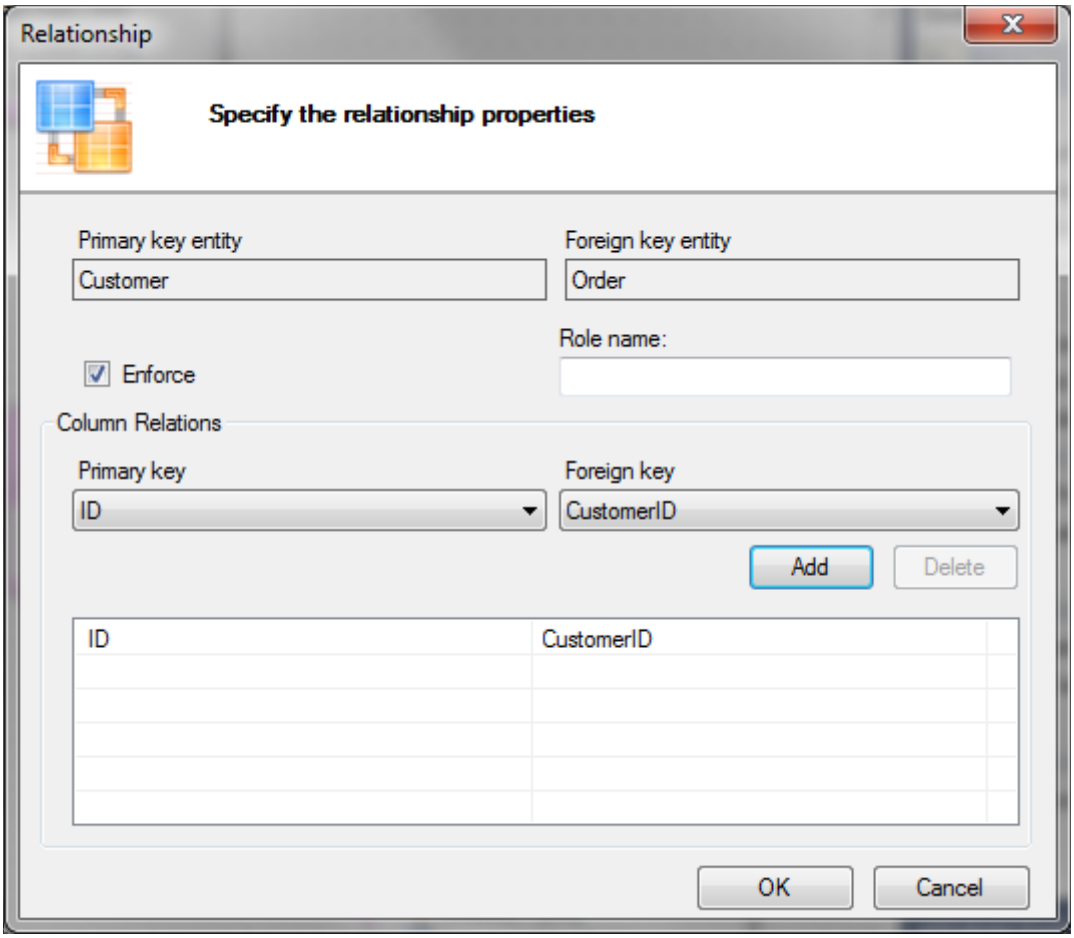


You can edit the name of the new property by typing directly into the entity shape.

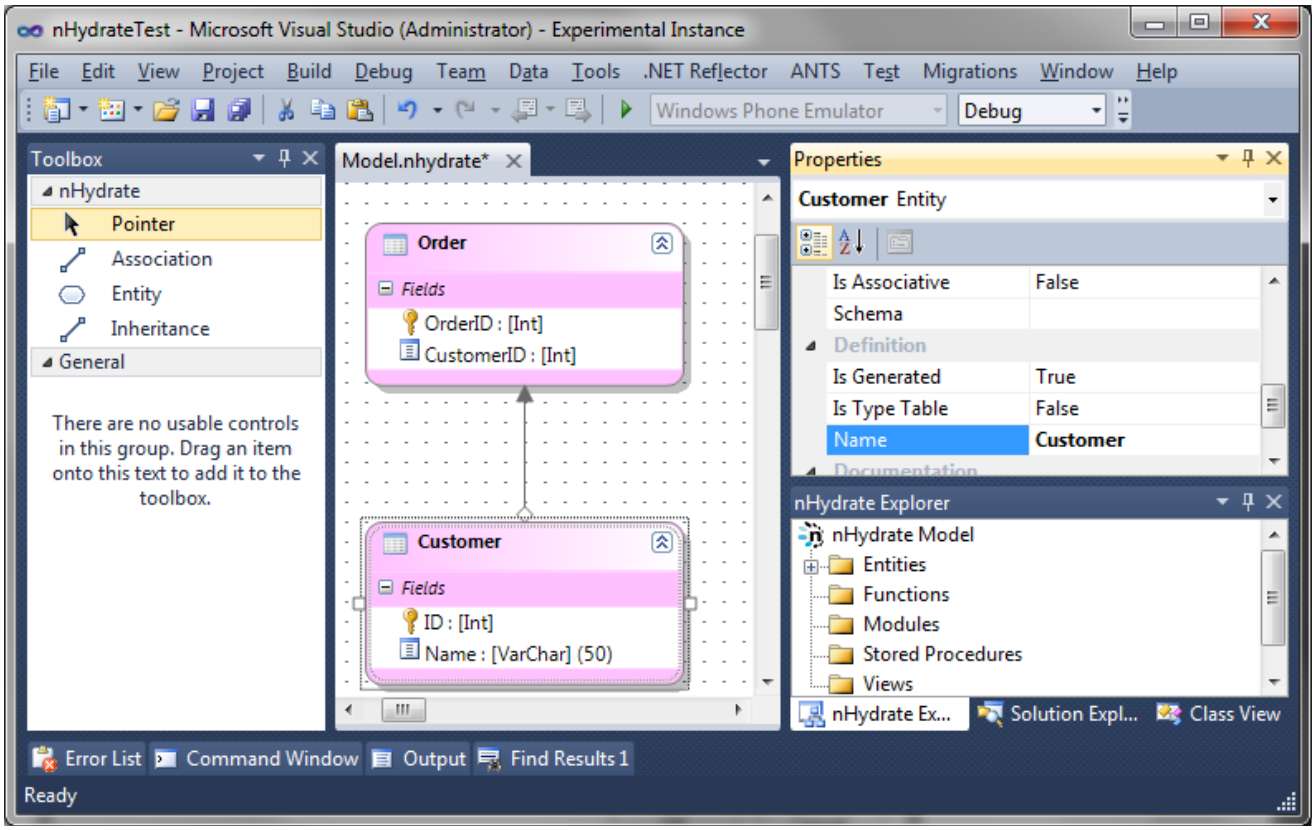


You can now enter in the entities that define you model. After you have two or more entities you can start drawing associations. I have created an additional Order table and want to relate the Customer

entity to the Order entity via a shared CustomerID field. I choose the Association item from the toolbox and click on the Customer and secondly on the Order entity to create the association. A dialog pops-up that to allow me to configure the relation. This is only an example of a relation and they will be handled later in more detail.



Once you have created a model the canvas will look similar to the following graphic.

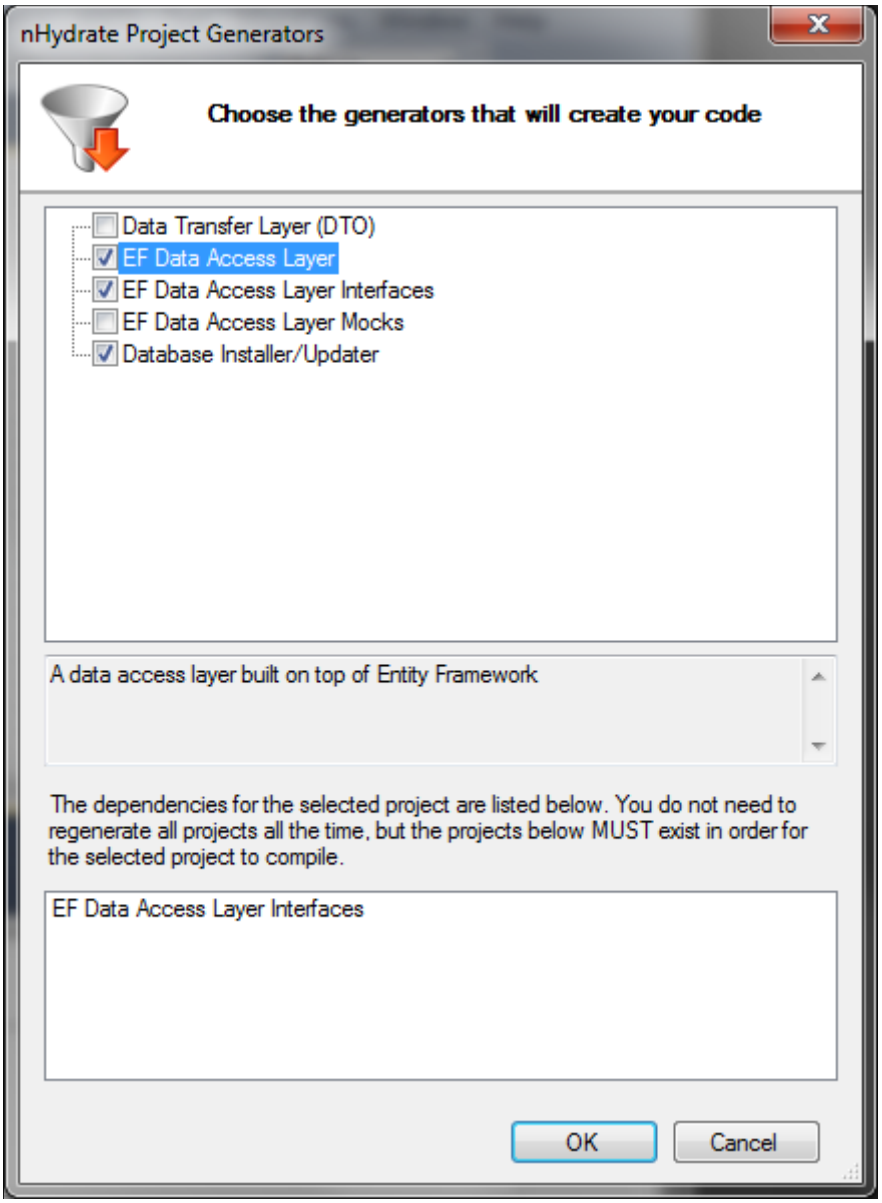


You can now generate your model and create the associated code.

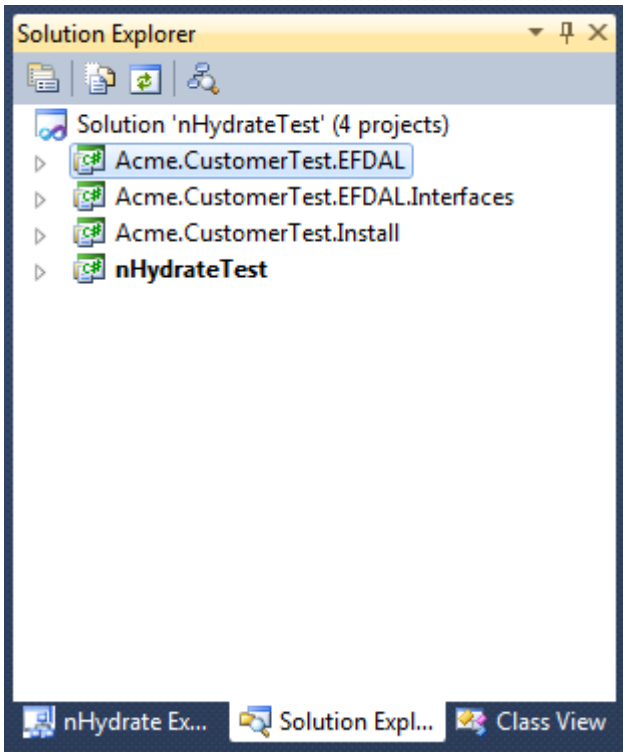
Creating Domain Models

Now that we have a working domain with two entities and a relationship, we can generate the code to create an domain model or API. This is based on Entity Framework with the added benefit of using interfaces.

We right click on the canvas background and select the Model|Generate menu. This will display a dialog that allows us to choose the generators we wish to use.



We will choose the Entity Framework data access layer (EFDAL) and the EF Interfaces project, as well as the Database Installer project. After you generation is complete there will be three new projects in your solution. The solution explorer will look like the following image.



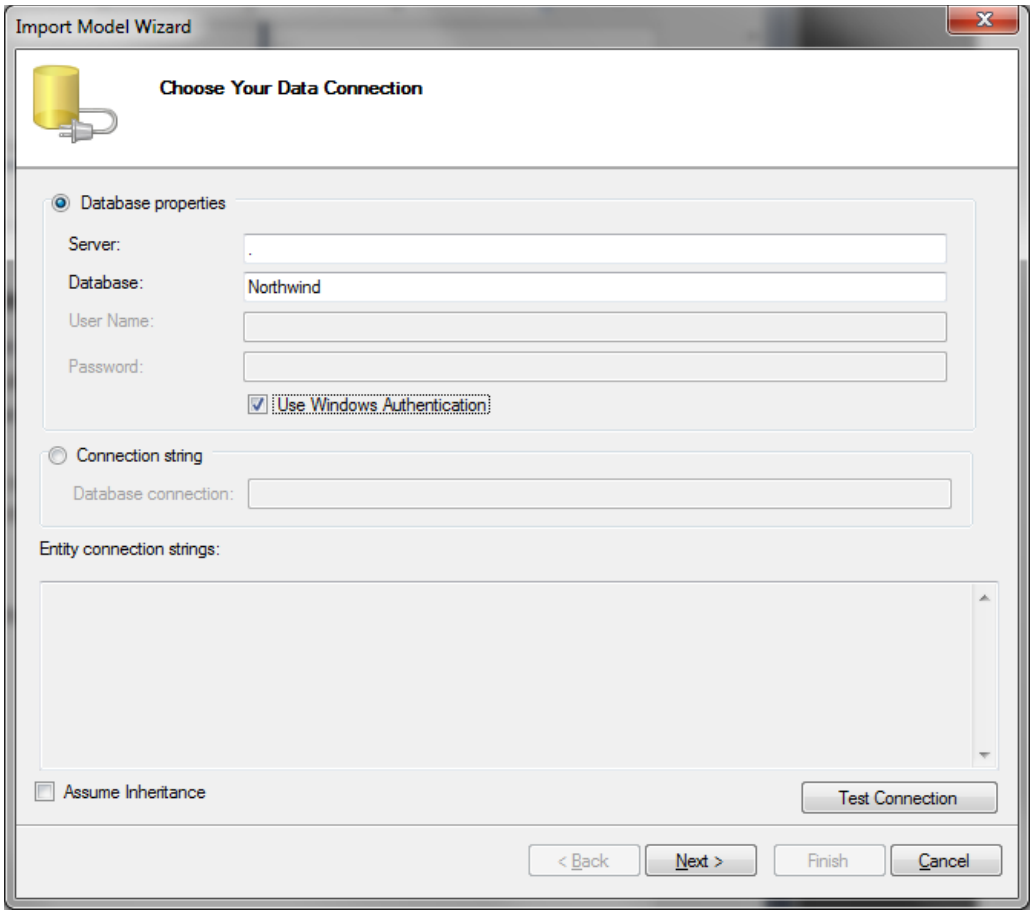
Now we have a ready to go domain layer based on Entity Framework.

Creating Database Table from Entities

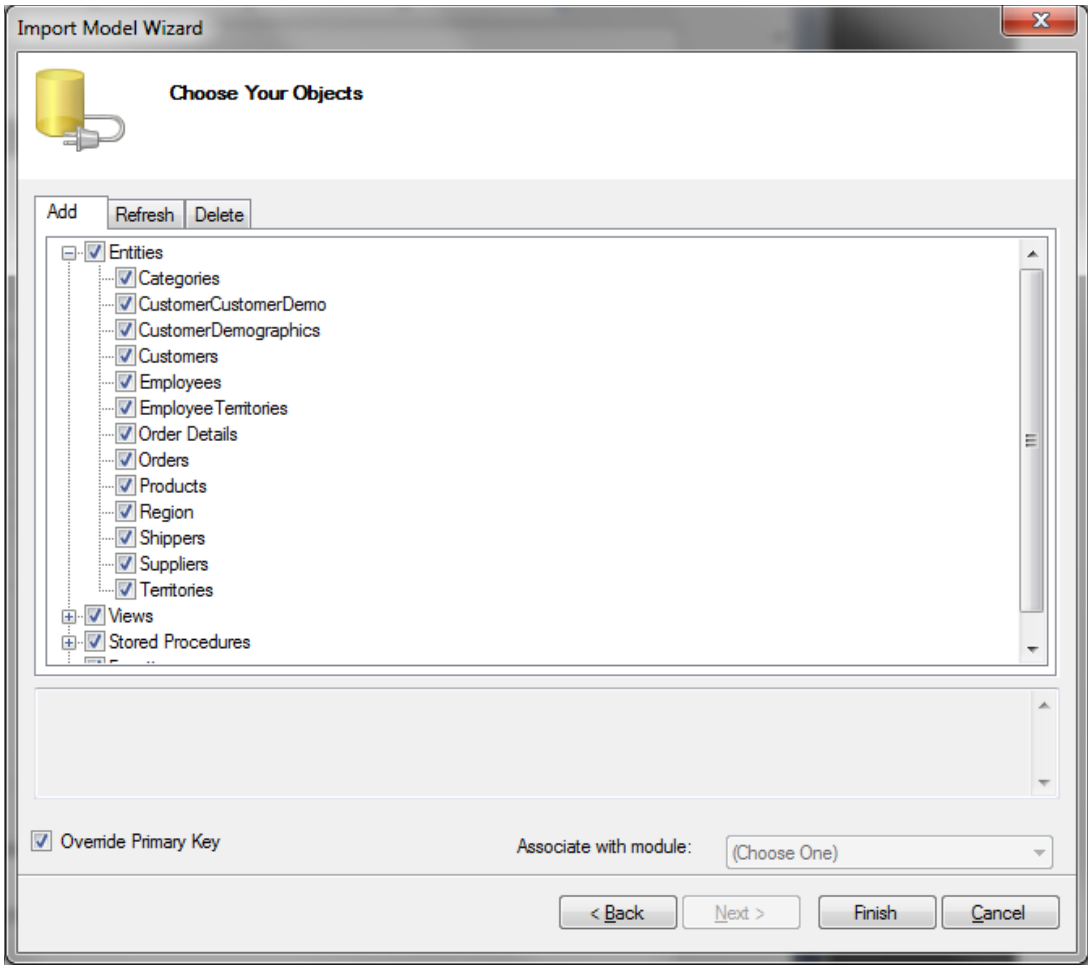
As discussed above, a domain model is realised in at least two ways: an object model and a relational database schema. The object model is automatically generated when you save the visual model file. You can also have nHydrate generate the database schema for you.

To do this, simply use the Import screen to define your SQL Server connection criteria and choose the objects you wish to import. This is very much like the Entity Framework import dialog with the addition that it will import Functions as well.

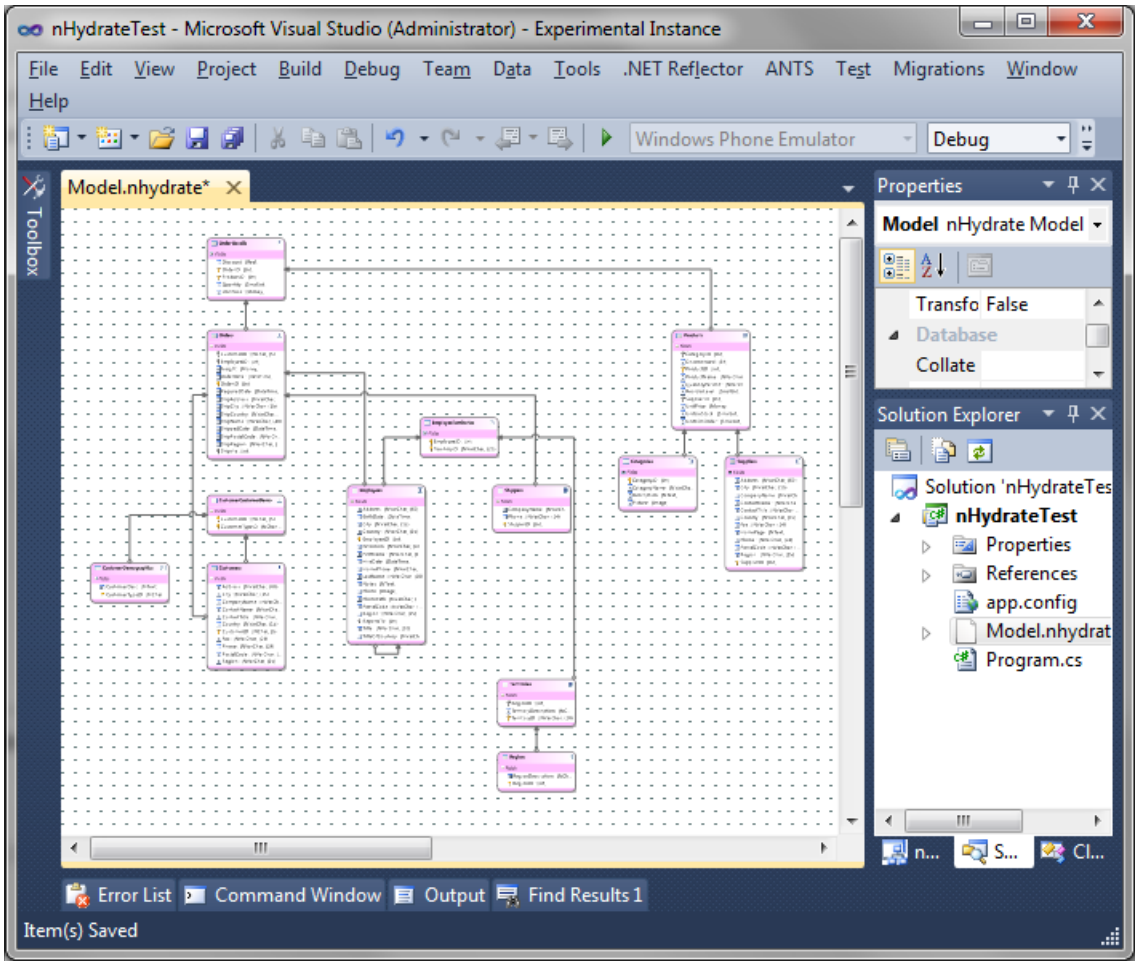
Right click on the background canvas and select the Model|Import Database menu. The following dialog will be displayed.



I have chosen to import the Northwind database on my local server. When you press the Next button a list of objects to import will be displayed.



Choose the objects to import. By default all objects are selected. Once imported you will see many tables and relations now on the model.

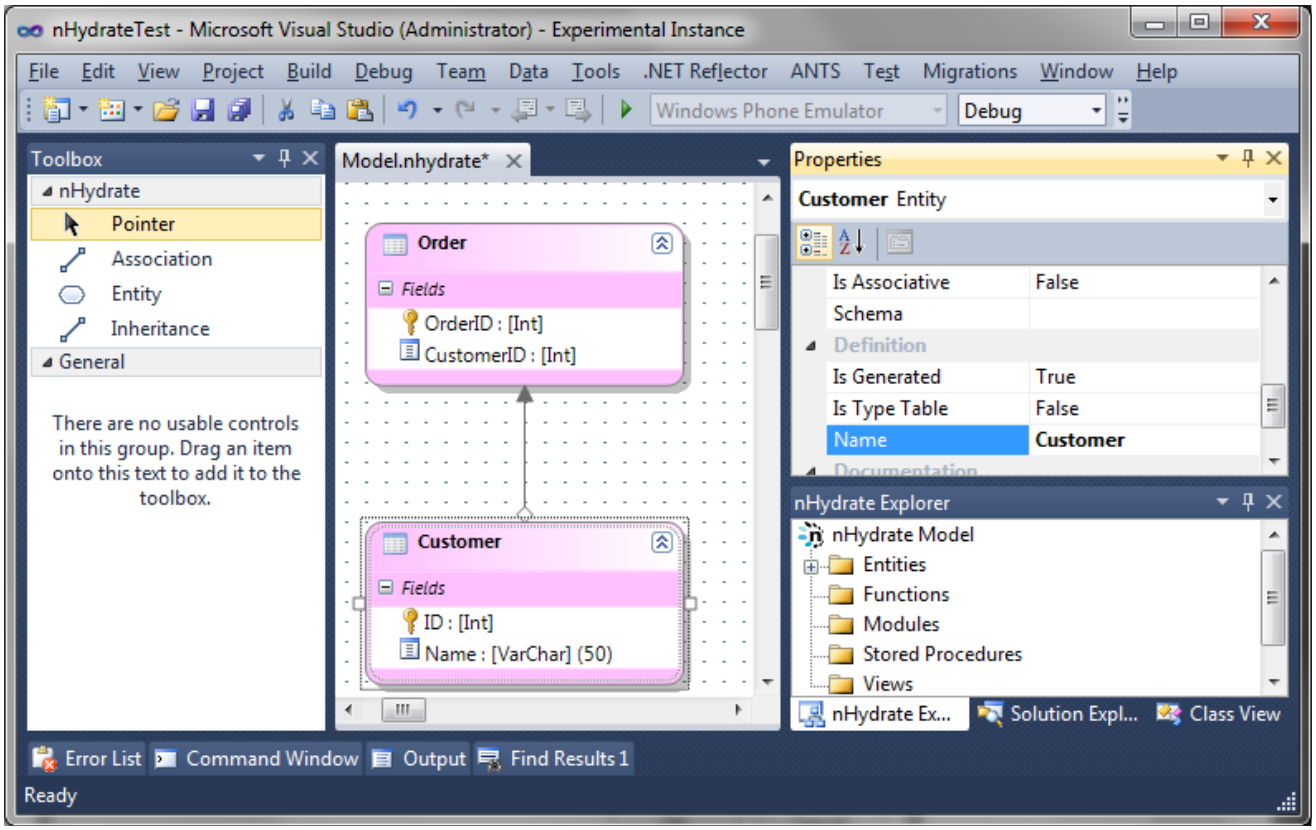


This is a full import from a database. This is a good way to get started working off an existing or legacy database.

Creating Associations Between Entities

nHydrate supports three kinds of associations: one-to-many, one-to-one and many-to-many.

To create a one-to-many association, select the Association connector in the toolbox, and drag an arrow from the ‘one’ end to the ‘many’ end. For example, if a Customer can have multiple Orders, drag the arrow from Customer to Order.



This creates a property at each end of the association. The ‘one’ end has a collection property, representing the collection of associated ‘child’ entities. The ‘many’ end has a backreference property, representing the ‘parent’ entity. In this example, Customer has an OrderList collection property, and Order has a Customer backreference property. nHydrate uses this naming convention so there is no ambiguity or misnamed properties based on English irregular words.

A one-to-one association is added in much the same way as a one-to-many association, except that it has a source and a target instead of a collection and a backreference. There is no need to explicitly define a one-to-one relation. Relations always start from the primary key of the source object. They terminate on a field of the target object with the same data type. If target field (or fields) are unique in the target table, the relation is one-to-one. This can mean one of two things. The target field(s) are the primary key of the entity or it has been marked as unique in the model. Either one will create a one-to-one relation.

When you use Update Database to create or update database tables, the Installer project creates a foreign key column in the appropriate table to represent each association.

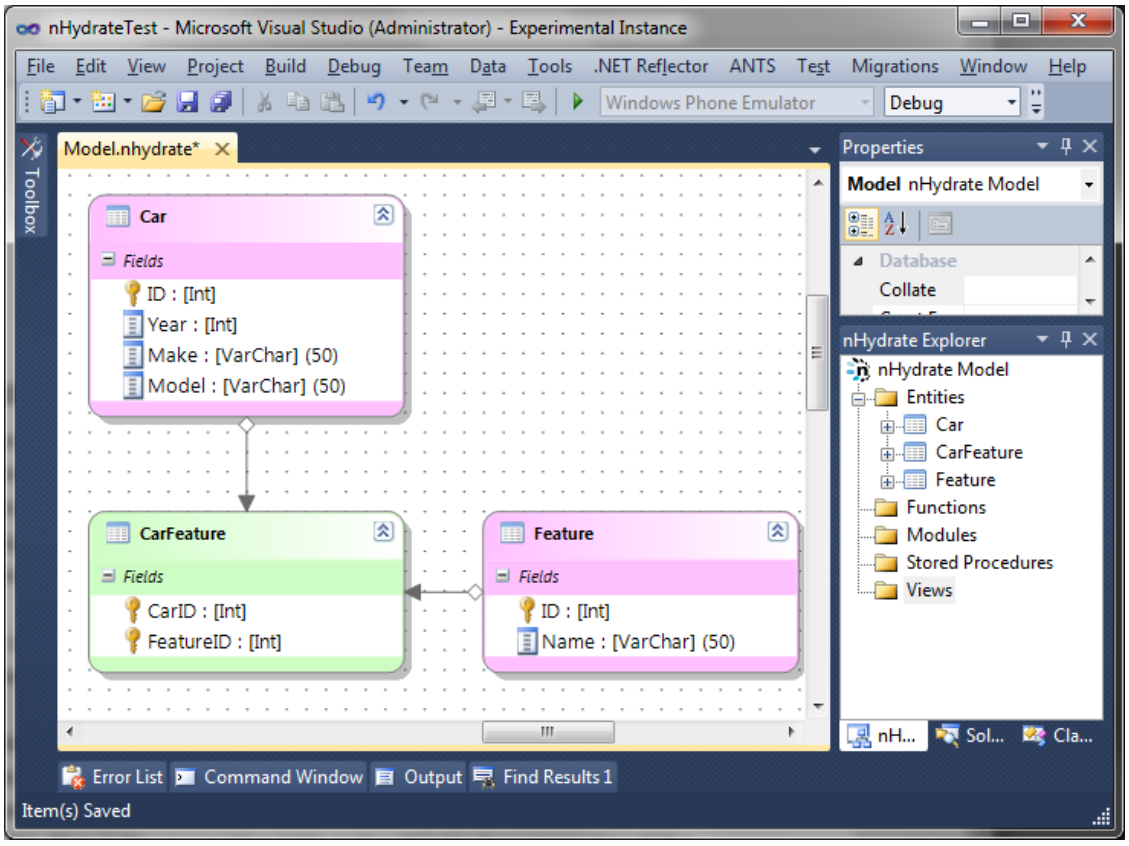
If you are creating entities from database tables, then the Installer project ensure that the associations are created and enforced when you upgrade a database.

Many-to-Many Associations

Many-to-many associations work in a slightly different way to one-to-many or one-to-one associations, because they have to be stored in a different way at the relational level. Instead of a simple foreign key, a many-to-many association requires a whole table of foreign keys. This table is variously known as a join table, relationship table, through table, or associative table. Each entry in the through table represents a pair of associated entities; and an entity can participate in multiple pairs.

A many-to-many association is modelled in nHydrate using an associative table. The intermediate entity corresponds to the associative table in the database.

To create an association of this type, you must create an entity with the primary keys of the two entities you wish to join. Then just like any other relation draw an association from the first entity and the associative table and then from the second entity and the associative table. Each time join on the primary keys of the source table. Now in the associative table make all fields the primary key of the table. Lastly, set the table property IsAssociative to true.



The code equivalent of this relation will be that a Car entity will have a FeatureList property and a Feature entity will have a CarList property. The CarFeature entity will not exist in the API. You will only work with the two primary objects.

Indexes

Database indexes are also imported and managed. There are three types of indexes in the model. When you define a primary key, the first type is created. You cannot remove or modify this index type. It is managed by the model and based on the columns marked as primary key.

The second type of key is a defined by the IsIndexed property of a field. If you wish to create an ascending index on a single field, simple set this property to true and it will be created by the installer and managed by the model.

The last type of index is user-defined. This is any custom index that you create. In the nHydrate model tree, each Entity node on the tree has an Index collection. You right click on the Entity in the tree to add a new index to it. You can define as many columns as you wish and configure them to be in ascending or descending order.

All indexes will be created and managed by the installer project. The installer defines the names and creates the scripts that will ensure that the indexes exist in the database.

XML Documentation

Your model can include documentation for entities, properties, one-to-many associations and stored procedures. Documentation will be emitted as XML documentation comments which are displayed in Intellisense or can be built into a Help file using a tool such as Sandcastle. Each object has a description property that will be used to create this documentation.

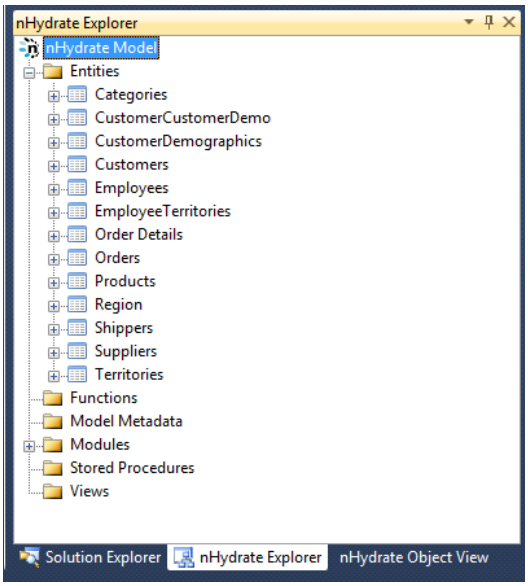
Tool Windows

There are three tool windows by which you can interact with the model. These are the nHydrate Explorer, Object View, and Documentation windows.

nHydrate Explorer

The primary window is the nHydrate Explorer. This shows the model a a tree diagram. You can add and delete objects from this tree. All model objects including modules and metadata are present and can be manipulated in this window. When an object is selected on the diagram canvas its representative tree node is automatically selected, as well as its properties displayed in the properties window.

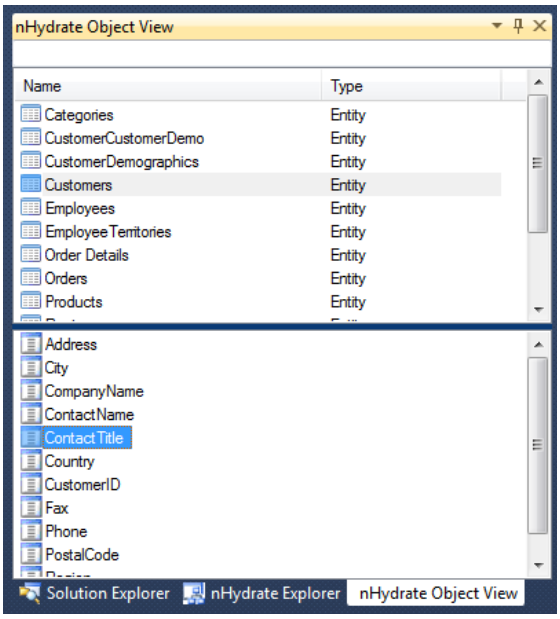
nHydrate Explorer Window



Object View Window

The Object View window can be used to access model objects. It is more functional than the Explorer in that it has many context menu items. Its column display can be customized selecting the “Select Columns” context menu. Object can be renamed in place by pressing F2 or single clicking an item twice. All primary model objects (Entity, View, Stored Procedure, and Functions) are displayed in the top portion of the window. When a primary object is selected, its constituent objects (fields and parameters) are displayed in the lower portion of the window. Context menus all you to setup relationships, view indexes, setup static data, and refresh from database as well as other functionality. When any object is double clicked in this window, its associated shape is highlighted on the diagram as well as its tree node representation highlighted in the nHydrate Explorer window.

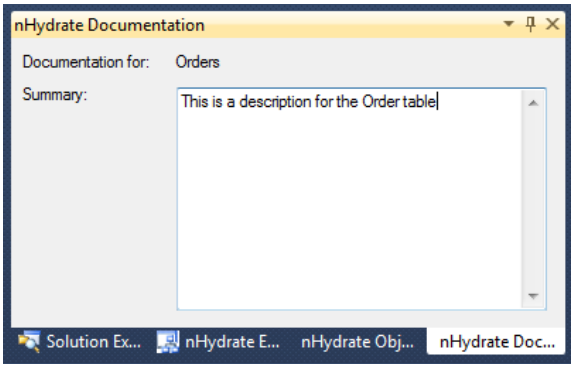
nHydrate Object View Window



Documentation Window

The documentation window simply allows you to have a larger canvas for editing documentation of model objects. When any object on the diagram is highlighted that can be documented like Entities or Fields, this window allows you to edit the associated object's documentation instead of using the Properties window.

nHydrate Documentation WIndow



Basic Operations

The core operations of nHydrate are the familiar CRUD (Create, Read, Update, Delete) database operations. The CRUD model lies behind the majority of database-backed applications and Web sites. This chapter shows you how to implement CRUD using nHydrate.

Working with Entities

All persistent data is represented in nHydrate as entities. An entity represents a business object with identity. It can be as big as a company or as small as a single order line. Of course, an entity can have associations to other entities, allowing an aggregate to represent a rich business domain. An entity can be loaded, modified and saved, and retains its identity throughout its lifecycle.

Connection Strings

Before accessing the database you will need to setup a connection string. Using Entity Framework requires you to use the new Entity Framework connection string. This is not as easy to remember as a traditional connection string. An EF connection string looks like this when using a generated assembly. This assumes you have a model with company name “Acme” and a project name “MyProject”.

Entity Framework Connection String

```
metadata=res://*/Acme.MyProject.EFDAL.MyProject.csdl|res://*/
Acme.MyProject.EFDAL.MyProject.ssdl|res://*/
Acme.MyProject.EFDAL.MyProject.msl;provider=System.Data.SqlClient;provider
connection string='Data Source=.;Initial Catalog=MyDatabase;Integrated
Security=SSPI;'
```

This is not at all easy to remember or write. You can see the real connection string embedded in the EF qualifiers. It would be nice to use just a normal connection string. Now you can with the generated assembly. This is not possible with traditional EF; however some convienence functionality has been generated into the object context as emitted with nHydrate. Now you can use a connection string like this.

Normal Connection String

```
Data Source=.;Initial Catalog=MyDatabase;Integrated Security=SSPI
```

You can use this overloaded connection string functionality from the App/Web config file or the object context’s constructor.

Querying the Database with LINQ

The generated Entity Framework API declares a strong-typed context class that exposes properties representing queries for different types of entity. To query the database using LINQ, we need to create a context of this type, associated with our specified configuration settings. We can then issue queries

against it using the normal LINQ syntax, just like any other Entity Framework API.

Common LINQ Techniques

To filter a query – that is, to specify which entities you want to return – use the where keyword or the Where extension method.

To sort a query, use the orderby keyword or the OrderBy extension method. Sorting is in ascending order by default: the orderby keyword allows you to specify the descending modifier, which corresponds to the OrderByDescending method. The orderby keyword supports sorting on multiple attributes; additional attributes correspond to the ThenBy or ThenByDescending method.

Selecting Orders by parent CustomerId

```
var recentOrders = from o in context.Orders
        where o.CustomerId == customerId
        orderby o.OrderDate descending
        select o;
```

To perform paging of a query, use the Skip and Take extension methods. If you don't also specify an order, either explicitly in the LINQ query or implicitly on the entity class, Skip and Take order entities by Id. You can combine Skip and Take if you want to page through a result set.

```
var ordersToDisplay = context.Orders
        .OrderBy(o => o.OrderDate)
        .Skip(pageStart)
        .Take(pageCount);
```

To work with the entities returned from a LINQ query, use the foreach keyword to iterate over the query, or use the ToList extension method to load the results into a list.

If you are only interested in a single entity, apply the First or Single extension method to obtain it. First returns the first matching entity, ignoring any others; Single checks that there is only one matching entity.

```
List<Order> allOrders = context.Orders.ToList();
        Order order = unitOfWork.Orders.Single(o => o.Id == orderId);
```

If you want to know how many entities fit the query criteria, apply the Count extension method. If you want to know if any entities fit the query criteria, apply the Any extension method.

```
int pendingOrderCount = context.Orders
        .Where(o => o.StatusId == (int)OrderStatus.Pending)
        .Count();
```

To perform a projection – that is, to select only a subset of the entity fields – use the select keyword or the Select extension method. If you perform a projection, then you will typically project into a non-entity type, and the data will not be associated with the context. This is therefore used for presenting partial, read-only information about an entity.

```
var orderSummaries = from o in context.Orders
        select new { OrderId = o.Id, o.OrderReference };
```

All of these methods are translated to server-side SQL statements so time and bandwidth is not wasted pulling back unwanted rows or columns. For example, if you specify Take(5) then Entity Framework will limit the number of rows returned to 5; if you specify Count() then Entity Framework issues a SQL COUNT query rather than materialising entities on the client.

LINQ Expressions

LINQ allows you to write queries of arbitrary complexity. Entity Framework handles only queries that can be translated to SQL on the database at hand. Consequently, if you write complex queries, you may encounter not supported error at runtime. This indicates that Entity Framework was not able to translate the LINQ query to SQL. Consider simplifying the query, and performing further operations on the client. You can use the ToList() and AsEnumerable() operators to partition work between the database and the client.

Creating, Modifying and Deleting Entities

The previous sections show how to query the database using nHydrate. In many applications you will also want to save changes to the database – adding new entities, modifying or deleting existing ones. As with querying, nHydrate supports these operations through the Entity Framework context.

How Changes are Saved

nHydrate saves changes to a context, not an individual entity. When you add, modify or delete an entity, it is not saved immediately. Instead, the context just notes that the entity needs to be saved. When you call the context SaveChanges method, all the entities that need it are saved. This means that you can coordinate the persistence of multiple related changes, and minimises the number of database round-trips.

Adding a New Entity

To add a new entity use the Entity Framework context. You can freely create entity objects as they have public constructors so there is no need to use a factory. Simply create an entity and add it to the context

Creating a new entity and adding it to a context

```
var customer = new Customer();
customer.Name = "John Doe";
context.AddItem(customer);
```

If a new entity is associated with another entity that is already part of a context, it automatically becomes part of the same context. This saves you having to remember to add the entity to the context separately. Any kind of association will trigger this.

Creating a new entity which implicitly becomes part of the context

```
var order = new Order();
order.OrderDate = DateTime.Now;
customer.OrderList.Add(order);
```

Note that because nHydrate saves contexts, you must add the entity to a context– whether explicitly or implicitly – in order for it to be saved. Just creating the entity is not enough! If an entity has a database identity or generated key, it will be populated with the actual database value after save Before a new entity is saved, its field is invalid and should not be used.

Updating an Existing Entity

To update an existing entity, load it from a context and set any required properties to their new values.

Updating an existing entity

```
var customer = context.Customer.First();
customer.Name = "John";
```

Entity Framework automatically determines that the entity has changed, and marks it to be saved.

Bulk Updates

In addition to modifying entities and persisting them to the database, you can update items in bulk. One of the performance issues with standard Entity Framework is that if you wish to modify multiple database rows, you must load their corresponding entities into memory to change them and then persist the container context. However this is not a workable solution for massive amounts of data.

The nHydrate framework adds some additional bulk operations. Each entity object type has a static method named *UpdateData* that can be used to update multiple records at one time in one transaction.

Update multiple database rows base on a LINQ statement

```
Customer.UpdateData(x => x.SomeDate, x => x.Name.Contains("John"),
DateTime.Now);
```

This code snippet will update the *SomeDate* field in the Customer table where the name field contains the string “John”. The specified value is strongly typed. The *SomeDate* field is a date so the specified value must be of type date as well or a compile-time error occurs. This ensures you do not have typing issue at runtime.

The caveat is that only one field can be updated at a time. There is no way using this method to update multiple fields in multiple rows in one call or transaction.

Deleting an Existing Entity

There are a few ways to remove entities. The first is the standard way EF handles deleting. First load the entity from a context and remove it from the context.

Deleting an existing entity

```
var customer = context.Customer.First();
context.DeleteItem(customer);
```

However this does require two database hits. The first to load the item and the second to remove the item. This is fine if you have the item in memory already. However many times you will load the enitty only to remove it in the next line of code.

Bulk Deletes

There is a more efficient way to remove data built-into the custom functionality provided by nHydrate. Each entity type has a static method that allows you to issue remove commands to the database without actually loading the data first. This makes remove large sets of data much faster and less memory intensive.

Deleting entities without loading them first

```
Customer.DeleteData(x=> x.Name == "John");
```

Using this syntax, we can simply issue a LINQ statement that will translate into a delete SQL command in the background.

When deleting data it is always importing to think about database integrity. You cannot remove a Custom object that has child Order objects. You must remove the children first and then the parent. These actions can of course be done in a batch by removing the objects from the context as in the first example and then saving the context.

Saving the Context

When you have made all the changes you need to make, call the context's SaveChanges method. SaveChanges validates all entities that are due to be added or updated, and will not save an invalid entity. (Invalid entities may be deleted.)

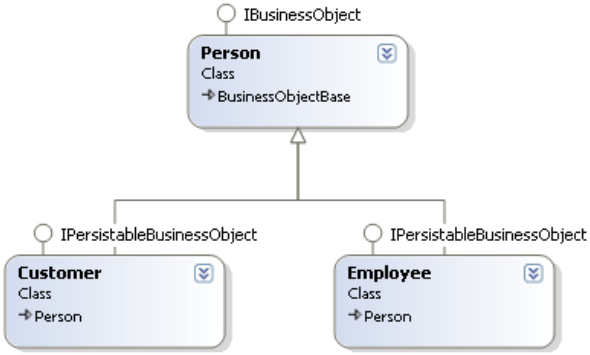
Transactions

SaveChanges is automatically transactional: that is, if more than one entity needs to be saved, Entity Framework guarantees that the changes to the database will be atomic and durable (provided the database supports transactions). To achieve this, SaveChanges automatically begins a transaction before sending the first change, and commits it after sending the last change if all changes have been successful. As far as Entity Framework is concerned, however, each SaveChanges is an independent transaction.

Documentation

All model objects with a *Description* property support API documentation. This field on Entity, Field, Stored Procedure, View, Function, and Parameter allows you to specify a short documentation snippet that is generated into the XML comments of the appropriate object. When you compile one of the generated projects, an XML documentation file is compiled as well. Developers that is the API can see Intellisense in Visual Studio corresponding these description values.

Entity Inheritance



An entity is not hampered by a one-to-one concrete mapping of database objects to classes. The model supports object inheritance. You can define a base type and inherit any number of objects from it. The back-end database synchronization is handled under the covers. There is no special programming around this concept. The only requirement is that a derived class has the same primary key, cardinality, name and data type. When you select an inherited object, you see all the fields like a truly inherited object. There is no way to tell that some fields are derived from a base table in the database. All the complex joins and field mappings are handled under the covers and you never have to worry about which field comes from which table.

In the example above, the associated class diagram would follow. Notice that Person entity is derived from IBusinessObject. This is an interface that defines no save functionality. The child classes of Customer and Employee are derived from Person but also implement the interface IPersistableBusinessObject. This interface exposes a Persist method.

Understanding the Default Mapping

By default, nHydrate maps the domain model to the database as follows.

Entity Classes Map to Tables

Each entity class maps to a database table. The name of the table is the name of the entity class. For example, an Employee entity class is mapped to an Employee table. The plural for objects follows the convention of adding “List” to the end of the object. Because of this, it makes for more readable code ifg all entities are Singular in name like Customer and not Customers.

Fields Map to Columns

Each field in an entity maps to a database column. The name of the column is the name of the field. For example, a FirstName field is mapped to a FirstName column. If you add custom properties to the entity partial classes, this has no effect on persistence layer. You may augment an entity class however your business rules dictate without worrying about how the persistence layer will react. The only information retrieved from the database or saved to the database are fields defined in the modeler, not custom written fields on the entity.

CodeFacades

The mapping of entities and fields is straight-forward, but there are times when you wish to alias your code. The CodeFacade property of entities, fields, views, stored procedures, functions, and parameters allows you to do just this. If this property is set, it will be used in code generation. So if you add a CodeFacade of FirstName to the database field Name, then the database will not change, but your code will reference an entity property named FirstName. This is a great convention for working with legacy system that might use table prefixes or Hungarian notated fields. By using the CodeFacade property your code can look much prettier than your database.

Associations Map to Foreign Key Columns

Each one-to-many or one-to-one association in an entity is created in code with a navigation property that points to its associated object. These navigation properties are the way you walk relationships. A Customer entity might have an OrderList property (in a one-to-many relation) and each associated Order entity would have a CustomerItem back link property. The navigation is handled in the background. It is completely controlled by your model settings. When you define a relation between two entities based on specific fields the walking functionality is generated based on it.

Fields

Fields have many properties that will be familiar to any developer. Each has a name and data type of course. There are also properties used to define validation rules on the field.

Primary Keys

When a field is marked as a primary key it will be one of the database primary keys for table represented by the parent entity. Each entity much have at least one field defined as a primary key.

Identity

Many people choose the have an integer based field for a primary key and mark it as an identity. This allows the database to manage the primary key creation for you. This is not required for an entity by it does remove key management from the application by assigning it to the database. If you have no business rules that require specific keys (like customer name) then an identity is a good choice.

Default

The *Default* property is used to define a default value for the field in the database. When an entity is created the default for the data type is used unless an explicit default is set. For example a non-nullable int will default to 0. However is you define a default of 3, this will override the data type default. In this example, unless a new value is assigned to the entity property the default of 3 will be used when creating the database row. This is enforced in the code as well as in the database.

Validation Properties

There are other properties used for validation: Nullable, Min, Max, Length. These are discussed later in the validation section.

Single Field Indexes

If you wish to create a single field index on table, simply set the field's IsIndexed property to true. This will create a database index on the underlying database column. This is by far the easiest way to add an index without writing any manual SQL scripts.

You can also add additional or multi-column indexes in the nHydrate Model tree. Each Entity object has an Indexes list. You can define a custom index much like you can in SQL Management studio. An index is made up of one or more columns. These defined indexes will be turned into SQL scripts by the installer.

Is Unique

Primary keys are unique by definition; however there are times when you wish to have unique constraint on a field but do not want to make it a primary key. Perhaps you have a business rule that customer names must be unique. You would not want to make that string a primary key, but you could enforce uniqueness in the database. Simply set the IsUnique field property to true and a unique constraint will be created on the database.

Is ReadOnly

If you wish to be able to retrieve a field value from the database but never set it, then mark it as read-only and there will be no public setter for the field.

Is Browsable

The browsable attribute ensures that a property will be displayed in a property grid or other controls when bound. If you do not want a field to show up in a UI display, set this property to false.

Overridding the Default Mappings

In greenfield development, it's a good idea to keep to the default mapping. This ensures that terms are used consistently across the domain model and the database, so that everyone is using the same language and confusion is avoided. In brownfield development, this often isn't possible, and even in greenfield projects there may be cases where the default mapping is problematic.

Mapping Individual Tables and Columns

The most common scenario for overriding the default mapping is that you are working with an existing database whose terminology you don't want to carry forward into your domain objects. In this case, you can override table and column mappings on an ad hoc basis, specifying an override for each table or column where you want to change the name.

Mapping Individual Tables and Columns in the Designer

- To specify the name of the table for an entity class, select the entity, go to the Properties grid, and set the CodeFacade property.
- To specify the name of the column for a field, select the field, go to the Properties grid, and set the CodeFacade property.

If you have dragged a table into the model from a database, and you're dissatisfied with the inferred names, you can quickly rename an entity or field while retaining its mapping by using the CodeFacade property.

If you have dragged a table into the model from a database, and the primary key column has a name other than ID, nHydrate will automatically create the identity column mapping for you so you must change the name property.

Reserved Words

Occasionally you will want to use a name in your domain model which is a reserved word in SQL. An example is "int" or "namespace", which clashes with C# keywords. This is just something to keep in mind. The validation rules will notify you when an identifier is invalid. The model will not generate when there is a model validation error.

Advanced mapping concepts

The generated objects have concepts attached to them like data type, nullable type, enumerated type, and error-checking. Depending on the settings of an object's fields in the model, the generated object will have certain properties associated with each field. Of course a field's data type is used to create a corresponding .NET type for the generated object. However there are more advanced features as well. A field marked as nullable, is generated with a nullable .NET type. You can look at all non-string fields and know if a null value is permissible since the .NET type either allows it or not. A string is a reference type, so you cannot determine this attribute just by its data type.

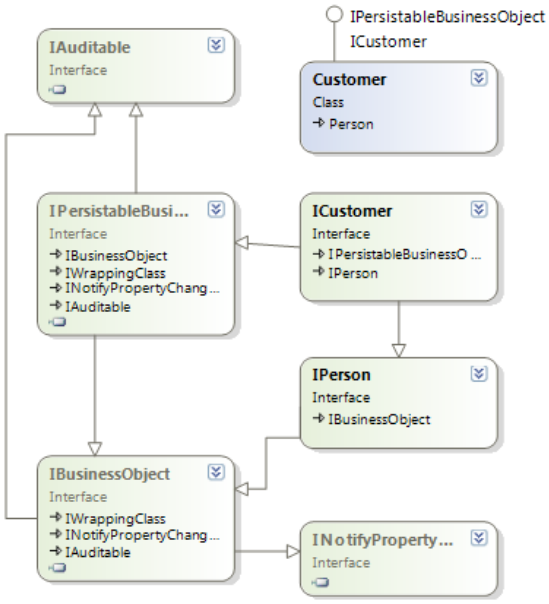
Strong error-checking is implemented by not allowing client code to assign values that will break the database, as much as possible. If a null value is passed to a non-nullable field, an error is raised. All objects have meta-data descriptors like length, nullable, type, etc. You can use these attributes to limit UI data entry to valid data. There are also common business object functions that allow you to set or retrieve data in a common way, based on interfaces and field enumerations. The common base functionality of the framework allows for the construction of application to be written that are not domain specific. In other words, you can write applications that can take any generated API based on any domain model and plug it in for user manipulation.

Enumerated types are static types that map to unchanging (or seldom-changing) data. A user typed entity is a good example of this. You may have a number of user types but they almost never change. Instead of assigning “magic numbers” to these values, simply let the generator handle it for you. From there you can assign an enumeration in your code and not a number. You code is much more readable and developers can tell immediately what the value means.

Interfaces and base classes

All generated objects implement one or more interface or abstract base classes. This allows for the creation of very versatile applications. All generated mapped objects implement numerous interfaces, some based on whether the object is a read-only or persistable or other such permissible actions. You can build applications that take in base objects and interact with the user based on the type of object passed in. At run-time you can get and set an object’s fields without know exactly what type of object it is. The audit fields are accessible by interface as are search objects and primary keys.

The business objects come in two flavors, read-only and persistable. You can define objects as read-only, so they can be selected but not saved. This comes in handy for type tables and also base object tables. The latter is quite interesting in that you can define a table that has no meaning by itself but is the base class of a concrete object. An example of this is an abstract Person that is the base for Customer and Employee. There may be no such thing as a generic Person in your business rules only Customers and Employees. In this case, make the parent table of each the Person entity and make the Person entity read-only. In code, developers can only create Customers and Employees but not Persons.



Notice in the above diagram that the interface model is quite robust. Customer inherits not only from Person but also ICustomer which in turn inherits from IPerson. All business objects are auditable.

Persistable objects are derived from the less functional, read-only, objects and interfaces. This model exposes many possibilities when creating user interfaces and business logic.

Callbacks and notifications

The generated objects follow the observer pattern in that you can register for notification of events. Each object has a global event that is raised before and after a field is changed. There are also events for each individual field so you can capture a message for a single field.

Compile-time checking

Since the generated framework is modeled, you get strongly-typed objects. This allows you to have compile-time errors, not run-time errors. When an object field needs to be changed in any way, be it the data type, name, nullable, etc, simply change the model and re-generate. When you try to compile your application you will get errors found by the compiler. This is much better than accessing fields by a string comparison that cannot be checked by the compiler.

Extending Objects

All objects are generated in partial classes. There is a shell class that is generated once and will not be overwritten and a “generated” class that is controlled by the model generated and will always be overwritten. This allows you to add your custom code the shell class and have it appear as a fully integrated component of the object.

Paging

An important feature of any database access technology is the ability to page through large sets of data. Not only does the generator allow you to do this but it allows created a strongly-typed paging object for all object types. This allows you to define in a very precise way how to page through data. For example, if you are sorting by a particular field and later remove this field from the model, a compile-time error will be generated since everything is strongly-typed. Ordering is very important when paging, since you the next page of objects returned must be deterministic. That said you can order by any number of fields with an entity’s paging object.

Standards

The generator creates code based on standards of the day. The entity objects are based on a hierarchy of interfaces and base classes. Each object can describe metadata about itself like type, size, friendly name, façade, etc. The generated framework implements industry standard software patterns as well. All business objects object implement the **Observer** pattern for field notifications. The **Active Record** pattern is used as the main interface with the database. This allows for fast and very flexible querying techniques. For more disconnected requirements, the DTO layer implements the **Repository** pattern. This separates database access from the developer and allows objects to be sent over the wire. The **Visitor** pattern is employed on all business collections to allow abstracted iteration or processing on collection objects. The **Composite** pattern is used to define entity relationships. Dependency walking is a relationship hierarchy that requires no special coding on the part of the developer. All entities have composite objects of their children and parents. This also aids in joining objects with no visible join clauses or statements like SQL. The **Interpreter** pattern is implemented in the mapping of database objects to .NET entity objects. All entities and fields can have a facade that masks the underlying database field. Keep in mind that one entity can map to multiple database tables and relationships. All of these coding standards are implied in the generated code and derived from the model. No special architectural knowledge is necessary to use these concepts.

Aggregation

Another convenience feature is provided with strongly-typed aggregate methods. Instead of selecting a bunch of data and performing aggregate operations, you can call one of the aggregate functions with one line of code. There are functions for Min, Max, Count, Sum, Average, and Distinct. They are all strongly-typed to the data type of the field begin queried. In other words, if you are querying the maximum cost (decimal) field, the result is returned as a decimal not an object. The where clause can be as complex as you wish based on relationships defined in the model. The ways in which you can define the where clause is not predefined. It is completely free-form based on the model.

Special Entity Types

In addition to standard entities, there are special types. These perform specific functions that help you code better.

Typed Entities

Type entities are entities that define a non-changing table. An example of this would be EmployeeType. Assume you have an Employee table and an EmployeeType table. Each Employee has a type like Manager and Standard. We can define an EmployeeType table and add two static data rows to it corresponding to these two types. The nHydrate system will create a read-only entity in code with an enumeration defined for its data.

Typed entities must have an identity integer primary key and a Name or Description property. The static data defined for these fields will be used to create a code enumeration with the primary key being the enumeration value and the Name/Description being the enumeration name. Now in code there is no need to use magic numbers anymore.

Use enumeration to set a typed value

```
//Magic number
customer.CustomerTypeID = 1

//Enumeration
customer.CustomerTypeID = (int)CustomerTypeConstants.Manager;
```

While the described example above used a typed entity that has a backing database table, you do not need one. If you wish to define static data and have an enumeration that has relations with other entities, you may do so with out a database table. There are two kinds of typed entities Materialized and CodeOnly. The first has a concrete database table that has real, database relations on it. The other has an enumeration and a relationship in code, but there is no need to have a database table for the entity. A database table is needed for reports other other applications that want to pull the typed value out of the database. It may also be useful if you wish to have real, referential integrity in your database.

Static Data

You can actually add static data to the model to initialize data in tables. If you have a type table, you might want to ensure that certain data is present in a table when a database is created. There is a GUI that allows you to set each field in a table to a specific value.

You may add static data to any entity (database table) however this is primarily designed for type tables. This allows type tables to be populated from the model with non-changing data and have an enumeration generated from it. Since in the end, this data is used to create a script so you can use it for

other table types too. The installer project just adds this to the CreateData.sql script file.

Associative Entities

Relational databases do not inherently handle many-to-many relationships. A common way to handle this relationship is to create a tertiary table that holds relations to both primary tables. The model handles this the same way but hides the intermediary table from you. There is no reason to see it. What you really want is the list of related items, not a like to a table that contains the related items.
To handle this you would add the intermediary table and mark it as an Associative table. It contains the primary keys of both primary tables and all fields are marked as the table's primary key. When the code is generated, a table1 object has a list of table2 objects and a table2 object has a list of table1 objects. It is truly many-to-many.

Immutable Entities

Related to type tables are immutable tables. All type tables are immutable but not all immutable tables are type tables. If you mark an entity as immutable there is no persistence method on the objects and there are no pubic setters for properties. They are truly read-only objects.

Working with Views

Working with views is much like working with entities. You can select them the same way and use them in code the same way. The notable exception is that they are read-only. There are a number of assumptions that follow from this concept.

- Properties are read-only (no setter)
- Views cannot be created (private constructor)
- Views cannot be added to a context
- A context SaveChanges call has no effect on a view

To add a View to the designer open the nHydrate Designer window. Right-click on the root node and choose the Add New View menu. This will create a view object. Select the newly created object to set its properties. At a minimum you must add a SQL statement and one field.

After you select a view object, you can access its properties and display the data much like any other non-read-only entity.

Select a view with LINQ and access a property

```
var view = context.MyCustomView.Where(x => x.Name == "John").First();
var name = view.Name;
```

Invoking with Stored Procedures

It is sometimes necessary to encapsulate very complex queries as stored procedures. You can define and invoke a stored procedure from nHydrate either to load a set of entities, to calculate a single value such as a count or total, or just to execute it with no return value.

Working with stored procedures is similar to working with views. You can select them but they have parameters. Once you have a reference to a stored procedure object, you can use it in code the same way as it has read-only properties. They are read-only as well. The same assumptions as views also apply.

- Properties are read-only (no setter)
- Stored Procedures cannot be created (private constructor)
- Stored Procedures cannot be added to a context
- A context SaveChanges call has no effect on a view

To add a Stored Procedure to the designer open the nHydrate Designer window. Right-click on the root node and choose the Add New Stored Procedure menu. This will create a Stored Procedure object. Select the newly created object to set its properties. At a minimum you must add a SQL statement.

After you select a stored procedure object, you can access its properties and display the data much like any other non-read-only entity. Notice that you need to specify parameter values when you call the stored procedure. Also the call does not return an IQueryable but an IEnumerable. If you define one or more columns for a Stored Procedure it will be a read-only entity just like a View. You can access fields of the Stored Procedure as entity properties.

Select a view with LINQ and access a property

```
var myproc = context.MyStoredProc("some value");
var field = myproc.First().Field1;
```

There are different kinds of Stored Procedures. If may create one with no parameters and no fields if you wish. If a Stored Procedure has no fields, it is considered an action. Actions are not generated in the Entity namespace (or folder) but in the Action namespace (and folder). You call an action with a static method and it returns void.

Invoke an action with one defined parameter

```
Acme.CustomerTest.EFDAL.Action.MyAction.Execute("param value");
```

Database Considerations for Stored Procedures

When loading entities through a stored procedure, the returned record set is treated as if it had been returned by performing a table SELECT. The exception is that the returned entity is read-only. You cannot create a new one and all properties are read-only. The set of columns returned from the stored procedures must correspond in name and type to the fields of the entity.

Building Applications with nHydrate

You can use nHydrate in a variety of application architectures, from Web pages to rich clients and service-oriented distributed applications. In fact, you can use it anywhere Entity Framework can be used since it is Entity Framework. In the following chapters, we'll discuss in more detail how to use nHydrate in specific scenarios. First, however, we need to address some areas which are common across application architectures.

Configuration

You can configure nHydrate through code or through the configuration file (web.config or appname.exe.config depending on the type of application). Since the generated code is Entity Framework under the hood, you can configure it in any way that is valid for Entity Framework.

How to Configure a generated API

Since the generated code is simply Entity Framework, you can configure it just like any other Entity Framework API. A common way to do this is in the app.config or web.config file.

Web.Config file connection string configuration

```
<connectionStrings>
<add name="NorthwindEntities" connectionString="metadata=res://
*/Acme.Northwind.EFDAL.Northwind.csdl|res://*/
Acme.Northwind.EFDAL.Northwind.ssdl|res://*/
Acme.Northwind.EFDAL.Northwind.msl;provider=System.Data.SqlClient;provider
connection string=&quot;Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=SSPI;Connection
Timeout=60;multipleactiveresultsets=true&quot;"
providerName="System.Data.EntityClient" />
</connectionStrings>
```

Another way to setup the database connection is with a connection string in code.

Define connection string in code

```
var connection = "metadata=res://*/Acme.Northwind.EFDAL.Northwind.csdl|
res://*/Acme.Northwind.EFDAL.Northwind.ssdl|res://*/
Acme.Northwind.EFDAL.Northwind.msl;provider=System.Data.SqlClient;provider
connection string=&amp;quot;Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=SSPI;Connection
Timeout=60;multipleactiveresultsets=true&amp;quot;"
using (var context = new NorthwindEntities(connection))
{
}
```

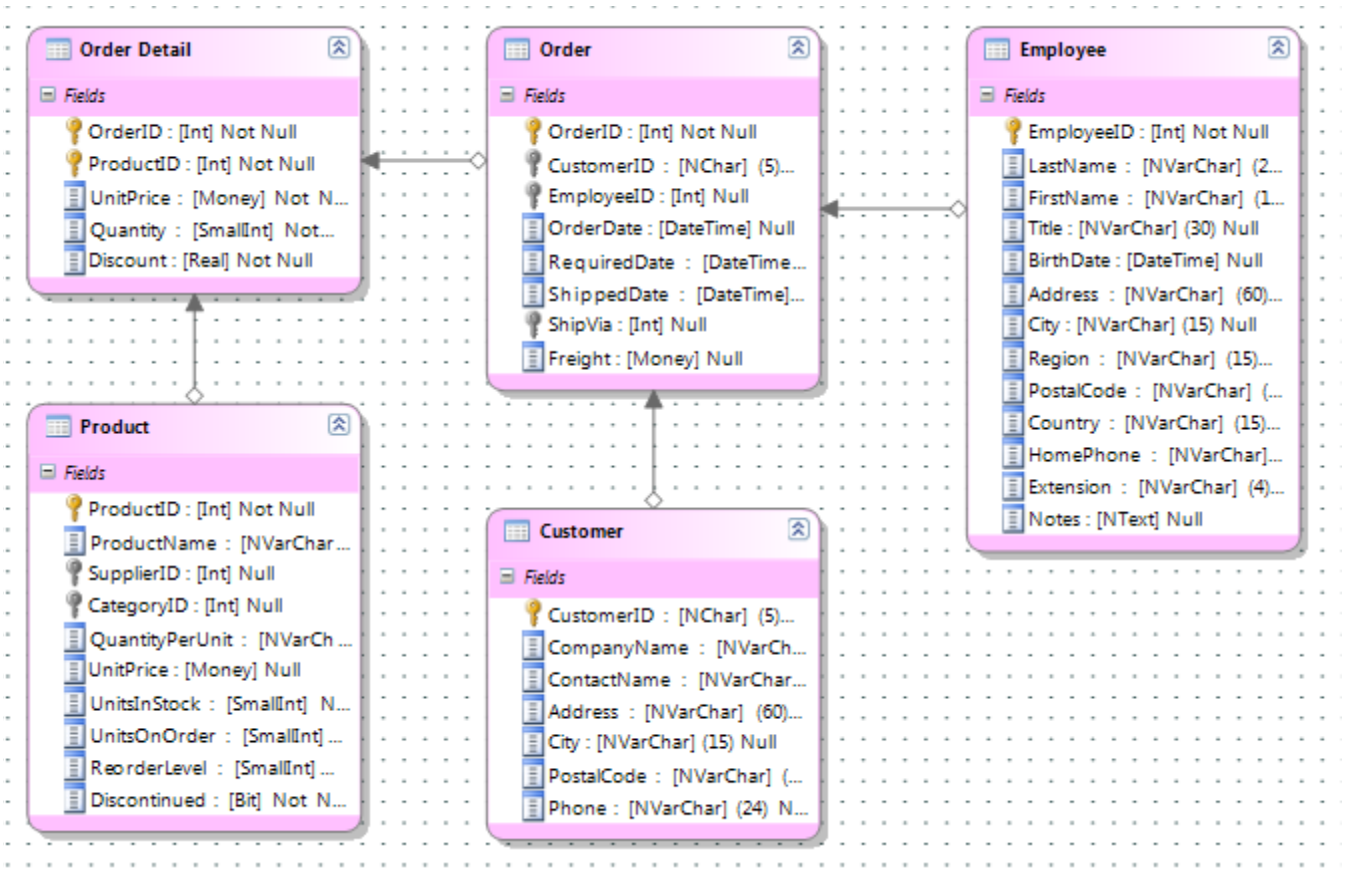
You can also use the pre-defined Entity Framework *System.Data.EntityClient.EntityConnection* object type to define the connection and pass it into the constructor of a context.

Modules

Modules are an advanced feature that is useful for large projects. This concept allows you to group model objects into logical modules that will be created as completely separate projects in Visual Studio. A Module is a logical grouping that will manifest itself as a separate API. This allows you to have a common model and share it amongst teams. This is quite useful for teams that work on products that have some commonality in their databases (or application parts) but should still be considered different API.

Let's discuss this more in detail. We can use modules to group functionality into separate API layers. Assume there are two teams working on two applications or pieces of an application. Each team has its own piece of a database (or even a different database) with which to interact and the teams have a lot of overlapping commonality. Team1 has 3 tables (Order, Order Detail, and Product) that overlap with Team2 and an additional table not shared with Team2 (Customer). Team2 shares the common three tables but has an additional table (Employee) not shared with Team1.

Full Database

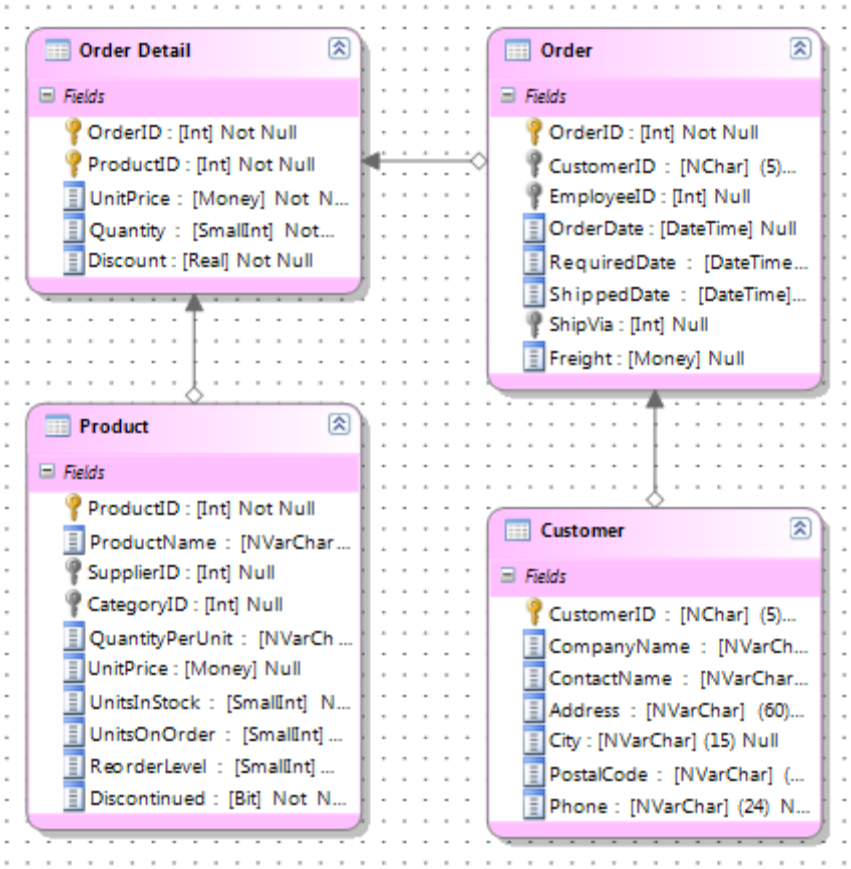


The goal is to share a database but have a limited API that will only expose the entities that each team needs. Team1 will handle the customer aspect of the application. They will lookup a customer by id, name, or email perhaps and then walk the relationship to the orders for that customer and finally display the details for each order. The API they use has no entity or notion at all of Employee.

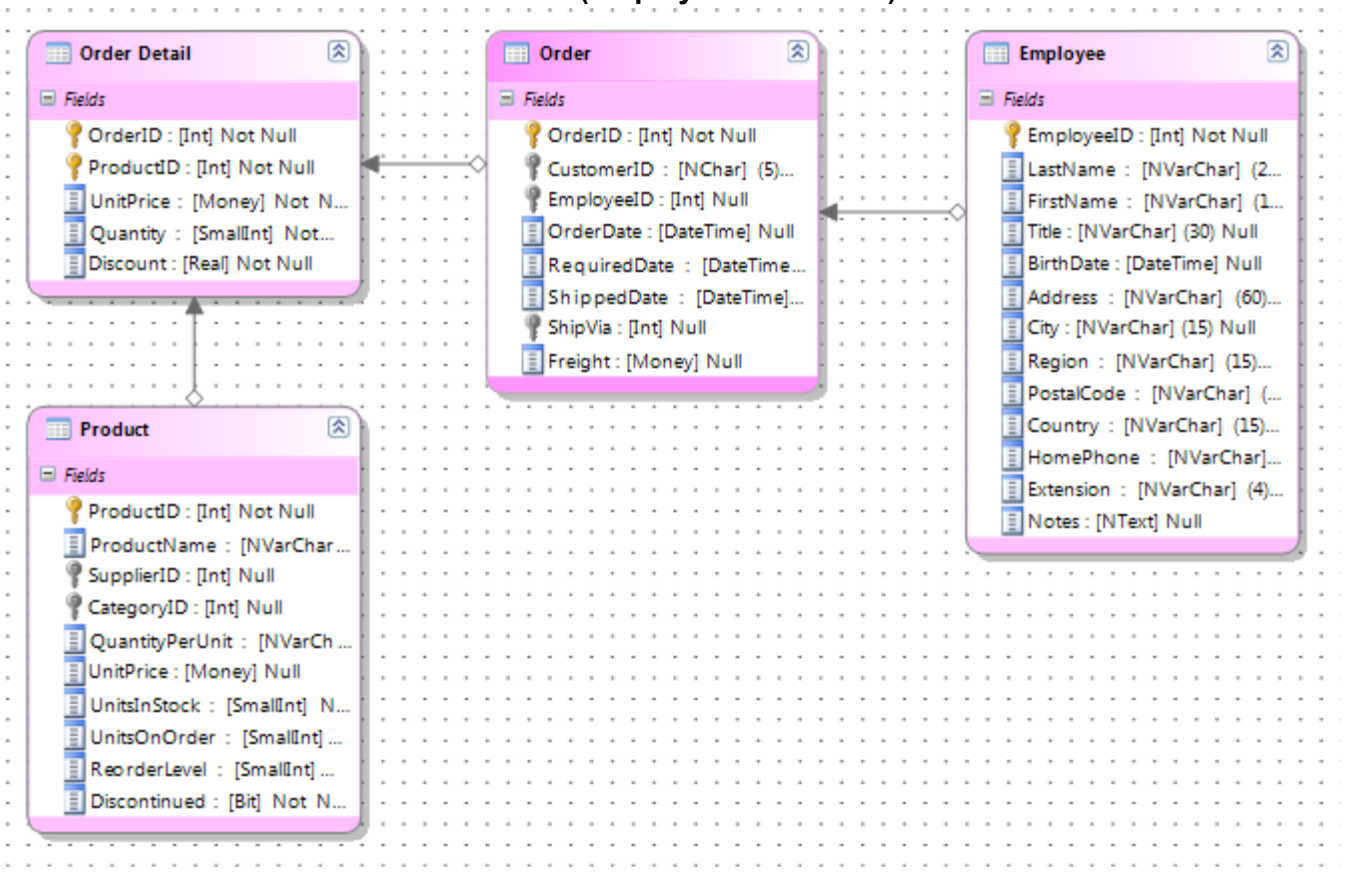
Team2 must write similar functionality however they only care about it from the Employee side. They might lookup an employee and use that entity to walk to the Order table and finally down to the details for the order as well. Team2 has no knowledge of the Customer table. It is not in their API.

Now the functionality structure allows the two teams to use the same database while providing the ease of maintaining one model. If any table in common changes its structure both APIs will be affected. For example, if the Order table adds a column, Tax, then both APIs will be regenerated. The installer projects will reflect this new column and each Entity Framework API will also reflect this change.

Module 1 (Customer + Common)



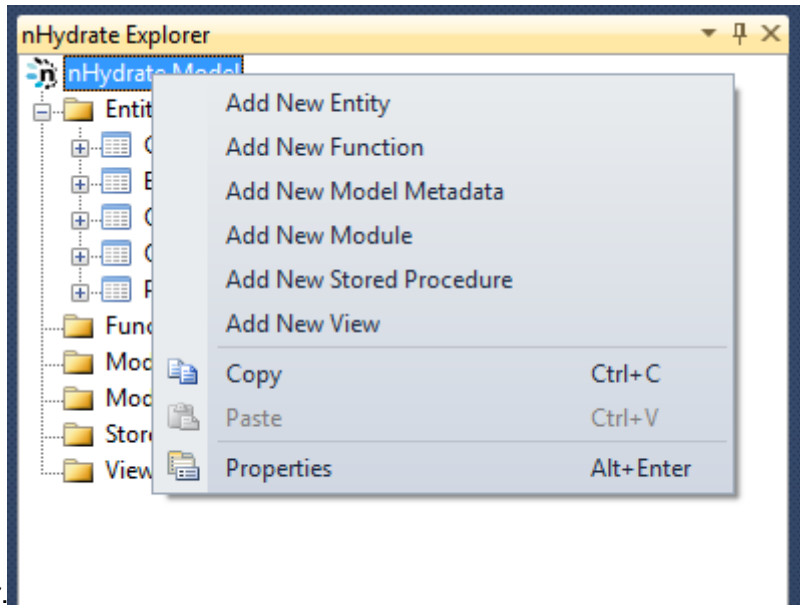
Module 2 (Employee + Common)



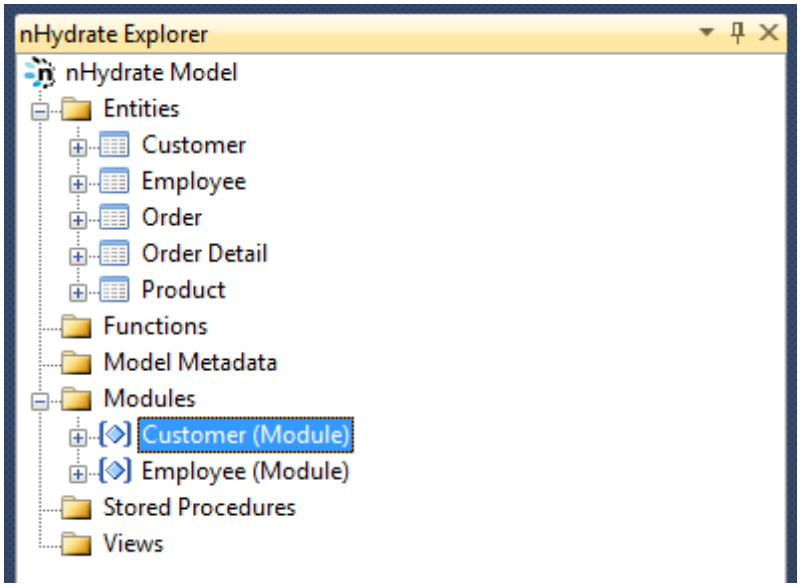
Normally to create a different API, you would create a model for each team and each one would generate a separate API. However with the overlap, there is a lot of duplicate maintenance of models. With two models, a change to the Order table would require a change to both models. This is error prone and inefficient. With a single model, the change would propagate to both module APIs from the same model.

We can easily accomplish this split by using the module functionality of nHydrate. First select the model and in the properties window set the UseModules property to true. Now the model cannot be generated until there are some defined modules. In the nHydrate Explorer window, right click on the root node and add two modules named Customer and

Employer.



Once these modules are created you see them in the explorer tree like all other model elements.

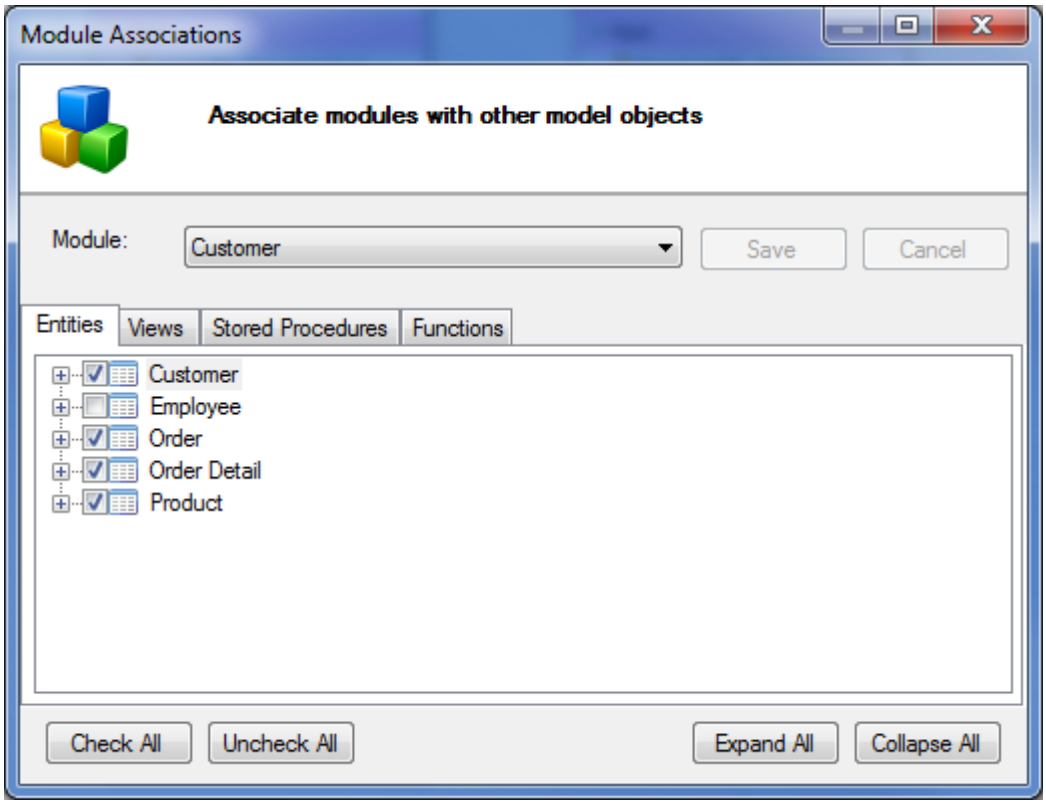


Now you must assign all model objects to one or more modules. Right click on any empty spot of the model canvas and select the menu Model | Module Associations. This will bring up the module assignment window. From here you can choose a module and assign objects to it. A table (or any other object) can be assigned to any number of modules. In this way modules can have as much overlapping functionality as your business rules call for.

Views, Stored Procedures, and Functions have all or nothing inclusion. In other words, they are in a module or they are not. You cannot include half of a view into a module. However tables are different story. You may include partial tables into modules.

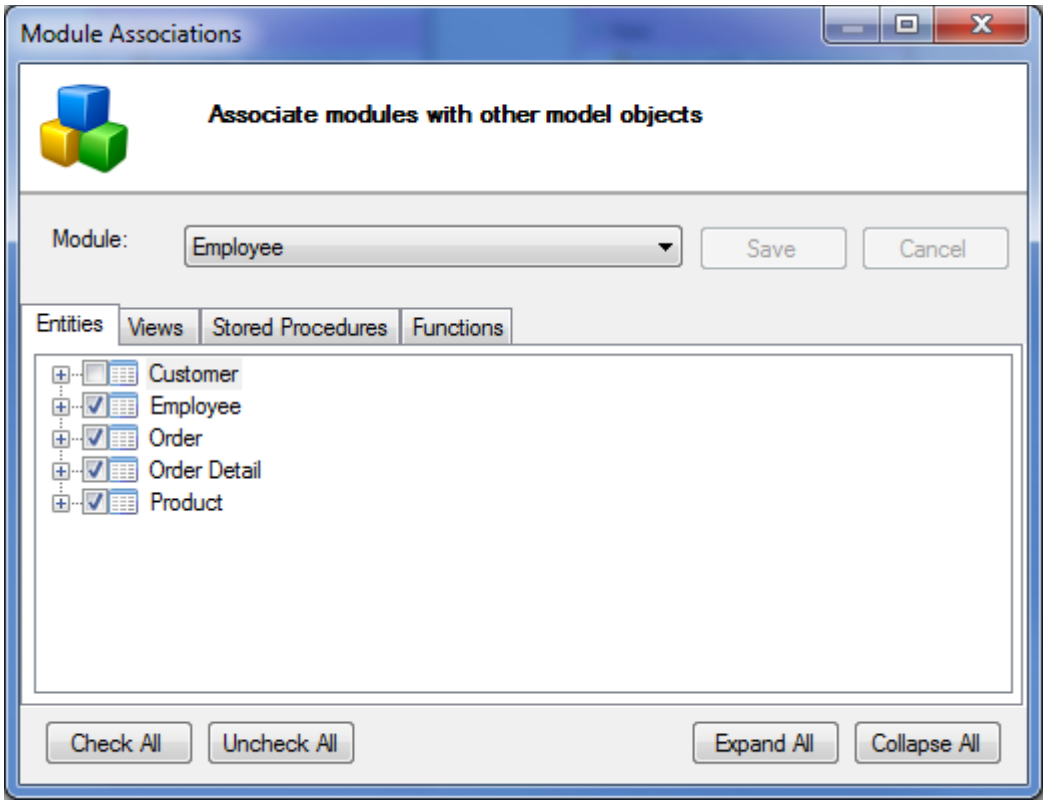
Partial tables are very useful when component functionality is required. You may have a very large table (lots of columns) however an application part may only need part of it. In our Employee module, perhaps there is no need to expose the Order table's Freight property; maybe it is not necessary or maybe it is a security concern. In any case, if the developers using the Employee module will never need the field then it need not be included in the first place. When you assign objects to the Employee module, simply uncheck the Freight field from the Order table. Now when the module API is generated there is an Order entity but it is missing the Freight field. Rest assured, this field is still in the database but cannot be accessed or set from the Employee module assembly.

Customer Module



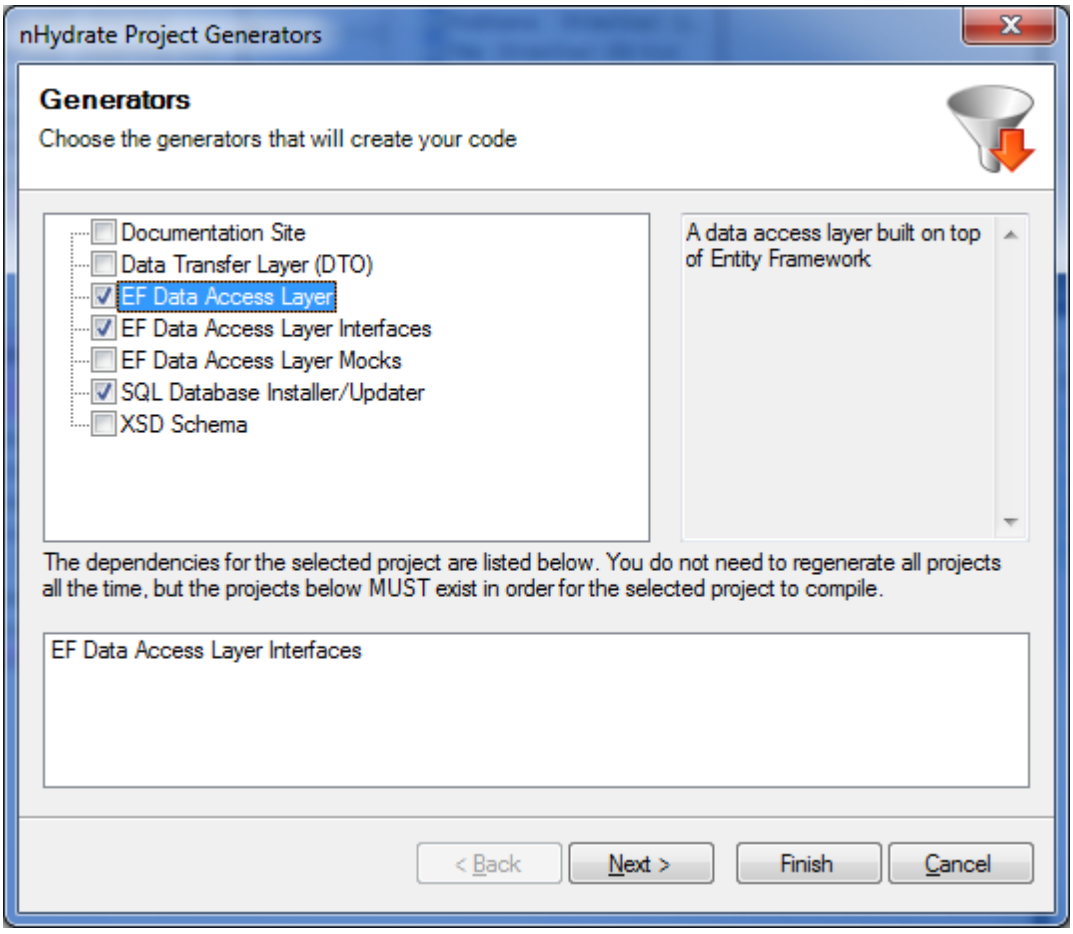
Now we can define the Employee module. Select the specified module from the drop down box and check which objects to include.

Employee Module



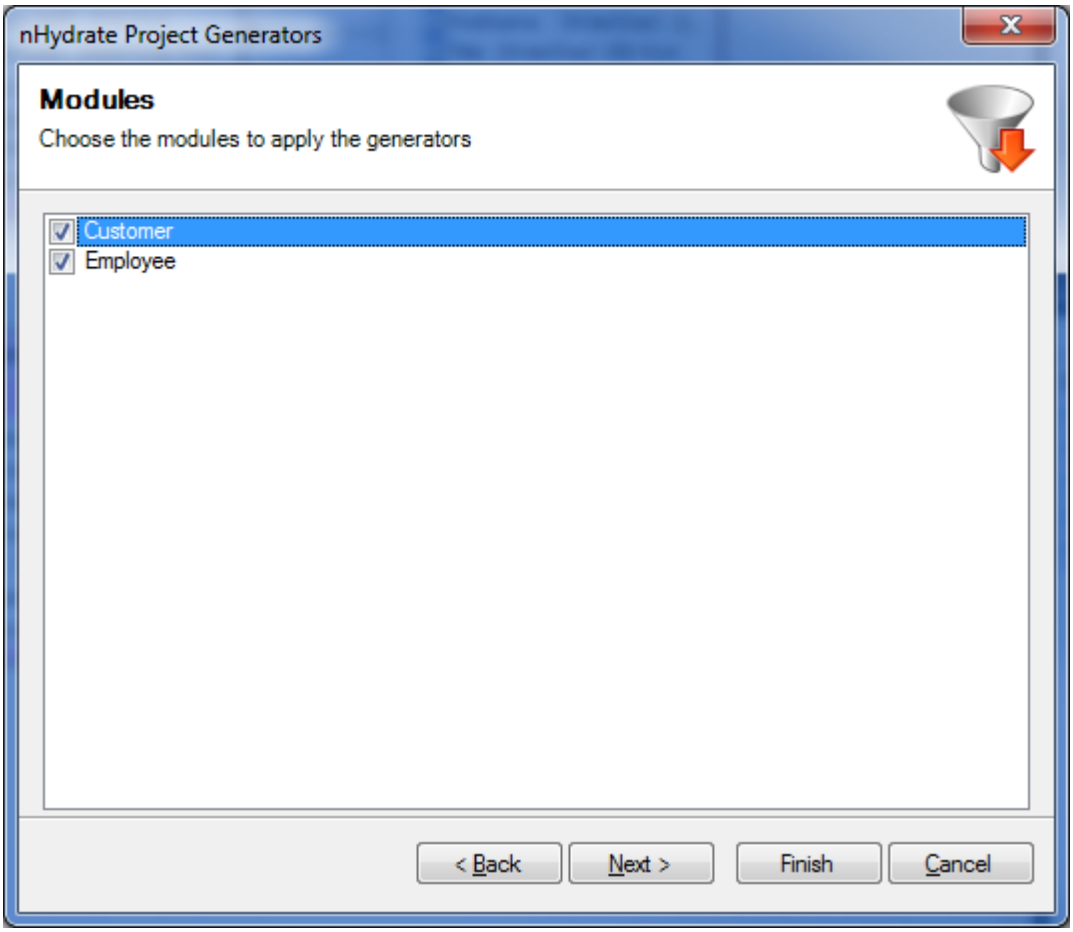
Now that we have defined our tables and assigned them to modules, we can generate code. Right click on any empty spot of the model canvas and choose the Model | Generate menu. From here you will be prompted for what to generate.

Generate Dialog



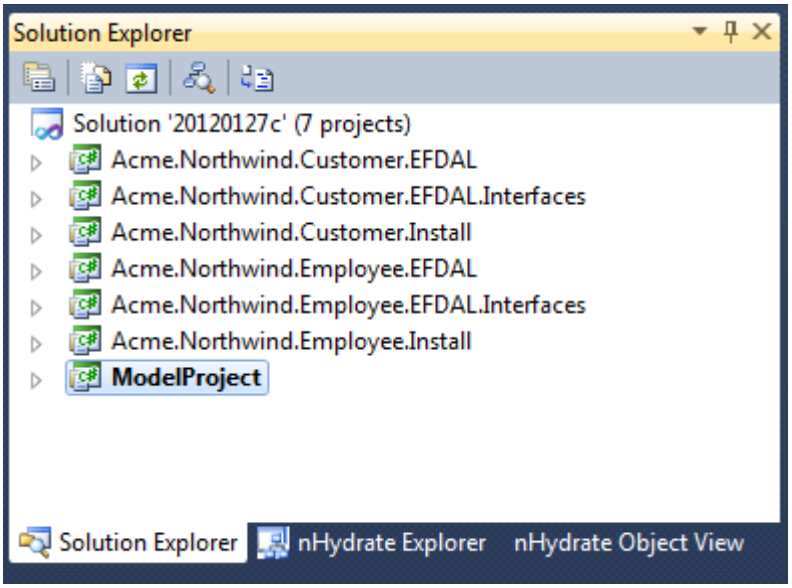
You must also choose which modules to generate as well.

Choose Modules



Once the generation is complete, you should see the new projects in the Solution Explorer.

Solution Explorer



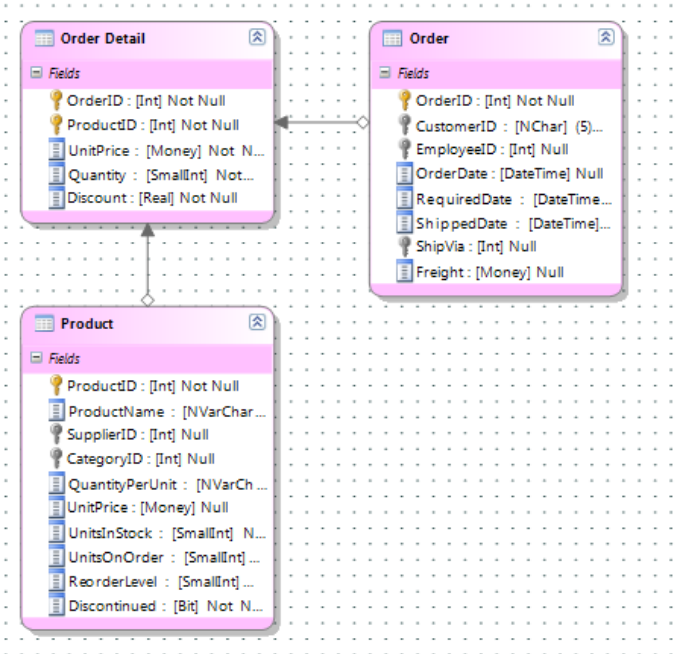
A Module will be run through the specified generators to create discrete APIs. Normally when not using Modules, a generation of a model would create a separate project for each generator. For instance, to create an Entity Framework API, three generator templates are used: EFDAL, Interfaces, and Installer. When using Modules, there will be a new project for each Module and each Generator. In this example, if you define two modules: Customer and Employee, there will be six projects created not three (3 generators x 2 modules = 6 assemblies).

Common Modules

We can now manage a single model for the entire application but actually generate a unique API for each team. This actually has some big benefits beyond the obvious ones we have covered. If you think about the two libraries generated above there is still a problem. We cannot write common libraries that are shared between teams. This is because each of these libraries is specific to one of the implementations above: Customer or Employee. What we really need is a common API.

We would like to have a common business library that can talk to either database (these two APIs might actually talk to the same database or different databases) with no chance of run-time errors for missing objects. If we create a Module that is a subset of both modules above, we have a common framework API. Create a new Module and add the common entities to it. Now we can write a common framework assembly that does common actions for both teams. This means that there are actually three distinct APIs now: Employee, Customer, and Common. However they are all managed from the same model.

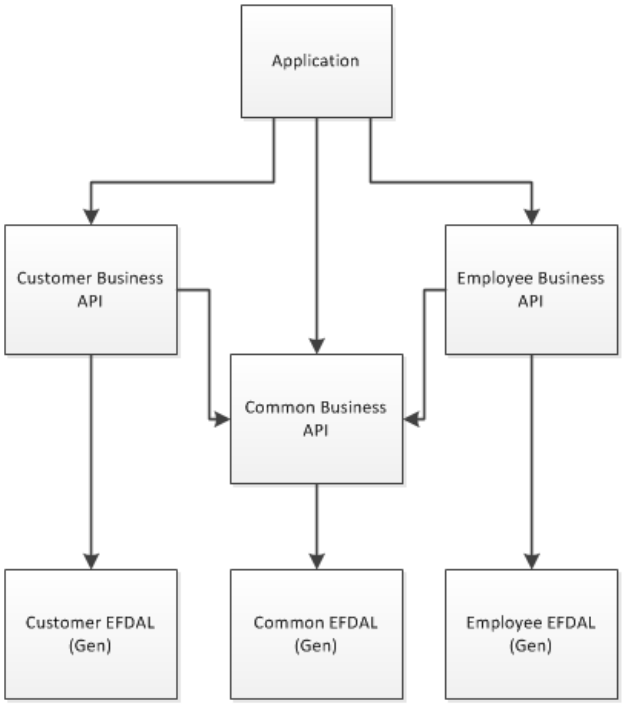
Module 3 (Common)



Since this is managed from a common model, it is always up to date. If we change one of the common tables in the model, a regeneration will ensure that all APIs and Installer projects are updated. Once we update the databases, the common API works on more than one database schema. We are now free write a shared assembly that encapsulates global (cross-module) business rules that all development groups can use. This shared assembly uses the common generated API to apply common business rules and can be referenced by both teams to perform supply functionality. Now there is no need for both teams to write the same code twice to duplicate business rules that they both share. We can write once with no duplicate effort.

Of course these would be low level, shared assemblies in our application. Each team writing its application part would also write custom assemblies that use its own specific API as well. This allows two applications with overlapping functionality to share code.

Application Structure



The diagram above shows a possible application configuration. The main application knows about the libraries that Team1 and Team2 have written: Customer and Employee business API. These two libraries can make use of the common business API since it is a subset of both. Each business API uses a generated Entity Framework API. All generated API libraries are managed by one model. A change to the model can be propagated across all APIs by regenerating. The custom API hand-written by each team use the generated APIs so they also talk to the proper version of the database, since the installer project always keeps the database up to date with the model.

Module Rules

The above example describes how one module is a subset of another. If this is the way you wish to organize your model, it would be best not to do it manually. If you add a table to the Common module and not to the Customer module, the Customer module is no longer a superset of Common. What we really need is a way for the model to enforce this model rule.

A Module Rule allow you to define inclusion or exclusion rules between modules. These will be enforced by the model and provide validation errors if necessary. A rule can be defined as a subset or an outerset. In the validation process, the rule is applied to ensure that the defined rule is met. The subset or outerset rule will be applied only to the object types specified in the inclusion property. If you want to ensure that the Common module is a subset of the Customer module, you could define a module rule as follows.

1. Create a module rule on the Common module.
2. Set its status to *Subset*
3. Set its dependent module as Customer.
4. Set its Inclusion property to Entity only.

This ensures that views, stored procedure, and functions are not checked for meeting the subset rule. This defined rule will cause a validation error if all the entities in the Common module are not in the Customer module and you will not be allowed to generate.

An outerset rule does much the same thing except that it ensures that there is no overlap between the two modules. You can apply any number of module rules. This is a very nice feature if you are managing a model with many interacting modules.

Auditing

In addition to tracking changed on the row level, you can also track entire tables. When a table is marked to allow auditing, a shadow table is created and maintained by the system, of all changes to the table. This includes additions, updates, and deletes. It is as easy as setting the table property and table activity is recorded no matter how the table is modified. This even includes modifications made outside of the generated framework.

Create Tracking

When an entity has its *AllowCreateAudit* property enabled, all objects will store the user and time of creation. When on entity object is declared, added to a context, and saved a row is created in the database. The *modifer* property of the context is used to define how created the object and this information is stored with the record in the *CreatedBy* field. The *CreatedDate* is set the current time. These two values are never reset. Their are corresponding rad-only properties on the entity, but there is no way to set them for persistence.

Create a context with a defined modifier "Admin"

```
var startup = new ContextStartup("Admin");
using (var context = new CustomerTestEntities(startup))
{
    var item = new Customer();
    context.AddItem(item);
    context.SaveChanges();
}
```

Modify Tracking

Modification auditing is much the same except that it does get reset each time an entity is saved. The default data columns for this audit is *ModifiedBy* and *ModifiedDate*. When an entity is created, these properties match the create audit information.

Concurrency

The *AllowTimeStamp* property specified that the database table should have a timestamp field and it should be used for concurrency. If you're developing a multi-user system you will want to think about concurrency. For example, you may enable optimistic concurrency checking and trap that exception on save if another user has edited the same entity between you loading it and saving it. How you deal with such a concurrency violation would be application specific but it is a situation you will need to address.

Be wary of concurrency concerns. In a multi-user system it's important to make sure that data is not lost by saving stale data. nHydrate always uses optimistic concurrency, so it won't prevent two users

editing the same record at the same time. Use concurrency checking to prevent accidental data loss. You should handle the thrown exception for the second save to prompt the user or perform some sort of error handling for concurrency violations.

Validation

There are different kinds of validation in the generated entities. There is rule-based validation and code validation.

Code Validation

Depending model attributes there is validation built directly in the code. The simplest is string length validation. If a field is of a data type with a defined length, the generated code will validate that the value is not too large for the field in the property setter. For example, if a field is defined as *varchar(50)* then the field setter will actually throw an exception if the string supplied is greater than the 50 character limit.

Also fields marked as non-nullable will throw an exception if a null is sent into them. This is not possible for non-string data types. If an integer is marked as non-nullable then the entity property will be of type *int* which will not accept a null anyway. However strings are different. A string can be null in .NET even if the entity field is marked not to allow it. Therefore there is validation code built into the property setter to catch for this situation.

Numeric fields (either integer or real) can also have minimum and maximum values checking applied to them. Each field has a MinValue and MaxValue property. When set these are used to add additional validation checks in the property setter method.

Rule Validation

In addition to the built-in code validation of the property setters there is also a rule based way of adding validation. Each entity has an *IsValid* method that can be invoked to determine if there are validation exceptions with the object. These exceptions are defined in another method that you override named *GetRuleViolations*. If you have specific validation rules you wish to add to an entity type, simply override this method. Upon checking your business rules, you can add exceptions to the collection. The idea is that you would use the rule collection to display messages to a user.

Refactoring

Another nice feature is refactoring. On the model canvas, there is a right-click menu for refactoring. It provides options for restructuring your model. When you apply a refactor, the necessary change scripts are automatically created on the next generation.

Split Entity

A common source of pain for modellers is table splitting. There are times when you have a large table and realize that it really should be two tables. This involves changes script to create a new table, copy data and maybe added a relation between the existing table and the new one.

Now this can be handled effortlessly. Simply select a table on the model and choose this refactor option. A wizard will walk you through the steps to perform this action. It is quite simple. Specify a new entity name, whether to use the same primary key and if there is a relation. Once applied the new entity is created. More importantly, upon next generation, the scripts to perform this complex action is created for you, including coping the data between the tables.

Combine Entities

Along with splitting an entity, you may want to combine two of them as well. This is easy too. Simply select two entities in the model and choose the refactor menu item Combine Entities. This operation will take the fields from the secondary entity and merge them into the primary entity. The secondary entity will then be removed. The scripts to perform this is also emitted into the next, generated change script.

Create Associative Entity

This option is a short-cut for creating an associative entity. After choosing two entities on the model, you may choose this refactor option to be prompted for details of the new table. An entity name is automatically generated for you, but you may override it. Once the change is applied an associative entity is created with the primary keys of the two source tables. Also two inbound relations from the two source entities are created as well with the entity being marked as associative.

Replace all NText

This option allows you to change all of the deprecated NText fields and replace then with Varchar(max) fields. The NText field has been deprecated by SQL 2005, though it is still supported. You can change a some or all of these fields in your model with the pop-up dialog provided with this refactor option. A dialog shows you all instances of fields with the NText data type and you can choose to replace them with the newer Varchar(max) data type. On the next geneartion, the change scripts for the database are automatically incorporated into the installer.

Installer

The installation project is a big piece of the functionality provided by the framework. One of the big challenges of any database driven project is keeping a database and code in sync. When changes are made to a model, the existing database is by definition out of date. Normally somebody must manually keep these two layers in sync. Sometimes it is a developer writing a one off change script for his modification. Other times there is a DBA or some other singular person responsible for maintaining changes. Either solution takes time and effort and is prone to human error and forgetfulness.

The modeler keeps these two layers in sync by generating an installer project. Moreover it generates the SQL change scripts. Each generation is a marked version internally. On the next generation, the present and past version will be compared and a change script emitted that will be run at the appropriate time. Each change script is marked with a version number made up of the four part build number you have specified in the model and a fifth, internal generation number.

When the installer is run on an existing, versioned database, it checks the database version, which is this five part, version number of the installer last run on it. The installer then calculates what are the remaining change scripts from the last point to the present point and runs them.

Execution Order

There are some databases with a large number of dependencies. Some applications that require numerous views, stored procedures, and functions all with inter-dependencies. By default, the model emits scripts in the order they are entered into the model. This works fine most of the time since people tend to enter objects in the order they need and run them which naturally build a dependency tree correctly. However when you import objects from an existing database en masse, there is a good chance that the dependency tree will not be built correctly if you have a large number of inter-dependencies.

To address this issue the modeller has some built-in tools. First, you can order your objects manually. On the nHydrate Utilities menu, there is an option for Precedence Order. Use this to view a list of all objects in defined, dependent order. You may move objects (views, stored procedures, and functions) up or down the order.

Also notice that there is also an Import button. This functionality allows you to specify an file with an “nOrder” extension if you have one. An nOrder file is an XML key file that orders objects in the model. The installer application will emit one if necessary. When the installer is run it tries to figure out the dependency tree of all objects it contains. If any of the objects are out of order, it will create an nOrder file in the same folder as the installer assembly file. After a successful installation you can check this folder for an nOrder file. If one exists, it can be used on the Precedence Order screen to define a order determined by the installer. Once the model, object order is defined correctly and the installer project is re-generated, you will not see the nOrder file re-created. It is only created if one or more

objects are defined out of order.

Skipping Sections

There are many generated scripts and all of them may not apply to your installation. The installer may have scripts blocked into sections. These sections can be selectively excluded from a particular installation or upgrade on demand. The pre-defined scripts are blocked off automatically. You can also define your own sections. This is a two step process. First you may define a section block. The syntax is as follows in one of the SQL files. The section below is named “MySection”. You may define whatever name you wish to a section. It must have a Begin block and an End block.

Database script section syntax

```
--##SECTION BEGIN [MySection]

--Add some SQL script here

--##SECTION END [MySection]
```

If you wish to exclude this section from the install, simply add it to a string list of skipped sections and pass it into the installer.

```
var skipSections = new List<string>();
skipSections.Add("MySection");

var dbInstaller = new Acme.MyProj.Install.DatabaseInstaller();
dbInstaller.Install(new Acme.MyProj.Install.InstallSetup()
        {
        MasterConnectionString = "...",
        NewDatabaseName = "...",
        ConnectionString = "...",
        SkipSections = skipSections,
        });
```

Multi-Tenant Functionality

Yet another built-in feature is the ability to add multi-tenant functionality. This feature allows you to design applications that may be used by multiple, independent application users that can use the application with total data independence. This contrasts with multi-instance software in that you can have one instance of your application and run many users with data separation. nHydrate implements a shared schema model with each tenant table having an assigned user that owns the data. The owner user is the logged-in database user. Queries against tenant tables only pull back data belonging to the database user that inserted the rows. This is based on the connection string defined when the EF context is created.

nHydrate implements this functionality by modifying the Entity Framework mapping files to route all queries to managed, internal views that pull back data only for the logged-in database user. In this way there is no way to pull back data not associated with the correct user using Entity Framework. This is an implementation of row level ownership. The permission functionality is completely transparent to the API. There is no need to append a user permission where clause to queries as this functionality is built into the framework.

To setup a multi tenant model, simply build a model as usual. On tables that are to be marked for tenant functionality just set the IsTenant property to true. That is the only setting you need to make. Upon generation, each tenant table has a hidden, managed column that stores the database user name. Each time a record is inserted into a tenant table, this column's data is automatically set to the logged-in, database user.

To pull data out of a tenant table, simply write an Entity Framework, LINQ query as normal against the table. Remember there is no need to specify the user permission in the where clause as this is handled for you. All queries are routed through the hidden, managed view that is the filter for the table. In fact it is not possible to pull data directly from the table at all. There is no Entity Framework mapping directly to any tenant table. You can join and manipulate tenant tables normally with other tables (tenant and non-tenant) with complete transparency. All data permission is handled by the framework.

The only caveat is that you need to create a database user for each tenant. You must specify that user in the connection string when you create an EF context. The only management you need to perform is database user management. As long as you connect to the database using the correct connection string, you will never get tenant data created by any other user.

Multi-User Development Environments

Using a model-driven approach is great for project management. It can save tremendous amount time when writing and dealing with data access layers and multi-group development on common models. However there there is a problem with model access. Since the model is saved in a file, it is essentially a single-user model. Source control does not merge XML documents very well. In any case you do not really want a source control system automatically merging something as important as your entire application architecture.

ModelToDisk

To address this issue, the model has a special property called ModelToDisk. When true, this property ensures that model is distributed to disk. Under the folder where the model file resides, a managed folder will be created. Its name will be based on your model file name. So if your model file is “mymodel.nhydrate”, there will be a folder created at this location named “_mymodel.model”. Under this folder will be a separate folder for Entities, Stored Procedures, Views, and Functions. In each folder will be files that define much of the model. These are text/XML files that can be edited outside of the modeller in any text editor if desired.

The Entities folder has XML files for different aspects of each entity. There are multiple files for fields, modules, diagram settings, etc. Most of the properties of an Entity is defined in a well-defined XML format. This allows multiple team members to work with the same model without locking a singular model file. This is very convenient when using source control. Since the model is split into many separate files, there is less chance of collisions when checking-in files.

The other folders hold multiple files per entity type: Views, Functions, and Stored Procedures. Each of these have a SQL property and this is broken into its own file for ease of editing as well. You may open this SQL file in SQL Management Studio or any text edit to make changes if desired. Simply save the file and it will be part of the model. Each one of these object types also has other XML files just like a table Entity that define the other object properties and states in a well-defined, XML format.

In short, this functionality does go a long way toward helping groups collaborate. Many groups have users who want to edit SQL files in a text editor of choice, so this functionality does provide for that scenario. The many files functionality ensures that a very limited number of conflicts occur with source control. The ModelToDisk property turns nHydrate into a group collaboration tool.

Summary

The ORM generator does add a lot of functionality to a project. The model allows you to control all aspects of your framework design. All objects have definite types and there is non confusion about which objects are being handled. All actions like adding, deleting, searching, paging, updating, etc are strongly-typed. A most important feature is that if the model changes in a binary incompatible way, compile-time errors will be created not run-time errors. This is one of the most important features of the tool. If objects are re-typed, added, changed, or deleted in the model, a compile-time error is raised and your code (both generated and hand-written) will not build. The issues can be found immediately without errors going out to the production.

The generated model was also designed for coding efficiency. There are numerous overloads for methods so you can load, save, and transfer data with few lines of code. In many cases, only one line of code is required to perform actions. You can actually load or save data, with where qualifying statements, in one line of code. This is very different from frameworks that require all sorts of objects to be declared and passed-in to other objects, in order to load or save data. There are convenient static methods on objects, so you do not need to declare an object to perform many functions like querying, aggregating, and bulk update. This ensures that very minimal code can be quite functional.
Paging is a difficult task to handle in a generic, programmatic way. The generator handles it well by giving you a strongly-typed way to query, sort, and page data. There is no way to interact (select, order, or page) with objects or fields that are not in the model. This truly alleviates the fear programmers have of making database changes. Now you know the code is not broken.

The model driven development imposed by the tool allows you to manage the database and code base with one tool. They are connected and stay in sync at all times.

However in the end the speed at which you can build software is the greatest benefit. You can literally reverse engineer a database into a model and start building an application in minutes not weeks. Developers should not waste their time synchronizing database schema, DAL, middle-tier objects, UI, etc. They should also be able to change to their framework without fear of code breaking in the production. We should not keep software developers busy building CRUD layer, data access routines, and complex stored procedure logic. The generator allows you to address all of these issues and more. By building less error prone, better organized, code much faster, we can concentrate on the more exciting and complex solutions we want to build.

Conceive, Model, Generate!