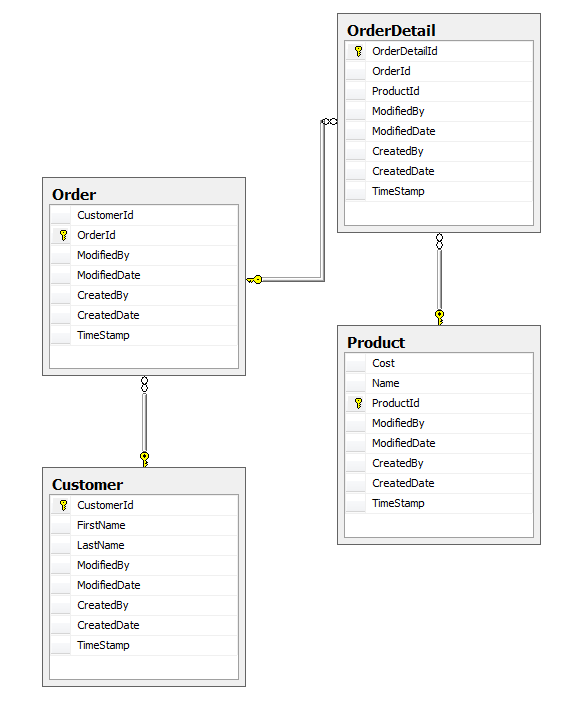
### Introduction

The nHydrate code generation platform is discussed in numerous articles on CodeProject.com but they focus more on what it is and how it works. This article will deep dive into how to use one specific generator: the ADO.NET data access layer. This is a very useful generative template. A lot has been made lately of Microsoft's new foray into generation technologies with Entity Framework. EF is a nice platform but much of the generation is built into the platform, which is the point of course, but I feel it makes customization somewhat cumbersome sometimes (like creating read-only entities). T4 templates come to the rescue in some cases, but it still does not move the model to the solution level, which is where I would like to see it. To find out more on how to download generators and install them see this article. http://bit.ly/f67Yxf

I like to use domain driven design (DDD) also called mode driven development. This includes having all (or just about all) information about your platform defined in a model. Then using the model to create a multitude of projects in your VS.NET solution that are defined by that model. Everything from the database (and database changes) to the API to inversion of control libraries and even to the user interface can be defined by the model. This is the true goal, if somewhat imperfectly realized, of model driven development.

As simple, yet effective, strategy of this is realized with the ADO.NET generator. Once a model is defined you can use it to create a DAL. This is a very mature generator template in use for some years now. It is LINQ supported and stored procedure backed. It uses a factory pattern to create and manipulate objects. The API incorporates lazy loading so you can walk relationships or alternatively use eager loading as well. The syntax is simple and everything about the usage is compiler checked. There are no "magic" strings or numbers sprinkled in your code. If you make breaking model changes such as field renaming or deleting, all you need to do is re-generate and compile and your breaking changes will be found by the compiler. There is very little to worry about. If you change relations or the way entities interact this will cause the relevant classes, methods, or fields to be updated so your existing code will not compile. Breaking changes will be caught by the compiler! This goes back to the model driven development mantra, you have a compiler so use it. If you have magic strings in your code, something is wrong. If you have magic numbers in your code, something is wrong. If you make breaking database changes and your code and it still compiles, something is wrong. There is a better way.

I assume that you have some experience with nHydrate before you read this article and if not there are some other articles on this site that explain the basics of creating a model. In essence a model is a design for organizing data. The nHydrate model can be thought of as a data model with meta data on it. Many people use their database as a model. This works for some simple situations. However you will not be able to create multiple layers using this methodology unless you hand code it. nHydrate tries to remove much of the mundane code writing.



Let's assume we have a model and we have added a few tables. The tables will be Customer, Order, OrderDetails, and Product. The relations will be straight forward in that Customer will have many Orders, an Order will have many OrderDetails, and each detail will have an assigned Product. Now after generating the API and database project we can start to write code using out new API.

## Insert Objects

First lets add a customer. We use the factory pattern to do this so we will need a collection that will create a new object on which we can populate settings and then add it to the collection.

|  |
| --- |
| //Insert a customer var customerCollection = new CustomerCollection(); var customer = customerCollection.NewItem(); customer.FirstName = "John"; customer.LastName = "Doe"; customerCollection.AddItem(customer); customerCollection.Persist(); |

In the code snippet above, a new Customer collection is created and a new customer object is requested from it. The new object is not part of any collection at this point. The NewItem method is static and returns a free standing object which you may choose to add back to the collection to choose to ignore it and never use it at all. I have set the required properties and then added it back to the collection. At this point all operations have been performed in memory. Only when I call the Persist method on the collection object does a database call occur. You may add any number of items to the collection which effectively queues the objects for later save and then saves all of them in one database transaction. You can actually save an entire object graph and add objects in any order. The dependency engine built into the framework ill determine which objects get inserted first based on dependency. There is no need for you to manually track this.

Now the more advanced stuff is not that much harder. Assuming we wish to keep all of our objects, no matter the type, in the same container and the same database transaction, we can use the following syntax to add a customer and a related order.

|  |
| --- |
| //Insert a customer var customerCollection = new CustomerCollection(); var customer = customerCollection.NewItem(); customer.FirstName = "John"; customer.LastName = "Doe"; customerCollection.AddItem(customer);   //Get a reference to the order collection in the same subdoamin and add an Order var orderCollection = customerCollection.SubDomain.GetCollection<OrderCollection>(); var order = orderCollection.NewItem(); order.CustomerItem = customer;  order.TotalCost = 100; orderCollection.AddItem(order);   customerCollection.Persist(); |

Notice that the top code is the same. This code snippet creates a Customer then add a related Order. Notice that the OrderCollection is not created anew. It is retrieved from the customer collection's container so it will be inside of the same container called a Subdomain. A Subdomain is an in-memory container of an object graph. When the persist method is called, the entire object graph is saved in one database transaction. You could of course create as many new collections as you wish, but they would all access the database independently. Using the methodology we have defined above, the new Order has access to the new Customer. Neither of these objects are in the database yet. Also keep in mind that the primary keys for both objects are database generated or identities. So we do not know what they are at this point. Since we have linked the Order object with the Customer object this does not matter. Let the framework do the work. These objects will be saved in the same database transaction and all object will save or failed together. There will be no partial saves.

When the save operation returns the objects will have the real primary keys (and all other data) that exists in the database. So assuming these are the first objects saved the customer will have an ID of 1 and the Order will have an ID of 1 too since both are identities. The important piece is that the Order object will have a foreign customer key tied back to the proper Customer object. You can follow this pattern to add OrderDetails, etc.

## Select Data

Now let's retrieve some existing data. There are generated stored procedures for select all queries. All stored procedures are generated based on the model as well and part of the database installer project. All stored procedures and dynamic SQL are issued to the database engine are parameterized so there is no worry of SQL injection attacks. There are also stored procedures for all select by primary keys and foreign keys. All update and delete actions will go through stored procedures as well. All LINQ queries will necessarily generate dynamic SQL queries due to their dynamic nature; however they are compile-time checked.

The first thing to understand is that the static RunSelect method is used to select data from any collection. The following code snippet shows how to get all objects and a single customer by primary key. There is a stored procedure run in the background and no dynamic SQL is used.

|  |
| --- |
| //Select all customers var customerCollection = CustomerCollection.RunSelect();  //Select a single customer by primary key var customer = CustomerCollection.SelectUsingPK(1); |

That is a very simple example. More likely than not you will need to select data based on very dynamic situations. The next code snippet selects data based on some lambda statement.

|  |
| --- |
| //Select all customers where name is John var customerCollection = CustomerCollection.RunSelect(x => x.FirstName == "John"); |

Again this is a rather simple scenario. Many times you with to pull data based on data or conditions in secondary tables. In SQL you do this with inner joins and with other ORM tools you need to know the structure of the data to issue join criteria. Using this DAL template you do not need to know too much about the model. Intellisense provides you with what you need write code. In the following example I will select customers based on data in a related table; however notice that I specify no join information. The developer does not know the exact field or fields on which the join is based. There is no need to know this. The model contains all of this information and the code is based on the model. There is no reason for the developer to again specify this information in code.

|  |
| --- |
| //Select all customers that have an order where the total cost is more than $100 var customerCollection = CustomerCollection.RunSelect(x => x.Order.TotalCost > 100); |

The predefined lambda syntax provides a way to walk the relation in a compile-time checked manner without specifying any metadata information about how to actually pull the data. Keep in mind that if I remove the relationship between Customer and Order in my model, this code will not compile. The compiler knows the relations of my model!

### Paging

Another important aspect of data querying is paging. This is a task that has not been very straightforward in many data access layers. Using this template it becomes quite easy. You simply create and use a custom paging object. Each entity type has its own paging object generated just for it so they are all strongly-typed. In the code snippet below I have created a customer paging object to get page 2 with 10 records per page. I then call the static RunSelect method with a lambda just as before except this time I passed in the paging object. This is all that is required to page through a result set. After the method call, the paging object has a property RecordCount that holds the total number of non-paged records. You may use this populate your pager UI.

|  |
| --- |
| //Seetup a paging object and run a query CustomerPaging paging =   new CustomerPaging(2, 10, Customer.FieldNameConstants.CustomerId, true); var list = CustomerCollection.RunSelect(x => x.LastName == "Jones", paging); System.Diagnostics.Debug.WriteLine("Total Records: " + paging.RecordCount); |

## Dependency Walking

After you have selected an object, you will of course want information related to that object. In the model we are using, it would be natural to have a customer and walk to his orders and then walk to each order's details. We may even want to see the product information related to each order detail. Again notice that no join information was specified. Also all 1:M relations have an item list representation and walking back up to the parent is performed with a singular item.

|  |
| --- |
| //Select a single Customer by primary key var customer = CustomerCollection.SelectUsingPK(1);   //Walk each order for this Customer foreach (Order order in customer.OrderList) {  //Wriet out the total cost of the Order  System.Diagnostics.Debug.WriteLine(order.TotalCost);    foreach (OrderDetail orderDetail in order.OrderDetailList)  {  //Write out the OrderDetail's product anme  System.Diagnostics.Debug.WriteLine(orderDetail.ProductItem.Name);  }   } |

## Bulk Operations

You can perform operations in bulk as well. This includes issuing database statements in a single database transaction. The entire action will fail or succeed as a unit with no partial saves. Deletes and updates can be performed this way. The syntax is a simple lambda statement again. Using a lambda statement means that the compiler will check the syntax of the statement and there is no fear that you will get a run-time error because of misspelled fields or entities. The first lambda is very simple but of course it can be as complicated as you wish to make it. You can even walk relations to other tables to delete data based on complex joins.

|  |
| --- |
| //Delete all customers with the first name John  CustomerCollection.DeleteData(x => x.FirstName == "John");  //Delete all customers with the first name John and  //with 1 or more Orders of less than $100 CustomerCollection.DeleteData(x =>   x.FirstName == "John" &&   x.Order.TotalCost < 100); |

To update data it is much the same. Simply issue a lambda and the field you wish to update. This only works for single field updates at present. However the data types are all strongly-typed. In this example we are setting a string property, so the third parameter is a string. If we were updating an field of type integer, the setting field would be an integer too. In other words the compiler is checking our code again to remove run-time errors.

|  |
| --- |
| //Update all customers with first name John to Dave CustomerCollection.UpdateData(x => x.FirstName, x => x.FirstName == "John", "Dave"); |

## Objects Events

Each object has events built-in that allow you to build callback mechanisms. Each property on each object has a PropertyChanging and PropertyChanged event. These are actually based on standard .NET interfaces of INotifyPropertyChanged and INotifyPropertyChanging. You can bind these to any component that supports these interfaces or hook the events yourself and perform some action. In the code snippet below, a customer object is retrieved from store and one of its events is captured. In the event handler, you can check which property is changing and perform some action. Notice that I did not compare the a string literal to find the property name. I used the generated enumeration for fields on the Customer object. This enumeration is used for a variety of functionality points to retrieve meta data like field length from the generated Customer object.

|  |
| --- |
| //Select a single Customer by primary key var customer = CustomerCollection.SelectUsingPK(1); customer.PropertyChanging +=   new PropertyChangingEventHandler(customer\_PropertyChanging);  void customer\_PropertyChanging(object sender, PropertyChangingEventArgs e) {  if (e.PropertyName == Customer.FieldNameConstants.FirstName.ToString())  {  //Perform some UI action  } } |

There are specific change events for all properties as well. On the model, you can set the EnableCustomChangeEvents setting to true to see this functionality. A change and changing event is generated for each property for every object. Now you can write much more specific code if you are tracking changes. In the following code snippet I have selected a Customer object and handled its NameChanging event. Each time the Name property is set, this event handler will be raised. I check the new value to be set and if it is "Dave", I cancel the setting. When I cancel the setter no error is raised but the new value is ignored and the previous value of the property remains. If the object was marked as unchanged (as opposed to dirty for saving) it remains that way.

|  |
| --- |
| //Select a single Customer by primary key var customer = CustomerCollection.SelectUsingPK(1); customer.FirstNameChanging +=   new EventHandler<BusinessObjectCancelEventArgs<string>>(FirstNameChanging);  void FirstNameChanging(object sender, BusinessObjectCancelEventArgs<string> e) {  if (e.NewValue == "Dave")  e.Cancel = true; } |

### Partial Classes

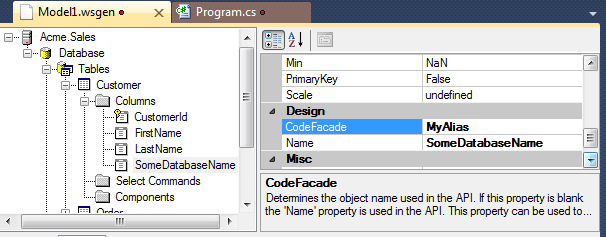
All generated items are created as partial classes. There is a generate once class, which you may modify since it will never be overwritten. There is also a generate always class, which will always be overwritten. This very much like the what VS.NET does with a Windows Form class. There is a designer that is overwritten each time you make a changes in the visual designer and a code class where you add your custom code for the form.

The Customer object has a FirstName and LastName property. You property need to display a full name in your UI so create a custom property in the gen-once file. There is no need to concatenate strings in your UI code. Simply expand your generated Customer class. The stub of this class is generated. I have simply added a read-only property called FullName. Intellisense sees this as just another property. You can use this anywhere that all other properties are used. There is no way to distinguish that it is hand written as opposed to generated other than it is in the gen-once file.

|  |
| --- |
| partial class Customer  {  public string FullName  {  get { return this.FirstName + " " + this.LastName; }  }  } |

### Code Façade

Many times you want your code to look differently from your database. Many people have to work with legacy databases that have table prefixes and ugly field names. For example, a table structure in a database might be TBLCustomer, TBLOrder, etc. This makes for very ugly C# code. There are similar problems for field names as well. Using table and field code façades, you can have pretty code without changing your database structure. In the following graphic you see an nHydrate model . There is a Customer entity with a field named "SomeDatabaseField". This is the actual name in the database. However notice that the codefacade property is set to "MyAlias". This is what will show up in code. So in code you have no knowledge to what physical database field you are mapping.



|  |
| --- |
| //Select a single Customer by primary key var customer = CustomerCollection.SelectUsingPK(1); customer.MyAlias = "MyValue"; customer.Persist(); |

### Relations

Not only are parent child, one-to-many relations supported, but one-to-one, and many-to-many are as well. If you define a relation between two entities any the linking fields are both marked unique in their respective tables, the relation is by definition a one-to-one relation. The generated code will reflect this in that the parent entity will have a child item not a child list and each child item will have a reference to a parent item. Both relationship walks are singular not plural.

A many-to-many relation is supported too. Define two tables like Product and Feature and assume one product can have many features and one feature can be associated with many products. To define this relation we create an intermediary table called ProductFeature and add the primary keys from both the Product and Feature table. Afterwards mark this intermediary table as associative in the model. That is it. The database script will be generated to create all three tables; however you will never see the middle table in code. In code each Product item will have a FeatureList and each Feature item will have a ProductList. There is no other special mapping or handlers to write.

### Query Plans

The architecture of the generated code works well with SQL caching. The generated code uses two distinct querying methods in the background. The first method is a façade over generated stored procedures. This method produces extremely very fast results because the query plan in SQL Server is cached. All (2-N) calls to the stored procedure execute about as fast as SQL can possibly perform the action. Using the generated methods will give your maximum performance because of SQL Server's optimizations and caching.

The second method is a parameterized query. This method is employed anytime a LINQ statement is written against the API. In the background a parameterized SQL statement is generated. This has the same benefits as the stored procedure. If you run the same LINQ statement again, the query plan is cached by SQL Server. The caveat is that since LINQ is more free form, you can create a great variety of queries. This attribute executes better in a real-world application than in theory, since applications do not normally issues thousands of distinct queries but issue the same query with differing parameters.

### Transactions and concurrency

All selections and updates are atomic and made inside of a SQL Server transaction. When you load an object, a collection of objects, or multiple collections, these items will exist inside of a subdomain. This is a container that holds all related information. You can have any number of subdomains loaded. They will not interfere with each other and have no knowledge of each other. Every object exists inside of a strongly-typed parent collection object. All collection objects exist inside of a subdomain container. This is implied. Even when you load one object, it already has a parent collection and it a parent subdomain.

This comes into play when objects are persisted. When the Persist method of a collection is called, its entire subdomain is persisted in one SQL transaction.

### Summary

There is of course much more functionality in the API. This is a good sampler of what you can easily do with it. The basics of creating, updating, and deleting data as well as selecting were covered here and you can see it is very easy to perform any action. All objects are strongly-typed and generated. You can define custom view and stored procedure objects that map to their counterparts in the database. Using the nHydrate platform also has another benefit in that you can generate multiple APIs and hit the same database. Using different templates you can create an ADO.NET, Entity Framework, NHibernate, Code First, or any other API and have all code generated from a single model. All API assemblies will necessarily stay in sync with the database.

When this API is generated in conjunction with the database installer, your database will always stay in sync and be versioned with your API. This can save countless hours of writing SQL upgrade scripts and knowing when to run them. Since the API is versioned with the database, there is never any ambiguity of whether the API will have errors if something is missing from the database.