Starting nHydrate

This is very much a work in progress. Many items have not been addressed and there are “TODO” stubs in some places. This is a constantly changing framework and writing documentation to keep up to date is a challenge. This document changes frequently so check the download location for updates. Many of the sections in this document are really stubs in that there is not much elaboration on the subject. In the future many of the sections will be expanded with more code samples and more verbose explanations.

November 29, 2010

# Part I Chapter 1 - Overview

## The need

There is a great need for model driven development today. Applications have become so complex and all encompassing that hand coding them is unwieldy and error prone. Developers get to the point where they are afraid to make necessary changes for fear of what that might break. Applications become big band-aids and after a while they are scrapped and restarted.

With a model driven methodology, you can in a model your application and generate much of its infrastructure. This allows you the flexibility to make changes to a model and re-generate. All of CRUD layer, service code, and various other pieces are automatically in sync with your model. There is no fear that small changes will break your framework in many unknown places. Also with model driven development it is possible to compile-time check all or almost all aspects of your application. This is true power in that you only need to concern yourself with logic afterwards. The infrastructure core is handled by the various generators that directly use your model. One model equals a lot of synchronized code.

## nHydrate Features

There are a lot of features right out of the box. Performance, compile-time checking, model driven development, multiple synchronized assemblies, pagination, LINQ, validation, dependency walking, Visual Studio integration, and much more allow you to start developing much more code and a lot less errors.

### 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 criterion 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.

### Built-in web service serialization

The built-in web service allows you to allow clients to interact with your database without knowing any database connection information. This also allows you to separate your clients and server across the Internet. All information is compressed as a further optimization.

### 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 with the Microsoft “InstallUtil” tool or plug it directly into a larger installation application.

### Two persistence paradigms

You can persist using a direct database connection or use the web service to remotely synchronize data.

### Typed Lists

You can define Type tables that generate 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

Singular and plural names of entities of lists are built-in. By simply looking at your code, it is obvious what items are single objects and which are list of objects.

### 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 pre-fetch functionality 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 result set 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!

## Model Driven Engineering

Model-driven engineering (MDE) is a software development methodology which focuses on creating models, or abstractions, more close to some particular domain concepts rather than computing (or algorithmic) concepts. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system.

A modeling paradigm for MDE is considered effective if its models make sense from the point of view of the user and can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, and members of the development team. As the models approach completion, they enable the development of software and systems.

As it pertains to software development, model-driven engineering refers to a range of development approaches that are based on the use of software modeling as a primary form of expression. Sometimes models are constructed to a certain level of detail, and then code is written by hand in a separate step. Sometimes complete models are built including executable actions. Code can be generated from the models, ranging from system skeletons to complete, deployable products. With the introduction of the Unified Modeling Language (UML), MDE has become very popular today with a wide body of practitioners and supporting tools. More advanced types of MDE have expanded to permit industry standards which allow for consistent application and results. The continued evolution of MDE has added an increased focus on architecture and automation.

## ORM Mapping

Object-relational mapping (ORM, O/RM, and O/R mapping) in computer software is a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.

## 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.

# Chapter 2 - Projects

## Overview

When a model is generated, various projects are created in VS.NET. The first project generated is the core DAL. It contains all the objects and collections for your database. Then the DTO layer is created. This has a mirror of the DAL but with lightweight objects with integrated serialization functionality to query over the wire. The third project is the DTO mapping library. This is used by the data service to map the DAL to the DTO layer. The fourth project is the data service itself. This service is used to process DTO containers sent over the wire. The fifth project is the unit tests. These tests can create a database and insert, update and delete information to ensure that your model has referential integrity. The final project is the installation library. You can run this with the .NET installation utility or include it in a larger application to provide installation capability to users.

## Data Access Layer

The DAL is the core of the system. It contains the concrete classes of the model. These classes can be loaded and saved to the database, as well has having inheritance hierarchies. They may be queried with LINQ syntax and everything about them is strongly-typed. This is the only generated library needed on the client, for database access. It alone is the gatekeeper for all data access. It implements the entities and logic for loading, saving, inheritance, etc. This library is extendable in that all classes are generated as partial classes into two distinct files. There is a user file that is generated once and will not be overwritten and a machine file that is managed by the generator and will be overwritten. There is no need to keep track of regions that the generator will overwrite or protect, like many ORM applications. This philosophy is messy and error-prone. Your custom code will never be overwritten. It is compiled directly into the related entity and there is no way for a user of the compiled library to know the difference between custom and generated code.

## Data Transfer Layer

The data transfer layer is a set of POCO objects that allow for native JSON and XML objects to be serialized across the wire. This can be used in the REST interface provided by the data service. These objects can be used as property containers to pass to façade layers or other layers of your architecture where direct database access is not necessary or restricted. They are used inside of the generated framework as the base for the REST-ful service.

## Data Service Project

The data server is a generated web service that can be used to interface with a database across the wire. There is no need to add any custom code to get it running. This is a REST-ful service and can expose objects directly to script on a web page with JSON and XML objects.

## Service Interfaces

This assembly holds the base interfaces for all Inversion of Control (IoC) patterned projects. It holds not classes or functionality but the interfaces that dictate the way in that all services communicate together.

## IOC Extensions

This assembly provides the IoC functionality. It consists of the way that generated objects can be loaded and persisted. It is actually has very little real functionality but it merely decides how to perform the requested action. The real functionality is in the proxy assemblies like DAL, WCF and Mock proxy. The IoC extensions assembly determines which one to use and shuttles the action requests to the appropriate library.

## DAL Proxy (DTO Extensions)

The extensions library maps the DTO layer to the DAL layer. It is used by the various generated services to perform bidirectional storage and retrieval operations with a REST-ful interface or other service to/from the DAL. This library is used on the server in conjunction with the DAL and the server copy of the DTO. It provides an extension library to extend the functionality of the DTO layer. DTOs by definition have no functionality. The DTOs are only objects with properties. Instead of writing the entire load, save, and translation functionality by hand, simply use this DLL to provide functionality to the DTO library through extension methods, much like LINQ’s implementation.

## WCF Proxy

The WCF proxy assembly allows you to connect to the remote, generated WCF service. If your client application does not have a direct connection with, you can connect to it using WCF. All of the specifics are generated and hidden from you. Simply configure the client to use this proxy assembly and communication starts working with no code.

## Mock Proxy

If you are using test driven development (TDD), you will need to perform some mocking to create standardized unit tests. The mock proxy gives you the power to simply configure your client application different and it is talking to a mock and not a real database. This is an invaluable tool when implementing an IoC mocking layer.

## WCF Service

This generated service allows you to send requests for data and persist changes to a WCF service that works seamlessly with all of your generated objects. There is no need to write a remote service to interact with data. This service provides virtually all the functionality you need. It is extensible with partial classes (like all of the generated projects), so you can extend the services functionality in a custom way if need be.

## Installation Project

The installation project contains all the code to create a new database or update an existing one. When a new database is created, all tables, stored procedures, indexes, relationships, and static data are created. The database is ready to be used by the DAL. Each database is versioned upon creation or modification. The installation application determines the current version of the database and upgrades it to the newest version. This allows you to keep a full record of all database changes so production can be updated with a new version automatically.

This project is compiled into a library not an application. The library can be run directly from the .NET environment by associating it with .NET tool InstallUtil.exe, or incorporated into a larger application. Running in the .NET tool means that you can execute your database changes while developing without ever leaving the .NET environment. Incorporation in a larger application usually means a custom installation application, but it can be any application really.

When you make model changes and regenerate your code, update scripts are created that will transform the previous database format into the new database format. You may also add your custom scripts to the provided versioned SQL files that will only be run when updating from version X to version X+1. Custom scripts can be added and scheduled to run before or after an update. Custom static data scripts can be added to the static data defined within the model. You can add any arbitrary script to the library and it will be executed. This provides a lot of flexibility to your deployment process.

It is important to note that all generated stored procedures are compiled into this library as well. You may add custom stored procedures, views, etc if you wish. You can truly treat this library as you database upgrade path. Since all of the scripts are embedded in to the library it could get quite large. To combat this issue, the library can natively read ZIP files. Simply create a ZIP file with any number of SQL files in it and they will be executed. This is a nice feature for those who have very large static script files like zip codes, company directories, etc. There is nothing special to do. Simply embed a ZIP file just like any other SQL file and all files will be executed in alphabetical order at the same folder level.

# Chapter 3 - The Model

## Import Model

When starting a nHydrate project, you can import a model from an existing SQL Server database or create one from scratch. We will import a model for simplicity. Importing allows you to get a working model very quickly. We will import the Northwind database.

In VS.NET create a blank solution. After you have a solution, right-click in the project explorer and choose “Add New Item” from the context menu. Choose the text file template and name it anything but change the extension to WSGEN. This will create a blank model in your solution and open it in the nHydrate designer.

Now click on the model and you will see a properties window. For this example, we will change the company name to Acme and the project and database name to TestProj. These settings will be used to create your projects and corresponding assemblies.

On the Tools menu you will three new menus: Generate, Import, and Verify. Choose Import and enter the database connection information to the Northwind database. When complete you will have a model based on this database.

Now for the sake of consistency, code beauty and best practice, we will change the names of the tables and columns. The generator engine uses the names to create objects, properties, methods, etc of course. To make pretty Pascal names in code, we need to format the name in a certain fashion. Since this is an existing database, we will not rename the actual tables and columns but add code facades to them. The database already exists and has data in it so we will map our code to differently named objects for ease of use in our C# code.

In the model designer, select the “Tables” node in the tree and right-click. Choose the “Update Table Codefacades” menu option. This will display an options screen. Use the default settings and press OK. You will notice that the grid of tables displayed now has the code façade column populated. Some tables are singular and some plural. It is recommended as a best practice to make all tables singular. This is because in code objects have a suffix of “Item” and groups of objects are named with “Collection” or “List”. We need to go to each table and manually change the name to singular. For example, change “Shippers” to “Shipper”. Now select the Tables node in the tree again, right-click and choose the “Update Column CodeFacades“menu option. Again an options screen is displayed. Use the default settings and press OK. This will add a code façade to all columns. It will format column names like “CustomerID” to “customer\_id”. The generator uses the underscore to build Pascal names in code like “CustomerId”. Now your code will be easy to read. Keep in mind we are only using code facades because this is an existing database. If we were starting from scratch, it would be best practice simply to name the columns with names formatted in this fashion. The façade functionality allows you to map a database field to a different name in code.

## Validation

There are numerous validation conditions that are tested. The model designer validates a model before it can be generated. This also helps you in application development since using just SQL Server and VS.NET you can create quite nonsensical graphs. The validation ensures that some error conditions never occur and some bad relationships are not created.

## Root Node

The root node of the designer allows you to define global settings of the model like company name and project name. These are settings that pertain to the model as a whole.

## Company Name

The company name is used to identity generated projects. All generated project are named according to the pattern “Company.Project.Function”. So the company name you use will be the first namespace for all projects.

## Copyright

You may wish to a have a copyright notice at the top of all your generated files. Enter the notice here and all C# files and SQL scripts will have this notice prepended to the top of the file.

## Project Name

This is the name of the project that the model represents. For example when importing the Northwind database, you will be creating model that allows you to build a Northwind application. In this case the name “Northwind” is quite appropriate. Keep in mind that the project name is used for all assemblies and namespaces.

## Use UTC Time

The setting defines whether all times are saved as UTC. When true the time zone of the client is used to convert the created date and modified date audit fields into UTC time before saving. This feature can be used for disparate clients and a server in a different time zone. This will ensures that the created date and modified date are universal no matter which client is using the generated API.

## Version

The version is used to version a database. As time progresses, you will need to modify your database schema by adding and deleting tables and fields. The version property allows you to set a version number and later increment it so the upgrade scripts know how to run. This allows you to create upgrade scripts that only run on version 1 databases but not on version 2 and so forth.

## Database

The database node contains the properties needed to define the database. There are very few properties to be set. There is rarely a need to change the defaults. This is where you may set the names of the created, modified, and timestamp audit fields.

## Created By Column Name

The created by audit field has a name by default, but you can change it if you wish. Set this name to any value token to rename this audit field to a custom value.

## Created Date Column Name

The created date audit field has a name by default, but you can change it if you wish. Set this name to any value token to rename this audit field to a custom value.

## Grant Exec User

This is an optional value that is used to grant executable rights to a specific user in the generated stored procedures. If you do not have any special permissions setup in your database, there is no need to set this property.

## Modified By Column Name

The modified by audit field has a name by default, but you can change it if you wish. Set this name to any value token to rename this audit field to a custom value.

## Modified Date Column Name

The modified date audit field has a name by default, but you can change it if you wish. Set this name to any value token to rename this audit field to a custom value.

## Timestamp Column Name

The timestamp audit field has a name by default, but you can change it if you wish. Set this name to any value token to rename this audit field to a custom value.

## Tables

This is the list of table that make up the database as well as the primary objects from which all generated APIs are created.

## Allow Audit Tracking

This setting allows a shadow table to be maintained that keeps all actions on a specific table. When true a table is created that tracks all inserts, updates, and delete from its master table. Any action performed on the database is recorded in this shadow table for later inspection.

## AllowCreateAudit, AllowModifiedAudit, and AllowTimestamp

These setting determine the audit functionality of a table. When true, extra fields implicitly become part of the table. There is no reason to add these fields (created by, created date, modified by, modified date, and timestamp) manually to the model. They are just part of the table and maintained by the system. The created fields will track the date an object was created and the entity that created it. The modified fields track the last date an object was updated and the entity that did so. The Timestamp field is used for concurrency. When an object is persisted to the database, its timestamp field is checked to ensure that data has not been changed since the load. This allows much optimized database functionality and optimistic concurrency. Database rows do not need to be locked while users are updating data.

## Associative Table

An associative table is an intermediary table that allows for many-to-many relationships. This type of table is invisible in the generated code as its only function is to map objects from table to objects in another table. This table type must be the child of exactly two relationships from other tables.

An example of this functionality would be to map customers to products. If a customer can have many products and a single product can be associated with many customers, we have a many-to-many relationship. To model this scenario we would create a CustomerProduct table and make it associative. Then we would add the primary key from both Customer and Product tables. This combination would become a composite primary key in the associative table. Now we define a relationship from Customer to CustomerProduct based on customer id and a relationship from Product to CustomerProduct based on product id. The generated code would like the intermediary associative table.

In the code snippet below notice that a customer and a product are loaded. Afterwards we loop through the product list of a customer and a customer list of a product. You do not see the intermediary table. It is completely hidden from view. It looks like the customer has a list of products and a product has a list of customers.

|  |
| --- |
| //Loop through the product objects of a customer  Customer customer = Customer.SelectUsingPK(1);  foreach (Product p in customer.ProductList)  {  Debug.WriteLine(p.Name);  }  //Loop through the customer objects of a product  Product product = Product.SelectUsingPK(1);  foreach (Customer c in product.CustomerList)  {  Debug.WriteLine(c.FirstName);  } |

## Description

Adding a description makes your generated code more useful. This information is used to create Intellisense tooltips, giving your API more professional functionality. When deploying your generated API the included XML file will provide developers with useful Intellisense they can use to increase their productivity.

## Generated

This property determines if the table is actually generated into code. While most of the time if you define a table in the model you will want to generate it, there are time you may not. For example, if you import a legacy database that is poorly constructed, there may be tables that you wish to exclude. These tables can be in your model so you have a complete model of the database, but you do not wish to use them. Another example is could be tables with hundreds of fields. There are some databases that were constructed with far too many fields. It may be a valid business decision to exclude these monster tables from your generation.

## Immutable

An Immutable table cannot be modified. Its respective collection has no addition or deletion methods. Objects can be selected from the database but all of its properties are read-only. In other words an immutable table is truly a read-only container of read-only objects. This functionality can be used to select data that should never change, for example type information.

## Is Type Table

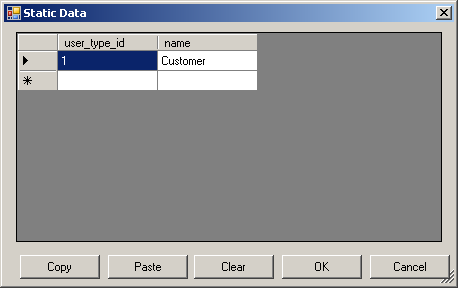
Closely related to immutable tables are type tables. In fact a type table is immutable but must define some static data. A type table allows you to define static values in the database and map them to an enumeration in code. This eliminates the need for “magic numbers” and makes code much more readable. The use of static data in type tables is explained in more detail in the Static Data section.

## Parent Table

The ParentTable property allows you to define an inheritance hierarchy in code and persist this design to your database. There are numerous schemes for mapping objects to a relational database. nHydrate takes the approach that all details should be hidden and you should not have to define any extra data to define this relationship. The only requirements to use this functionality are to define the parent table and define a relationship from the parent to the child based on primary key. The primary key of the child table must the same name, type, and cardinality as the parent table. You can define an arbitrarily deep inheritance hierarchy. In other words table1 can inherit from table2 which inherits from table3, etc.

The functionality is very useful for defining types of objects. For example, you may have a Person table and then have two tables Customer and Employee that inherit from it. Assuming that Person object has a first name and last name defined. The Customer and Employee objects will also have these properties. There is no need to define properties again on each of these child objects. In code you can actually cast a Customer or Employee to a Person object. This is true inheritance.

## Static Data

Another great feature is static data. This is especially useful for type types; however you can add static data to any table. When adding static data, you see data entry dialog. It is just a simple grid that allows you to add the values necessary to populate the table. You see the column names and an entry grid. Values can also be pasted from Excel. If you copy a spreadsheet with the proper number of columns, it will paste directly into the grid. When the installer project is generated, a static data SQL file is created. It holds all the data that must be inserted into each table. This SQL file is run after the create schema SQL file. It checks based on primary key is the data is already in the table and if not inserts it.

For type tables the functionality is extended. Not only will the data be added to the database but an enumeration will be created that maps the values to code. This is very useful in eliminating “magic numbers”. Instead of setting a number in code that maps to a database value, you can use the enumeration. The code below shows three ways to set the UserId field of a Customer object. The first way is very bad and used by many people. The second shows how to use the enumeration for better code readability. The third uses a convenience property that is generated based on relationships to type tables.

|  |
| --- |
| Customer customer = Customer.SelectUsingPK(1);  //Magic Number  customer.UserTypeId = 3;  //Use Enumeration  customer.UserTypeId = (int)UserTypeConstants.Customer;  //Even better use convenience property  customer.UserType = UserTypeConstants.Customer; |

## Code Façade

The CodeFacade property allows for a separation of code from database. This property allows you to specify a completely different name for your code than is used by your database. This functionality is especially useful when dealing with legacy databases in which you cannot change the names of tables. If tables are poorly named in a database, you can still have pretty code by specify this mapped name. All references in code will have this name and will map to the database table implicitly. There is no other action you need to take to perform this mapping.

For example suppose you have a database with a table named “tbl\_Customers”. This will make a very ugly name in code as the class type will be “TblCustomers”. You can specify a table codefacade of “customer” and the type in code will now be “Customer”, so your code would change like the example below.

|  |
| --- |
| //Old Name TblCustomerCollection myList1 = TblCustomerCollection.RunSelect(); TblCustomer o1 = myList1[0];  //New Name CustomerCollection myList2 = CustomerCollection.RunSelect(); Customer o2 = myList2[0]; |

## Name

This is the name of the table. This name is used to generate SQL scripts to create or update the table in the database. It is also the basis of the name used in code if there is no codefacade property set. The generated code does not use this name exactly. It performs a transformation on it in the following ways. All letters following an underscore are capitalized. All underscores are removed. The first letter is capitalized. This means in code, the name used is formatted like these examples. The formatting applies to tables, columns, and any name of a generated item.

|  |
| --- |
| Customer = Customer customer\_product = CustomerProduct customer\_id = CustomerId my\_first\_field = MyFirstField |

## Relationships

The relationships collection defines the relation from its containing table to a child table. The relation items in the collection allow you to define one or more columns that define a connection between two tables. Self-referencing relationships are supported so a relationship may relate a table to itself. These objects correspond one-to-one with database relationships. If more than one relation exists from a parent table to a child table, it must have a role name to distinguish it from the other relations.

# Columns

Colum objects correspond to the database table fields. Many of the properties are familiar but some are metadata that defines behaviors for the generator.

## Allow Null

This setting determines if the field can allow a null value. When true the database is marked to allow null values and the generated code will also allow null values. When this property is false null values are not allows in either. There is an issue in that strings in C# always allow nulls. This is addressed by checking the value in the property setter and raising an exception if it is null. This allows you to get an error in code and not when you persist to the database.

## Computed Column

The property allows you to create a column that is based on a formula. This is a great way to define read-only fields derived from complex formulas.

## Collate

The collate property allows you to specify a database collation for specific columns. When blank the default database collation is used. However if you are using custom collations, this property can be used to define them such that the information is coded into the database generation scripts. This information is necessary for some internationalization scenarios, case-sensitive databases, etc.

## Default

The default property of a column can be used to define a value that will pre-populate a field when the object is created. This field value will persist to the database if no other value is set. Of course number must be used for numeric types and a string can hold any value. There are some more advanced rules for dates. Date fields can define an explicit date value, but also a simple formula as follows.

|  |
| --- |
| getdate(the current date) getdate+1-day (current date plus 1 day) getdate+4-month (current date plus 4 day) getdate+3-year (current date plus 3 year) |

Description  
Adding a description makes your generated code more useful. This information is used to create Intellisense tooltips, giving your API more professional functionality. When deploying your generated API the included XML file will provide developers with useful Intellisense they can use to increase their productivity.

## Formula

Used in conjunction with the calculated column property, this defines a formula on which the field value is derived. A calculated field is read-only as its value is based on this formula.

## Generated

Setting a field to not be generated may be useful if you do not want to push out changes at present. The generated property allows you to selectively turn off generation for specific fields. When false these fields will not be a part of the SQL scripts that create the database and will not be part of any generated code.

## Identity

The Identity property allows you to define auto-incremented fields. Supported data types include integer types (int, bigint, etc) and unique identifiers (GUIDs). When an integer field is marked as an identity, the generated SQL ensures that there is an identity constraint on that field in the database. As records are added to the table, the field’s value is controlled by the database. There is no need to set it. GUIDs are the same concept. If you mark a unique identifier as an identity, a new GUID will be used to populate the field when a new record is created. There is no need to manually set it.

## Is Indexed

There are times when you need to add an index to the database. The IsIndexed property allows you to define fields that need an index and manages it for you. If there are fields that you know will be used in searches quite often, simply mark this property as true to improve search speed.

## Is Searchable

If you need a custom select method to exist on a table for a specific field, set the isSearchable property to true to get it. A method named “SearchBy[FieldName]” will be generated as well as the necessary stored procedure and infrastructure to pull the data. All searchable fields are necessarily indexed as well. In the code snippet below you can see that the FirstName property of the Customer table has been marked as searchable. Now there is a static method named “SearchByFirstName” on the CustomerCollection class. Enter the value on which to search and a collection is returned with all objects that match the criteria.

|  |
| --- |
| CustomerCollection customerCollection =  CustomerCollection.SelectByFirstName("Chris"); |

## Is Unique

All primary keys are by definition unique; however you may also wish to mark non-primary keys are unique. In this instance use the IsUnqiue settings. When true, no duplicate data can be inserted into the marked field. An example of this might be email. In your customer table, you would have a primary key of CustomerId. In addition, you may have a business rule that declares that all email addresses must be unique. Simple mark the email field in the Customer table as unique and this business rule is satisfied.

## Length

This property defines the length of the field. This property does not apply to data types that have a predefined value. For example an integer is always four bytes, so this field cannot be modified. However for a varchar (and other string types), you can define the maximum length of the field.

Added functionality related to this setting is the length checking of the API. The generated code will check the length of the value when setting a property. If the value is too big, an exception is raised.

## Min/Max

For numeric fields, you can define range boundaries depending on your business rules. This value can be used on any number field whether integer or decimal based. In the generated code, the property’s setter will have validation code in it to ensure that a client cannot enter data outside of the defined range. Invalid values will not make it to the database because the value is checked in the client API. An exception is raised if the value is invalid.

## Primary Key

This property is used to mark a field as a primary key. Any number of fields can be marked as part of the primary key. When more than one field is marked, they are all used to uniquely identity a database record and by extension an object.

## DataType

This is the SQL data type defined for the field. All but few datatypes are supported. The unsupported types are Xml, Udt, and Structured. The Length property is related to the datatype in that some types like varchars need a length setting.

## Code Façade

The CodeFacade property is very much like the one defined for a table object. It allows for a separation of code from database. This property allows you to specify a completely different name for your code than is used by your database. You can define a database name for a field and a code name. The codefacade property allows you to map code names to database name. This really helps keep your code pretty no matter what your column is named in the database.

## Name

This is the actual name of the field in the database. This is name is used to compute the name in code as well if there is no CodeFacade property set.

# Relationships

There are numerous types of relationship in the model. The standard 1:1 and 1:N applies. However the mode is more robust than a standard relational database can handle so some functionality has been added.

## 1:1 relationships

The standard 1:1 relationship is handled when you create a relation from a parent table on a unique field like the primary key to a child table on its column that is unique. These fields are usually primary keys of course but they do not have to be. Each column object has a property IsUnique which determines if the column must have unique values in it. While this is always true for primary keys, there may be column you wish to mark this way that are not primary keys. For example you have a CustomerId field that is an identity integer, but you also want the social security number to be marked as unique. While the latter is not a primary key it is still unique.

## 1:N relationships

This is the most well-known relationship type. It defines a relation from a parent table on a unique column to a child table on a non-unique column. This is the famous customer/order relation. One customer has many orders.

## Hierarchy Relationships

Hierarchical relationships exist in most databases. These define a relation from a table to itself. The child column must be nullable since it would be impossible to add the first record to the table if it were not. This relation type is used to build hierarchies like employee/manager: an employee reports to another employee that is his manager.

## N:N Relationships (Associative)

The relationship type that is not supported by most relational database products is a many-to-many relationship. An example if this might be cars to features. A car may have many features like FM radio, anti-lock brakes, etc. Also a feature is related to many cars since many cars have anti-lock brakes. Each table have many related items in the other table. To define this relationship a nHydrate model allows you to define an intermediate table that holds references to the two primary tables. The intermediary table is not visible in the generated code and is used merely to relate a many-to-many relationship.

# Select Commands

Select commands provide you with more flexibility in querying data or performing custom actions on the database. There are some things that the standard querying mechanism cannot handle. You may define a select command with free-form SQL to perform any action that is valid with transact SQL. Keep in mind that since it is free-form you can write poor or even invalid SQL. The error checking is complete up to the writer.

Description  
Adding a description makes your generated code more useful. This information is used to create Intellisense tooltips, giving your API more professional functionality. When deploying your generated API the included XML file will provide developers with useful Intellisense they can use to increase their productivity.

### Generated

Setting an item to not be generated may be useful if you do not want to push out changes at present. The generated property allows you to selectively turn off generation for specific model items. When false these items will not be a part of the SQL scripts that create the database and will not be part of any generated code.

### SQL

This defines the actual SQL used for execution when the generated select command is used. This can be any valid transact SQL statement. You may use the defined parameters collection as well in the SQL.

### Use Search Object

You can optionally use a search object in code to search by specific fields. When this property is true all of the defined columns for this item will exist on a search object class for this select command. You can then create an instance of the search class and use it to pass specific values for searching to this select command.

### Code Façade

The CodeFacade allows for a separation of code from database. This property allows you to specify a completely different name for your code for the object than is used by your database. You can define a database name for an object and a codefacade name. The codefacade property allows you to map code names to database name. This really helps keep your code pretty no matter what your database object is named in the database.

### Name

This is the actual name of the field in the database. This is name is used to compute the name in code as well if there is no CodeFacade property set.

### Parameters

This collection holds a list of parameters that can be used to pass information to the select command. You may use the parameters in the free-form SQL statement. This is a mechanism that allows you to define strongly-typed parameter information for the select command and use it to write more useful SQL.

### Allow Null

This property determines if the specific column allows null values. When true, you can pass a value of null to the select command for the specified column.

### DataType

This is the SQL data type defined for the field. All but few datatypes are supported. The unsupported types are Xml, Udt, and Structured. The Length property is related to the datatype in that some types like varchars need a length setting.

### Default

This property determines the default value if none is passed in.

### IsOutputParameter

There are times when you will need to pass information out of a select command. For example you may select a page of records and return it as the return of the select command. However for display purposes you need the total number of records as well. To get two pieces of information with one query call, simply create an output parameter in the model. In the SQL you write, set the output parameter as desired. When you execute the select command in code, the output parameters will exist on that object and be set.

### Length

This property defines the length of the field. This property does not apply to data types that have a predefined value. For example an integer is always four bytes, so this field cannot be modified. However for a varchar (and other string types), you can define the maximum length of the field.

Added functionality related to this setting is the length checking of the API. The generated code will check the length of the value when setting a property. If the value is too big, an exception is raised.

## Components

There are times when you do not want to select all fields for a row in the database. A row might have a blob field with images in it or some other large data that you do not want. To reduce loading-time and memory footprint, you can use table components. These are objects which map to specified fields of exactly one table. Essentially this is a subset of fields for a table. These are fully fledged business DAL objects that can be loaded and saved, but not created. So you can load, modify, and persist changes to the database.

Description  
Adding a description makes your generated code more useful. This information is used to create Intellisense tooltips, giving your API more professional functionality. When deploying your generated API the included XML file will provide developers with useful Intellisense they can use to increase their productivity.

### Code Façade

The CodeFacade allows for a separation of code from database. This property allows you to specify a completely different name for your code for the object than is used by your database. You can define a database name for an object and a codefacade name. The codefacade property allows you to map code names to database name. This really helps keep your code pretty no matter what your database object is named in the database.

### Name

This is the actual name of the field in the database. This is name is used to compute the name in code as well if there is no CodeFacade property set.

## Custom Views

Custom views allow you to map a database view to a generated object that can be used in code. A view is read-only of course, so there is no way to make changes to persist changes. This functionality is simply a convenience that allows you to use views like any other generated object. Views can be used just like tables when querying using the compile-time checked, strongly-typed, LINQ syntax. You can query a view using an arbitrarily complex syntax that is checked by the compiler and does not issue a run-time error. This is very powerful. With this functionality you can create a virtual table in and query it and use its data as if it were a real table.

Description  
Adding a description makes your generated code more useful. This information is used to create Intellisense tooltips, giving your API more professional functionality. When deploying your generated API the included XML file will provide developers with useful Intellisense they can use to increase their productivity.

### Generated

Setting an item to not be generated may be useful if you do not want to push out changes at present. The generated property allows you to selectively turn off generation for specific model items. When false these items will not be a part of the SQL scripts that create the database and will not be part of any generated code.

### SQL

This defines the actual SQL used for execution when the generated view is used. This can be any valid transact SQL statement. You may use the defined parameters collection as well in the SQL.

### Code Façade

The CodeFacade allows for a separation of code from database. This property allows you to specify a completely different name for your code for the object than is used by your database. You can define a database name for an object and a codefacade name. The codefacade property allows you to map code names to database name. This really helps keep your code pretty no matter what your database object is named in the database.

### Name

This is the actual name of the field in the view, whether it is really in a table or not. Since this is a virtual table construct the name can be any mask you assign in the associated SQL statement.

## Custom Stored Procedures

Custom stored procedures allow you to map a database stored procedure to a generated object that can be used in code. These objects provide a way to run complex actions or queries on a database and address it with the generated API. You may have quite complex logic that returns a result set or simply performs an action with no return value. Custom stored procedures allow this interface to be seamless with the rest of your generated API.

Description  
Adding a description makes your generated code more useful. This information is used to create Intellisense tooltips, giving your API more professional functionality. When deploying your generated API the included XML file will provide developers with useful Intellisense they can use to increase their productivity.

### Generated

Setting an item to not be generated may be useful if you do not want to push out changes at present. The generated property allows you to selectively turn off generation for specific model items. When false these items will not be a part of the SQL scripts that create the database and will not be part of any generated code.

### SQL

This defines the actual SQL used for execution when the generated stored procedure is used. This can be any valid transact SQL statement. You may use the defined parameters collection as well in the SQL.

### Code Façade

The CodeFacade allows for a separation of code from database. This property allows you to specify a completely different name for your code for the object than is used by your database. You can define a database name for an object and a codefacade name. The codefacade property allows you to map code names to database name. This really helps keep your code pretty no matter what your database object is named in the database.

### Name

This is the actual name of the field in the database. This is name is used to compute the name in code as well if there is no CodeFacade property set.

# Chapter 4 – Using the DAL

## Overview

The Data Access Layer (DAL) is remarkably versatile. You can query data in the expected ways by using primary key, foreign key, object fields, and table relations. There are also some more advanced ways of query data like free-form LINQ and pagination. All of this adds up to a really robust API for your database. Using the inheritance features you can turn your database into an object base.

## Select Commands

You may create an empty subdomain and define select commands that are executed in batch. There may be a business reason for loading in batch, for example a disconnected data set. In this case, there are numerous predefined select commands that can be added to the rules cache of a subdomain and execute at one time. This is in contrast with the dependency walking described later which hits the database every time data is needed. There are pre-defined select commands for select all, select by field, select by relationship, etc. However you can also define your own select commands and use them interchangeably.

The select commands are actually used during dependency walking but can be used separately to load developer specified sets of objects. There are predefined select methods for all primary and foreign keys, as well as fields marked as searchable. All select commands are generated as partial classes and can be extended with custom select commands. An example of this is a situation where you want to load objects based on some set of non-key fields. You can create a select command that calls a custom stored procedure and then add it to the select queue like any other generated select command. Of course you can select by arbitrary fields using the LINQ functionality described below, but these cannot be queued and run in a single transaction.

## 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.

You can also page through results using the entity’s paging object. This allows you to define an ordering and load one page at a time.

|  |
| --- |
| CustomerSearch search = new CustomerSearch(Widgetsphere.Core.DataAccess.SearchType.AND);  search.BaseName = "SomeValue";  CustomerPaging paging = new CustomerPaging(1, 10,   Customer.FieldNameConstants.PersonId, true);  CustomerCollection customerCollection = CustomerCollection.RunSelect(search, paging, ""); |

Or you can use the LINQ syntax.

|  |
| --- |
| CustomerCollection customerCollection =  CustomerCollection.RunSelect(x=>x.BaseName == "SomeValue"); |

## LINQ

The RunSelect method of each collection is overridden many times. One of the included ways to query data is to pass in a LINQ query. This feature provides you with maximum flexibility since you can perform complex queries that are just not possible with the out of the box, generated methods. Also you can perform complex queries based on your relationships to other tables (or objects). For example, the where condition can be defined based on table field, a parent table field, a related table’s field, a related table’s parent table’s field, etc. Or perform all of this in the same query. As a provided convenience, there is no need to know the inheritance hierarchy of entities because you interact with objects as if they contain all their fields as well as the fields of the parent objects. This allows you to use an object without having to know that it has five layers of inheritance under it.

Another fact worth mentioning is that you **never need to join tables**. All of the relationships are defined in the model, so you can describe the where condition by referencing tables but never defining linking criteria. The makes your code much easier to read and ensures that developers do not have to memorize the structure of the database in order to write correct joins. This is a big win for code readability and consistency. Developers cannot join on incorrect fields that produce too many (or too few) results. This condition often causes unintended consequences in that the data looks correct but actually is being joined on the wrong fields. The model architect ensures that all developers are accessing data in the correct way.

## Hooking into Object Events (callbacks)

Every persistable object has events that can be used get notifications about changes. There are two events for each property of an object: Changing and Changed. The first event allows you to cancel the operation and the second is simply a notification that the property changed. The following code snippet loads a Customer object by primary key and then hooks both events. Notice that in the Changing event you can get the new value that will set the property. The current value can be access directly from the object.

|  |
| --- |
| private void LoadCustomer()  {  Customer customer = Customer.SelectUsingPK(12);  customer.FirstNameChanging +=   new EventHandler<BusinessObjectCancelEventArgs<string>>(  customer\_FirstNameChanging);  customer.FirstNameChanged +=   new EventHandler(customer\_FirstNameChanged);  }  void customer\_FirstNameChanged(object sender, EventArgs e)  {  //Do something after FirstName has changed  }  void customer\_FirstNameChanging(object sender,   Widgetsphere.Core.EventArgs.BusinessObjectCancelEventArgs<string> e)  {  if (e.NewValue == "SomeValue")  e.Cancel = true;  } |

There is also a PropertyChanged event for the object as a whole. If any property value is changed the PropertyChanged is raised. You can hook into this event to get notification of a state change in the object.

## Using Partial Classes to Expand the Object

The generated framework is extendable in the fact that all classes are rendered as partials. The top level file name “Someclass.cs” is generated once. If the file exists, the generator does not overwrite it. You can add you custom modifications to this file and they will not be overwritten. There is a second file with the naming format of “Someclass.Generated.cs”. This file is controlled by the generated and will always be overwritten. You should not make any changes in this file. The VS.NET compiler meshes these two classes together to make a ”Someclass” object type. When you instantiate or use this class, there is no way to distinguish that is generated code and what is custom code. You see only one class type.

For example you may have a table “Customer” with two fields “FirstName” and “LastName”. If you wish to also have a read-only property that concatenates these two fields together you could create one in the partial class. In the following code snippet, I have extended the “Customer” class with a new property “FullName”. Now in my code I can address the customer property “FirstName”, “LastName”, or now “FullName” as well.

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

## Validation Layer

There is a validation layer that allows you to add custom validation rules to the framework. You can add as many rules for an object as you wish so the logic can be arbitrarily complex. Every business object has an “IsValid” method. This is always true by default since there are no rules to pass. You can add validation rules to an object that will pass or fail and the result will be reflected by the IsValid method. To use this functionality simply override the “GetRuleViolations” method in a business object. Perform whatever checks your business rules dictate and add the appropriate custom violation rules to the violation list. If there are any violation the IsValid method will return false, other it will return true.

To demonstrate this will use the Customer object. I will override the GetRuleViolations method and check if the first name is greater than four characters. If not I will add a custom violation rule I wrote to the list. I have added this code to the extender class of the Customer object. Remember that each generated object has a generated partial class and an extender partial class. These are used at compile time to create object definition. I have merely extended the Customer object.

|  |
| --- |
| using System.Collections.Generic;  using Widgetsphere.Core.DataAccess;  namespace Acme.TestProj.Business.Objects  {  partial class Customer  {  public override IEnumerable<IRuleViolation> GetRuleViolations()  {  List<IRuleViolation> retval = new List<IRuleViolation>();  if (this.FirstName.Length <= 4)  retval.Add(new CustomerNameValidationRule());  //Add any number of other rules...  return retval;  }    }  } |

Now we must define the custom violation rule for this situation. This is a simple class and only need to implement a message property that can be used to identify what failed.

|  |
| --- |
| public class CustomerNameValidationRule : IRuleViolation  {  public string Message  {  get { return "The name must be greater than 4 characters."; }  }  } |

Now in the client code I create a Customer and check its validity. If it is not valid I print out the violation messages.

|  |
| --- |
| CustomerCollection customerCollection = new CustomerCollection();  Customer customer = customerCollection.NewItem();  customer.FirstName = "dd";  if (customer.IsValid())  {  //This is valid so do something  customer.Persist();  }  else  {  //NOT VALID  foreach (IRuleViolation violation in customer.GetRuleViolations())  {  Debug.WriteLine(violation.Message);  }  } |

## 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.

## Subdomain

Since all objects are in a subdomain and subdomains are persisted inside of transactions, you can persist a group of changes in a single transaction. When you load one or more objects of a single type into a subdomain and persist, this is of course done inside of a transaction. Now if you want to perform a group of opertions transactionally, how do you do it? Well you simply load all objects into the same subdomain. This is done by adding typed collections into a subdomain and using these parent collections to add, edit, delete object of the appropriate type.

The first example shows how to load (or create) a collection and then use its implicit subdomain to get another collection type from it. You can get a reference to any number of collections. After you have a reference to a collection, you can add, edit or delete objects from it. All objects in collections that are parented in the subdomain will be saved in the same database transaction.

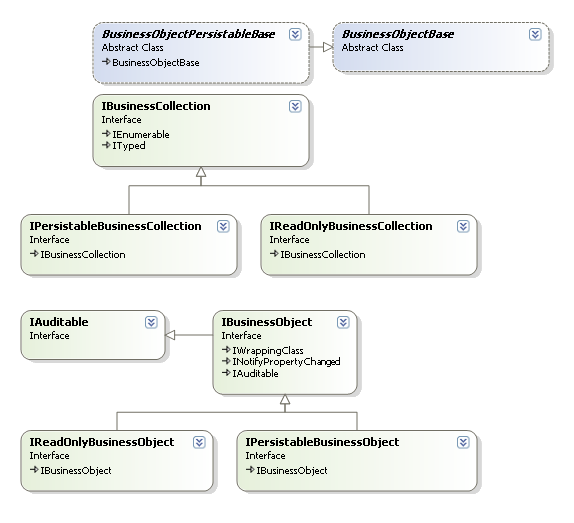
|  |
| --- |
| //Select all customers  CustomerCollection customerCollection = CustomerCollection.RunSelect();  Customer customer = customerCollection.NewItem();  //Set Customer object here...  customerCollection.AddItem(customer);  //This is a normal transactional save  //customerCollection.Persist();  //However now we will add orders  //Get an order collection in the same subdomain as the customers  OrderCollection orderCollection =  (OrderCollection)customerCollection.SubDomain[  Collections.OrderCollection];  //Create an order  Order order = orderCollection.NewItem();  //Set Order object here...  orderCollection.AddItem(order);  //Call the persist of any collection here  //The entire subdomain is saved transactionally  customerCollection.Persist(); |

Alternatively, you can declare a subdomain explicitly and then get a reference to collections. The following code is identical to the code above but is easier to read since it follows the creation of the subdomain object, adding objects, and saving the subdomain. This makes more logical sense. However the fact that you must manually add select command obejcts to a subdomain might not be so intuitive. When a subdomain is created it has no collections or objects in it. The code below shows how to get a reference to a collection, add the select all customers command and execute the new command against the database.

|  |
| --- |
| //Create a subdomain explicitly  SubDomain subdomain = new SubDomain();  //Get a reference to the two collections  CustomerCollection customerCollection =  (CustomerCollection)subdomain[Collections.CustomerCollection];  OrderCollection orderCollection =  (OrderCollection)subdomain[Collections.OrderCollection];  //Add the select rule to get all customers  subdomain.AddSelectCommand(new CustomerSelectAll());  //Execute the rule against the database  subdomain.RunSelectCommands();  //Add a customer  Customer customer = customerCollection.NewItem();  //Set Customer object here...  customerCollection.AddItem(customer);  //Add an order  Order order = orderCollection.NewItem();  //Set Order object here...  orderCollection.AddItem(order);  //Persist the entire subdomain here  subdomain.Persist(); |

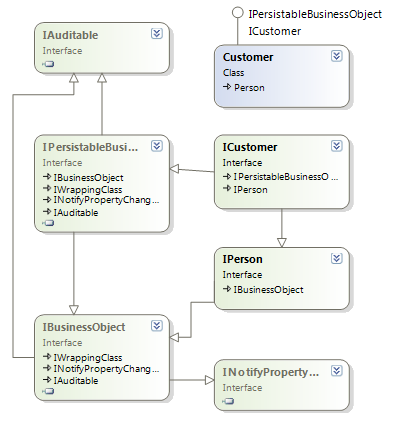
Advanced mapping concepts

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 type table 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.

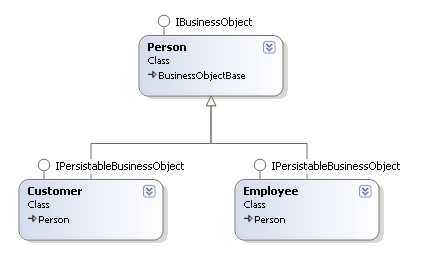
## Interfaces and base classes

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.

## Inheritance

The generator 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.

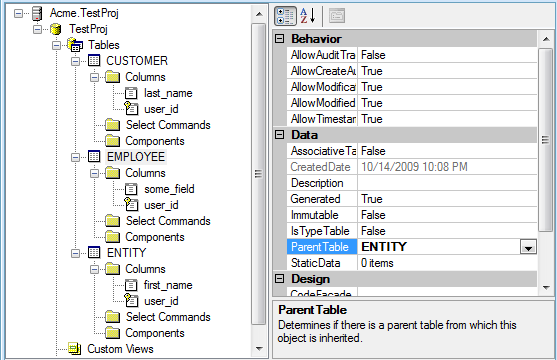
Inheritance is a feature many people want but have a difficult time implementing. There are numerous ways of implementing this functionality. All of which have pros and cons. There are two basic ways to create an object database. The first is the put all fields in a single table from the entire object hierarchy. This may duplicate data across parent child tables but makes saving and loading simpler. The second way is to split the data logically based on the object hierarchy of your model. The properties of parent objects are stored in parent table fields and the child properties are stored in child table fields. This makes inserts, deletes, referential integrity, etc more difficult but in a generated framework there is no reason to care. The nHydrate generator opts for the latter. Different tables hold logical objects and are tied together with API.

Let us take a simple example of an Entity and a Customer. Let us say that Customer derives from Entity and both have a primary key of UserId. Now in code you would create an Entity class, then a Customer class that derives from it. In code, when you instantiate a Customer you actually have an Entity as well. You can of course cast the Customer as an Entity and use the properties and methods of the base class. This is completely possible to model and generate with nHydrate. You simply define your parent table in the model and define a relationship from Entity to Customer. In the generated code, there is an Entity class and collection; as well as a Customer class and collection.

The code works as expected but how is it saved? In the database, there is also an Entity and Customer table. There is also a one-to-one relationship from Entity and Customer. This relationship dictates that a Customer row cannot exist without a parent Entity row. This is just like your code. You cannot instantiate a derived class without also having its base class. You can cast to the base, so you essentially have a base class in memory as well. They are of course the same object but I am making a point. A Customer cannot exist without is Entity base. This relationship also defines the rule that all Customers are Entities but not all Entities are Customers.

Now in the database there must exists one row in Customer table for a Customer object, but also one row in the Entity table since the base properties are there. It is not possible to have a row in the Customer table without a corresponding row in the Entity table. This is all completely transparent to you the developer. In code you only need create a Customer object and save. The back end API knows how to split the data across tables. When you load a Customer, the API knows how to load the data from different tables into your objects. In fact, you need not be aware of any of these details or even that the objects are derived in the first place. You can use them like any other object in the framework. That is the beauty of generated code. Then entire complexity of inheritance hierarchies is hidden from you. You do not care how tables are implemented in the database or how each relates to the others. The database truly becomes a data repository that stores stuff for you. The details of the load and save are not anymore so important.

Let us expand this example to have multiple derived tables. Not only do we have a Customer derived from Entity, but we also have Employee as well. They are both Entity objects but play different roles in our system. Now when you create a Customer and save it, there is one row inserted into the Customer table of course and one row inserted into the Entity table. There are two database rows created for each Customer. Now when you create an Employee object, there are also two database rows created: one in Employee table and one in Entity. This may look odd to traditional relational database programmers. After adding one Customer and one Employee, the Entity table has two rows in it. This is completely logical when you think about what we are doing. We are splitting data across tables to facilitate object inheritance. This is not functionality that a relational database handles implicitly. We have turned our relational database into a sort of object database.



All of the relationships of the table and all base tables can be used as if the child object contained all of them. For example, if Entity has a relation to an EmailAddress table to hold multiple email address for an Entity, each Customer and Employee would also have this relationship. In fact, there is no way to know in code which layer has the relationship. It is only important that Customer and Employee have an EmailAddressList hanging off of them so the developer can walk the relationship and pull the needed data.

This functionality can be taken to any depth you desire. An object can be derived from another that is derived from another, and so on. A relationship is visible from all layers in any object above the level at which the relationship is defined. This also facilitates LINQ queries. In the example above you could select based on the following query.

|  |
| --- |
| CustomerCollection customerCollection =   CustomerCollection.RunSelect(x=>  x.EmailAddressList.Email == "q@q.com"); |

Notice that you cannot tell that EmailAddressList is actually related to Entity. Since Customer is a derived class of Entity, it too has the LINQ capabilities and properties of its parent object. This allows you to perform quite powerful ad hoc, strongly-typed queries across whole hierarchies of objects. In fact, if all tables in a database are related in some way through a relationship (no islands), you can query any set of objects based on the existence of other objects with LINQ. Keep in mind that all queries are strongly-typed and compile-time checked. You do not need to deploy an application to discover a logic error. If you have an error, your application will not even compile.

Inheritance is a powerful tool that has been a long time coming. The intricacies and problems associated with it are not easily solvable, which is why most applications avoid it all together. The nHydrate framework provides an easy to use interface to define and use inheritance hierarchies in code, with little understanding of the database particulars relating to persistence of such a framework.

## Business Objects

All objects implement the IBusinessObject interface. This provides all objects with some basic functionality that you can use to build generic frameworks that work with any obejct type. In addition, all persistable objects derive from IPersistableBusinessObject that provides persistence functionality. All objects also implement other interfaces that make accessing properties and fucntionality more general purpose.

## Implicit vs Explicit SubDomain

As discussed previously, there are implicit and explict subdomains.You can declare a subdomain object and use it to retrieve or save objects. This is an explict subdomain. You know of its existence because you create it. However all objects are part of a collection and all collections are part of a subdomain. So when you call a static method on a business object or collection the returned object or collection is part of a new, implict subdomain. It always exists. You can get a reference to the subdomain by accessing the Subdomain property from any collection. There is no difference in the object structure base don how it was created. Either creating a subdomain and populating it or calling a static method an object or collection will give to you the exeact same object structure and hierarchy.

Automatic Retrieval  
TODO

## 1:N Relationship

The most common relationship type is a one-to-many. This allows you to define a single independent object that has many related, dependent objects. An example of this is the Customer/Order scenario. Each Customer object has many Order objects and each Order object has exactly one Customer. When you define a relation in the relationship dialog this relationship type is determined when the relationship parent table column(s) are unique and child table column(s) are not. The model determines that this is a 1:N relation.

For this type of relationship, the generated code will create an OrderList list on a Customer obeject to signify that you can loop through related Order objects. Also there will be a CustomerItem property on an Order object signifying that it has one related customer. The second scenario is complicated by the fact that the foreign key on Order might allow null. If this is true then the CustomerItem property of the Order object will also allow null in the generated code. If the foreign key does not allow null, then you can be sure that the CustomerItem property will always have a value, since the Order cannot exist in the database with a null customer value.

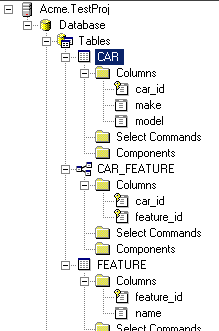
## Hierarchy Relationships

TODO

## Parent-Child Relationships

In addition to the normal relationships in a standard relational database, nHydrate also defines inheritance. The database details of these relations are somewhat hidden, however they must resolve to a database of course. The .NET framework allows you to use object inheritance. Child objects can derive from base objects allowing the child to inherit all properties and functionality of the base. This relationship is not supported by a relational database. There are many frameworks that implement this functionality in a myriad of ways. nHydrate allows you to keep this information in the model and let the generator engine

## Many to Many Relationships (Associative)

Many-to-many relationships are the most cumbersome relations to model in a database. They are not built into SQL and so an alternative scheme must be devised to represent them. The most common way, and the one employed by the nHydrate, is to create an intermediary table that holds the key to each of the other two tables. This intermediary is an associative table and is visible in the model but not in the generated code.

Take the example of a car having a list of features. In a database we would have a Car table of course and a Feature table to hold a list of possible features for a car. A car object can have many features and a Feature object can be related to many cars. In our model we will also create a Car\_Feature table and add the primary keys of both the Car and Feature table. All fields in the new table should be marked as primary key, the table shoul dbe marked as associative, and the two relationships should be defined Car -> Car\_Feature and Feature -> Car\_Feature.

|  |
| --- |
| //Get a list of all cars  CarCollection carCollection = CarCollection.RunSelect();  //Get the first car in the collection  Car car = carCollection[0];  //Get a list of features for that 1 car  IEnumerable<Feature> featureList = car.FeatureList;  //Get a list of all features  FeatureCollection featureCollection = FeatureCollection.RunSelect();  //Get the first feature in the collection  Feature feature = featureCollection[0];  //Get a list of cars for that 1 feature  IEnumerable<Car> carList = feature.CarList; |

In the above code the Car\_Feature table does show at all in the snippet. There is no CarFeature object and you do not need to walk a relationship through two tables. The code is very simple; a Car object has a list of associated features and a Feature object has a list of associated cars. The true relationship is hidden in this intuitive code methodology. The database schema is shown below; however this is not what you experience when using the generated code.



## Parent object validation retrieval

TODO

## Searching and Paging

There are numerous ways to search. First you can search with a predefined search object. There is a strongly-typed search object generated for each entity type. It allows you to define any of the object’s field for searching based on a “AND /OR” connector.

|  |
| --- |
| CustomerSearch search = new CustomerSearch(Widgetsphere.Core.DataAccess.SearchType.AND);  search.BaseName = "SomeValue";  CustomerCollection customerCollection =   CustomerCollection.RunSelect(search, ""); |

You can also page through results using the entity’s paging object. This allows you to define an ordering and load one page at a time.

|  |
| --- |
| CustomerSearch search = new CustomerSearch(Widgetsphere.Core.DataAccess.SearchType.AND);  search.BaseName = "SomeValue";  CustomerPaging paging = new CustomerPaging(1, 10,   Customer.FieldNameConstants.PersonId, true);  CustomerCollection customerCollection = CustomerCollection.RunSelect(search, paging, ""); |

Now with LINQ you can actually performs some ad-hoc querying while still using strongly-typed objects.

|  |
| --- |
| CustomerCollection customerCollection =  CustomerCollection.RunSelect(x=>x.BaseName == "SomeValue"); |

The code above returns the same result as using the search object, but now you can write arbitrarily complex queries, even using implicit joins.

|  |
| --- |
| CustomerCollection customerCollection =  CustomerCollection.RunSelect(x => x.BaseName == "SomeValue" &&  x.UserType.Name == "MyType"); |

Notice that the complex LINQ query above actually is joining three tables. A Customer is derived from Person and Person has a relation to UserType. The parent-child relationship is transparent in this query. It looks like Customer has an associated UserType object, which it is through its inheritance from Person. Also note that there are no join operators in this query. There is no need to define the relationships because they are already described in the model.

Not only can you query based on implicit joins down the inheritance chain and laterally with related entities, but you can also query across multiple relations with different cardinalities. In the code below, Customer has a one-to-many relationship with Package. You do not notice it right away because the query looks exactly the same as when looking down the inheritance chain or laterally in the relationship graph. There is no identifier of singularity or plurality.

|  |
| --- |
| CustomerCollection customerCollection =  CustomerCollection.RunSelect(x => x.BaseName == "SomeValue" &&  x.Package.Name == "Pack1"); |

There is no reason to know the cardinality of the relationship. You simply want find Customers that have a related Package object with some field value. This is same question we are asking above when we want to find Customers with a related UserType object, even though a Customer can only have one UserType. This is another aspect of how the database model is abstracted from the code.

In the model you can define fields on which to search at runtime. This aids in development since there are many times when you are searching based on the same field of an object. When a field is marked as searchable in the model, a database index is created and a custom static search method is generated on the entity’s related collection class. This allows you to search for a group of objects with one line of code.

|  |
| --- |
| CustomerCollection customerCollection =   CustomerCollection.SelectByBaseName("SomeValue", ""); |

This is quite useful of course but if there are many objects you may need to page through them as well. You can use the Customer paging object to do this.

|  |
| --- |
| CustomerPaging paging =   new CustomerPaging(1, 10, Customer.FieldNameConstants.PersonId, true);  CustomerCollection customerCollection =   CustomerCollection.SelectByBaseName("SomeValue", paging, ""); |

## 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.

## Client-side LINQ

TODO

## Dirty Objects

You can also determine if a subdomain needs to be saved by checking its “IsDirty” property. All objects exist in a subdomain. So you can determine if anything has changed since loading by checking it.

|  |
| --- |
| bool b = customer.ParentCollection.SubDomain.IsDirty;  bool b = customerCollection.SubDomain.IsDirty; |

## Bulk Updates

The bulk update functionality is quite useful for updating a single field in a table across many rows based on a conditional. You may need to update all last names, expiration dates, product names, etc based on some condition. Normally you would need to load these objects into memory, set the appropriate values and persist these values back to store. There is an easier way using strongly-typed, compile-time checked LINQ statements.

In the following code snippet the Category’s Name field will be update to the value “Category V” for all categories with the name “Category I”. This is not the best example since we would probably expect all categories to have a unique name, but the principle is demonstrated.

|  |
| --- |
| CategoryCollection.UpdateData(x => x.Name,  x => x.Name == "Category I",  "Category V"); |

A better example might be to update a customer status based on some complex conditional. The following code snippet updates the Customer object’s Status field to the value “Expired” if the expiration date has passed and the current status is not that value already. The conditional can be arbitrarily complex. Also notice that all facets of this statement are compile-time checked. The fields are strongly named (not in quotes) and the values are type checked: date and string.

|  |
| --- |
| CustomerCollection.UpdateData(x => x.Status,  x => x.ExpirationDate < DateTime.Now && x.Status != "Expired",  "Expired"); |

## Aggregates

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.

The code below will return the maximum value of the “basename” field for all Customers with a “personid” less than 100.

|  |
| --- |
| string name = CustomerCollection.GetMax(x =>   x.BaseName, x => x.PersonId < 100); int customerId = CustomerCollection.GetMax(x => x.CustomerId); |

There is also an UpdateData aggregate method that will set exactly one field in one or more database rows that match a where condition. This is a convenience method that allows you to set many database rows without having to actually load the rows, set them and save them back to the database. You can update all matching rows in one line of code.

|  |
| --- |
| CustomerCollection.UpdateData(x =>   x.BaseName, x => x.BaseName == "Name1", "Name2"); |

The code above changes all Customers with a “basename” field value of “Name1” to the new value of “Name2”. This functionality is also strongly-typed. You new value cannot be an “object”, but must be a string since the field is defined as a string. This ensures reliable code.

There is also a method for deleting in bulk. Using much the same syntax as above, you can call the DeleteData method with a where expression to remove all matching records. This functionality allows you to delete in bulk without selecting data to the client.

The ability to pass in LINQ queries is actually even more useful than just implementing strongly-typed data access based on a model. It actually allows you to define complex queries and use them in multiple places. In the following example, there is a complex where clause for accessing and updating data. Instead of defining the conditional twice, I have defined it once and used it in two aggregate statements.

|  |
| --- |
| //Build the “where” statement for later use  Expression<Func<MessageQuery, bool>> where = x =>  x.RecipientUserId == physician.UserId &&  x.ViewedDate == null &&  x.SenderUserId != null &&  x.IsDeleted == false &&  x.NotificationSent == null &&  x.CreatedDate > threshold;  //Use the where statement to get the count  int count = MessageCollection.GetCount(where);  if (count > 0)  {  //Do something...  //Update the notification date to now with where statement  MessageCollection.UpdateData(x =>   x.NotificationDate, where, DateTime.Now);  } |

## Dependency Walking and Lazy Loading

A very useful feature of the generated framework is the ability to load an object and “walk” its relationships. There is no reason to define what objects you need up front. Simply load an object and start calling on its related objects. This works in both directions for both parent and child relationships (indeed even self-referencing relations).

|  |
| --- |
| //Load all Customers  CustomerCollection customerCollection = CustomerCollection.RunSelect();  //Get the first Customer  Customer customer = customerCollection[0];  //Write out the UserType  System.Diagnostics.Debug.WriteLine(customer.UserTypeItem.Name);  //How many Packages does this Customer have?  System.Diagnostics.Debug.WriteLine(customer.PackageList.Count);  //The Customer's first Package's Customer (pointer back same customer)  System.Diagnostics.Debug.WriteLine(  customer.PackageList[0].CustomerItem.BaseName); |

In the example above, you can walk the Customer relationship to its one UserType object. A Customer has many packages, so there is a PackageList not a PackageItem property. You can then pull the first item from the Package list and find its one Customer, which is of course the same Customer you started with. The relations can be walked up and down the relation graph.

It is interesting to note that all objects are loaded into the same subdomain container. You can make changes to any number of objects in this subdomain as you walk the hierarchy and save with a call to one persist method that is performed on the database in one transaction. All objects will be updated together or fail together.

## Typed lists

Typed lists provide a great way to map database values to code values. There are times when you have numeric values in a database that correspond to items that seldom if ever change. For example user types, category types, etc. Many people use “magic numbers” to associate data with some database type. A developer might set a Customer object’s UserType to 1, 2, or 3. This is hard to read and quite error prone. New developers on a project do not know what these “magic numbers” mean. nHydrate addresses this issue by providing type tables. Create a table with an integer identity field and a name field. After you add some static data to the table, essentially defining the map, the object can be generated. Now your code can reference an enumeration instead of a literal number. The enumeration is generated and maintained by the framework. The “Customer.UserType” field is a faux property created by the framework. You would have an integer UserTypeId that maps to the primary key of the UserType table, however now there is an additional property that allow you get or set the value in code using the new enumeration.

|  |
| --- |
| //Old Way  customer.UserTypeId = 1;  //New Way  customer.UserType = UserTypeConstants.HighRoller; |

## Nullable types

Everyone has experience the problems of setting database values to DbNull and code objects to null. There is no need to ever do this now with a generated API. All nullable fields on tables can be set to null, even integers, GUIDs, or other traditionally non-nullable types. If a field is defined as nullable then the database will support nullable of course. However the generated code will as well. If an integer is nullable, the API uses the nullable types in .NET to represent the value. This means that you can actually set your decimal “Cost” field to null if your business rules specify all new products have a null value. This also makes code easy to read in that you can look at the type of a field and know whether it is nullable or not.

## Advanced Querying

TODO

## Audit Tables

Many times it is necessary to know the state of all database rows at all points in time. Financial auditing is one use of this functionality. It can be used for database administrators (or developers) to determine when any value changes in any row in any table. This is quite a useful feature for anyone who actually has need of this functionality and quiet non-trivial to actually build. This is not information that can be retrieved unless an auditing system is in place, since a normal database only stores the current state of a row and not all past states.

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.

# Chapter 5 – Application Configurations

## POCO objects

The data transfer layer is built on POCOs (plain old CLR object). The objects have no functionality and are essentially data containers. Each object is serializable and can be used to pass around strongly-typed data in a standard way. The POCO objects are a little less than POCO in that each has an enumeration of field names for convenience in building strongly-typed UIs. This is still not functionality or methods, but is a little more than a property container. Strictly speaking though, the DTO is data with no functionality with is the hallmark of a properly constructed DTO layer.

## Data Transfer Extensions

The POCO DTO layer must have functionality somewhere. Since the objects cannot have functionality, the generator creates assemblies with extension methods on them to extend the functionality of the DTOs. This ensures that behavior is separate from data. The DTO layer does not have a reference to any of the extension assemblies. They can optionally be used to provide behavior to the data container. This will somewhat mirror the DAL in that you can execute methods against the DTOs as if they had behaviors.

## REST

The REST services project is a complete service to use the generated objects from JavaScript using XML or JSON. This is much like the WCF service in that you can remotely access and manipulate objects but this assembly is specifically designed to be used in a REST-ful way.

## WCF

The generated WCF service allows you to perform querying, persisting, and dependency walking across the wire using WCF. There is a proxy assembly that communicates directly with this service or you can use this service in a stand-alone fashion any way you like. It essentially provides all the necessary functionality to use the generated objects as defined in a model remotely.

## Configurations

There are numerous ways to build and configure your applications. The nHydrate framework allows you to build different types of apps. You can target multiple platforms as well. We will address some of the configurations below.

## Direct Connection with DAL

The easiest way to build an application is to use the DAL directly. This can be a Windows or web application with a direct connection to the database. The DAL is quite function and allows you to write compile-time checked, dynamic LINQ queries. All aspects of its usage are compile-time checked in fact. This is a good choice for small projects. You will need to generate only two projects: the DAL and installer project. The installer assembly is used to compile the needed stored procedures into your SQL Server database. It can also be used to populate static data and run any initialization scripts. The DAL assembly communicates directly with the database. You can use its objects to load and save objects in a transactional fashion. It can also be used to walk dependencies in an intuitive way.

## Inversion of Control

A more sophisticated configuration uses the Inversion of Control (IoC) pattern. This methodology allows you to write code against a data transfer object layer (DTO) and use dynamically loaded proxies to communicate with your database.

Software now runs in many different domains. By this I mean the same core application or pieces of it may run in a website, a Windows application, a mobile device, etc. All of these domains have their own idiosyncrasies, needs, memory limitations, etc. There is no one master application that will fill the needs for all spheres of execution. The IoC pattern allows your application or pieces thereof to run in different domains without requiring the entire application framework. This is especially needed for mobile devices since there have more stringent memory constraints, but other domains can quite nicely use this methodology as well. For example, disconnected Windows applications where you do not want direct connections to databases.

An IoC framework allows you to define a core application framework that may be heavy-weight since the bulk of your functionality and business rules are there and light-weight proxies that are deployed to your clients machines or devices and can communicate with a heavier-weight server.

The nHydrate generation framework as first released created two main projects: installation and (Data Access Layer) DAL. These projects were generated from a defined model and could be used to build applications from the generated objects with ease. Depending on the size of your model (i.e. database structure), the DAL assembly could get quite large. If your database contains hundreds of tables, as some do, the DAL assembly certainly would not fit on a mobile device. Even using a standard Windows application or website, there is really no need to have all functionality present at all times. The nHydrate framework has new generators that allow you to create additional projects from your model. The DAL is still created as always however there are now projects that can access it remotely and reduce the footprint on clients.

## Generated Projects

The IoC framework works by defining proxies that communicate with a server using Data Transfer Objects (DTOs). The DTO layer is the cement that holds all other functionality together. The DTO layer has name property objects that hold data. Just the raw data is passed around keeping network traffic to a minimum. Also these objects have no functionality. There is no code that defined on these objects. This allows you to write UI code against the DTO layer with a very small footprint. The interesting part comes when defining the communication. Your client application whether UI, unit tests, etc is written against the DTO layer. It has no reference to any other layer of your framework. The communication is handled by a proxy. Out of the box the nHydrate generator can create three proxies for you: DALProxy, WCFProxy, and MockProxy. The DALProxy proxy is used for a direct connection. The DTO layer is mapped to the DAL and database saves and loads are performed quite quickly. The WCFProxy talks to a generated WCF service. This allows your application to be deployed anywhere on the Internet and talk to a public WCF service. All DTO objects are transparently shuttled to and from the WCF service. Your remote client can interact with the data store as if it were a local connection. The MockProxy allows you to build mocks for unit tests. The beauty of the proxy methodology is that your written application does not need to know any information about the proxies. The same code can execute against any proxy. So your application can run locally or remotely via WCF, or using mocks for unit tests. The power of this framework is that you can write once and run anywhere. Your hand-written application code does not need to be changed for any of the proxies. They are simply the mechanism for interacting with the data store. They are completely swappable. You can extend this framework by writing a proxy for any functionality you wish. Notice that the mocking proxy does not even talk to a data store. It raises events that your unit test writers can use to populate known data scenarios for unit tests. You can even write a proxy for nHibernate or any other generative framework if desired.

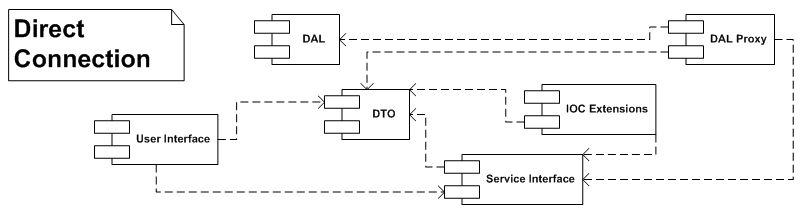
The projects needed for a complete IoC implementation are as follows.

1. The Service Interfaces
2. The IoC extensions
3. The DTO layer
4. One or more proxies

The first three are needed on the client and are very light weight. At least one proxy assembly is needed on the client as well and it is similarly light-weight.

## DAL Configuration

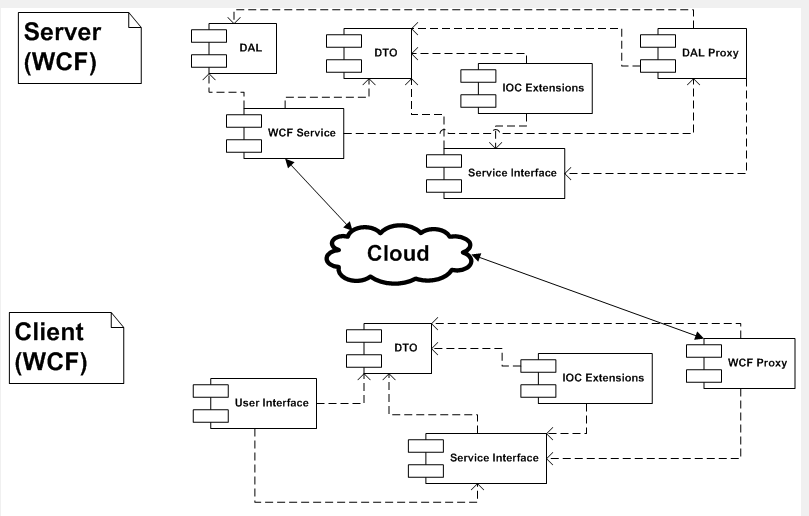
The direct connection is the simplest and fastest way to communicate. It relies on the DAL proxy assembly to shuttle information between the DAL and DTO layers. All objects in the diagram below are generated except the UI. This of course is the part of the application that you must write. Notice that the UI (your custom application layer) only references the DTO and Service Interfaces assemblies.



The Service Interfaces assembly knows how to create and save objects based on the configured proxy. The proxy itself does not need to be referenced. When your application calls for a DTO to be created, it asks the Service Interface for a new object. Conversely, when the application needs to save an object (or list of objects) it asks the Service Interfaces assembly to do it. That assembly knows how to handle the requests because of the proxy configuration.

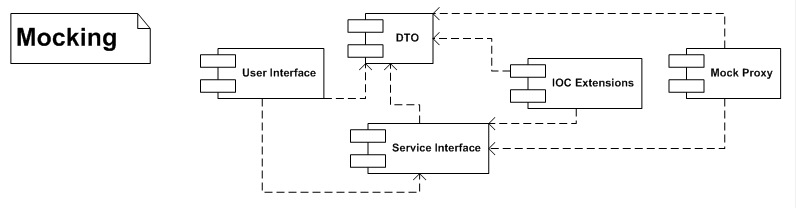
## WCF Configuration

The WCF configuration allows you to have a remote client that communicates through a WCF service with the data store. The client configuration looks much like the direct connection except that the configured proxy is not the WCF proxy assembly. The UI talks to the Service Interfaces assembly as in all cases, however the SI layer is now talking to the WCF proxy. That proxy knows how to communicate with the remote WCF service. This service is also generated and extendable. Once the request is on the server side, it acts exactly like the direct connection because it is a direct connection at this point (by default). If desired you can setup the WCF much like the application client. It too can connect to a proxy other than the DAL. It can connect to the mock proxy or a custom proxy. By default the WCF service uses the DAL proxy to communicate with the data store like a locally connected application.



## Mocking Configuration

The mocking configuration allows you to hook your application to a mock. This is a simulated data store where all data is loaded or saved with logic. The mocking proxy assembly has events to which you can attach to handle every situation that a data store would. This allows you to simulate a load, save, advanced query, etc and return a canned set of results. This is used to build unit tests. For test driven development (TDD) this is a must. The mocking layer allows disconnected simulation of a data store without having an actual database. Again your application does not need any special code to handle this additional layer. Simply swap the proxy used in the configuration file and you are mocking.



## DAL and IoC

The DAL (data access layer) can still be used to write application directly against a database. This is a very simply way to write an application. However if you have application requirements for TDD (test driven development), mocking, inversion of control, etc you will need to opt for the multi-assembly IoC application configuration. The latter is a bit more complex but provides a way to deploy an application in a distributed way. For many projects, using the DAL directly will be a very good implementation design, especially for small to medium sized projects. The DAL provides you with a great number of features like dynamic, compile-time checked LINQ syntax that is just not possible with the IoC layer. However building enterprise level software does require a more robust application framework, so the more complex IoC layer will probably be required.

## Mobility

This framework also provides a way to deploy a mobile version of your application or part thereof. The IoC components and containers are so small that there is no issue moving your application to a mobile device with a much smaller memory footprint if desired. The nHydrate framework does not generate the actual mobility classes, but creates the interfaces on which you would build this application piece.

Application Design

The first different n application building from using the DAL directly will be an absence of DAL objects. So far we have discussed creating a Customer collection directly as follows.

|  |
| --- |
| CustomerCollection customerCollection = CustomerCollection.RunSelect(); |

This syntax is easy and gets all customers with one line of code. Using the IoC layer we will write code against the DTO layer and use extension methods to perform actions. The following code snippet creates an empty generic list of CustomerDTO objects and runs an extension method from the IOCExtension layer.

|  |
| --- |
| List<CustomerDTO> customerCollection = new List<CustomerDTO>();  customerCollection.RunSelect(); |

The first thing you notice is that the same operation took two lines of code now. Since we are using extension methods, you must have an object instance to actually perform any actions. In the DAL, there are many overloaded static methods that allow you to perform action with no object instances. This is very convenient and allows you to perform many actions with one line of code. When writing against the DTO layer the static method methodology is gone.

Despite this small limitation, much of the code is very similar or identical. For example saving a collection or object is the exact same syntax. Since we have an object instance to save, we can simply call the extension method on it to do so.

|  |
| --- |
| //Save one object  customerCollection[0].Persist();  //Save the whole collection  customerCollection.Persist(); |

While dependency walking is not supported though the generated property construct, it is supports with extension methods. The functionality is equivalent but the DTO must use methods instead of the easier to read property syntax. Also the DAL caches the relations once you walk down that path. There is no hit on the data store for the second walk. In the DTO, since you are calling a method, it is executed each time your call it and thus the database is hit each time you execute the method.

|  |
| --- |
| //DAL dependency walking  Customer customer = …get a customer here…;  BusinessObjectList< Order> orders = customer.OrderList;  //DTO dependency walking  CustomerDTO customer = …get a customer here…;  OrderDTO[] orders = customer.GetOrderList(); |

# Chapter 6 - nHydrate vs Entity Framework

The nHydrate generator and Entity Framework (EF) both address a need to add a generation layer between the software developer and database. There is a lot of code that sits on the database and it used to be manually written. There are many products and applications now available to create this repetitive code for you. The nHydrate generator is one solution and we will compare it to Microsoft’s EF. First realize nHydrate is a higher level developer product than is EF. While EF gives you a platform to build applications framework based on a database, nHydrate generates much of that framework for you. So keep in mind that the comparison is not really an apples-to-apples comparison. There are many actions and much functionality that you can write on top of an EF model that nHydrate generates out of the box with no custom code writing at all. In this respect, nHydrate attempts to provide the software architect and developer with a higher level view of an application framework not a mere database snapshot.

## Models

First, EF is a generator based on a loose database model. The model need not match the database exactly. There are numerous complex mapping scenarios that allow you to map code entities to physical database implementations. This is very inclusive but provides little in the way of guidance. Essentially the tool is all things to all people. It really has to be since Microsoft supports virtually the whole world and all solutions within it. The generated code is based on third-party T4 templates that provide the basic CRUD layer functionality.

nHydrate is a little more restrictive by design. It is also an extendable framework but does have a very robust set of templates right out of the box. You most likely will build an entire application without ever extending the framework. The included templates are based on real-world application development.

The main issue with EF from an application standpoint is the lack of a metadata driven, single model. EF does have a design canvas. This is a graphical model; however the necessary metadata to distinguish complex entity types and interactions is complex or non-existent. nHydrate is driven by a single model that allows you to fashion your entities and define their associated intricacies.

## Stored Procedures

Also there are security concerns; EF runs dynamic SQL queries right out of the box. This has the shortcoming of not only causing the SQL caching mechanism of not being as effectively as possible but also requiring a loosing of security to allow free form SQL statements being run. You will need to enable this permission to the process the running of queries in this fashion. This is not an insurmountable issue of course, but merely one to keep in mind when planning. This restriction can be overcome in EF by action mapping to stored procedures. This is a bit cumbersome process but will remove the security loosing for the EF default functionality. nHydrate generates stored procedures by default and uses them build the CRUD layer. This not only ensures the fastest data access thanks to SQL query plans, but also alleviates you having to manually write the stored procedures by hand. This is a cumbersome and time-consuming process for the smallest of entities; however for inherited ones it is not pleasant at all.

## Inheritance

This actually brings us to the case of inherited entities. In EF, there are two ways of representing an inheritance hierarchy. First you can have a monster table with a lot of nullable fields with a discriminator column to determine the actually object type. This not only uses more space than necessary but creates complex objects that are more complicated for non-developers, report writers, etc. nHydrate uses the second methodology to maintain simplicity. Each object type has its own table and there is a relationship hierarchy between the tables. The main difference between EF and nHydrate is that to setup an inheritance hierarchy in EF is the ease of use. In EF it is a bit awkward to load all of your base tables and then rearrange fields into different entity representations until you have built a new entity object. All fields in the new inherited entity must be mapped back to the correct base table. Using nHydrate, you simply define a table and add fields to it. Then you can define another table representing an inherited entity and add the additional fields it will contain. To define the inheritance simply set the child’s table parent table property to the parent table. Create a relationship between the two and you are done. It is that simple.

All stored procedures that need to perform CRUD functions on all tables (there can be N-level inheritance) is generated automatically. Dynamic SQL is not created for standard CRUD operations. This is all implied and there are no complex mappings to complete this operation.

## Entity Splitting

EF employs entity splitting that is used to implement inheritance. This is a manual and cumbersome process that splits fields between tables, maps relations, and creates new entities. This entire process is replaced in nHydrate by the parent table property of each table. Using this property and a relationship you to define complex, inheritance hierarchies with no other setup required.

## Data Concurrency and auditing

If you do go through the trouble in EF to define stored procedures manually and then map them to the proper entity fields, there is still the issue of concurrency. nHydrate has concurrency backed in from the start. In fact you have to opt out of using concurrency as it is default functionality. All tables by default will have a created and modified audit fields that allow you to track changes. There is also a hidden timestamp field that is used for concurrency. Data cannot be saved to the database if its concurrency fails. This allows you to use the much more efficient optimistic concurrency methodology to synchronize your database.

In addition to the above functionality there is also a feature to track all row changes across all time. You can define a table to save a copy of the row before each save or delete. This allows you to have a snapshot of all changes for all time. Actually, all addions, modifications and deletions are logged to a shadow table not present in the model that you can query to find any modifications to a row. This extends even to deleted rows. This functionality can be used for data verification, backups, record keeping, etc.

## Syntax

nHydrate employs a more optimized syntax for manipulating entities. In EF you must first declare your model then use it to select the items you want. nHydrate uses static methods to return needed data in one line of code. Also the collection object returned has many convenience functions directly accessible on it. There where syntax is similar in both in that they use LINQ. EF can use inline LINQ as in the example below. nHydrate always uses a parameter where statement though it is still LINQ.

|  |
| --- |
| //Entity Framework MyModel model = new MyModel();  var q = from x in model.Customers  select x;  //nHydrate  CustomerCollection customerCollection = CustomerCollection.RunSelect(); |

When dealing with an inherited object in EF that uses the type per hierarchy setup, there is the additional syntax of adding the “OfType” keyword. This is a cumbersome syntax when compared to the nHydrate syntax of just specifying the child table in the exact same fashion as seen above.

nHydrate also contains much aggregate and bulk update functionality. There is no corresponding functionality in EF. This allows you to perform complex actions with one line of code most of the time.

|  |
| --- |
| //Bulk Update the Customer.City to the  //new value 'Berlin' where Customer.FirstName = 'Sally' CustomerCollection.UpdateData(  x => x.City,  x => x.FirstName == "Sally",  "Berlin");  //Delete all rows where Customer.FirstName = 'Sally' CustomerCollection.DeleteData(x => x.FirstName == "Sally"); |

## Database creation and updates

There is a built in generated project for database creations and updates. This allows you to create a database right from the VS.NET environment or include the functionality in a separate application installation application if desired. This functionality also extends to the database update. All databases are versioned so you can define update scripts that convert lower to upper versions. The upgrade scripts are mostly automatic so that if you add or delete tables or fields, the appropriate modification scripts are created for you. There are some exceptions for complex changes of course but for the most part this works just fine. The versioning allows you to have multiple databases in your enterprise for a model and upgrade them one at a time as needed and all upgrades are handled by the installation application automatically. This is a godsend for anyone managing a multiple version database deployment.

## Complex entities

Because the nHydrate model has metadata on it you can define compound functionality of entities. For example, entities can be defined as immutable such that they cannot be modified or persisted. More useful is the ability to define type tables. These are entities with no other purpose than to be discriminators for other tables. This might include a user type if you have different types of users like customer, employee, manager, etc. They are all in your user table but you need a type to discriminate. Type tables not only contain static data and are immutable but they generate an enumeration so you never use “magic numbers” in code. All related tables have the foreign key field of course but also an enumeration field that can be set with a strongly-typed enumeration setting.

|  |
| --- |
| //Set the customer type with enum NOT the actual user type primary key  customer.UserType = UserTypeConstants.Customer; |

Another complex entity is the associative table. This links two tables that participate in a many-to-many relationship. Since relational databases do not handle N:N relations you must create an intermediary table to facilitate this relationship type. nHydrate complete handles this mapping for you. You simply add the intermediary table to your mode and define the field and relations to it. In the generated code you will never see the table. Table 1 with have a list of table 2 objects and table 2 will have a list of table 1 objects. The associative table is not visible. This is a serious improvement in the way standard ORM tools handle this situation.

## Paging

Both platforms support pagination. The syntax for Ef is a bit more cumbersome than the nHydrate syntax. nHydrate supports strongly-typed pagination obejcts to fully define how to paginate a resultset. This is where the platform really shines. Paging is built-in from the ground up. It is designed to be paginated with a simple syntax. The code snippet below shows how to retrieve a paginated resultset sorting by one field. You can of course sort by any number of fields with either framework.

|  |
| --- |
| //Entity framework - cumbersome syntax  int pageNumber = 1;  int pageSize = 20;  using (var model = new MyModel())  {  //Get the recordSet  var recordSet = (from x in model.UserAccount  where x.UserId > 100  orderby x.UserId  select x). Skip((pageNumber - 1) \* pageSize).  Take(pageSize);  }  //nHydrate - declare a paging object and load data  UserAccountPaging paging = new UserAccountPaging(1, 20, UserAccount.FieldNameConstants.UserId, true);  UserAccountCollection recordSet =   UserAccountCollection.RunSelect(x => x.UserId > 100, paging); |

## Components

With EF you can use entity splitting to create a subset of a table such that you need not load all data from a database row. This is especially useful for legacy systems with wide tables or tables with blob fields or images. You must create an entity and remove the excess fields or map new ones to the entity. All the complexities of field mapping still apply. Using nHydrate, you can select a table and right click to create a Component object under it. Then select the fields for the component from the provided list and give the object a name. That is all there is to creating a table component.

## Persistence

A really good example of the way the two platforms are designed is in how they save data. EF gives you flexibility in how you handle its low level objects and you can write wrappers or extensions around their functionality to hide the intricacies of their handling. But of course that is more code to write. nHydrate has a simple yet robust container that allows you to load, change, and persist changes to a database in a few lines or even one line of code.

Below is some code I took from the Microsoft website. It demonstrates how to get an order record and add a new order detail item to it then save all changes.

|  |
| --- |
| SalesOrderHeader order =  context.SalesOrderHeader.Where  ("it.SalesOrderID = @id", new ObjectParameter(  "id", orderId)).First();  // Change the status and ship date of an existing order.  order.Status = 1;  order.ShipDate = DateTime.Today;  // Load items for the order, if not already loaded.  if (!order.SalesOrderDetail.IsLoaded)  {  order.SalesOrderDetail.Load();  }  // Delete the first item in the order.  context.DeleteObject(order.SalesOrderDetail.First());  // Create a new item using the static Create method  // and add it to the order.  order.SalesOrderDetail.Add(  SalesOrderDetail.CreateSalesOrderDetail(0,  0, 2, 750, 1, (decimal)2171.2942, 0, 0,  Guid.NewGuid(), DateTime.Today));  // Save changes in the object context to the database.  int changes = context.SaveChanges(); |

Using nHydrate is similar but the selection syntax is a bit simpler. Also the child object creation is more straight forward and easy to read.

|  |
| --- |
| //Load the order by ID  Order order = OrderCollection.RunSelect(x => x.OrderId == orderId)[0];  SubDomain subdomain = order.ParentCollection.SubDomain;  // Change the status and ship date of an existing order.  order.Status = 1;  order.ShipDate = DateTime.Today;  // Delete the first item in the order.  order.OrderDetailList[0].Delete();  // Get a reference to the OrderDetail collection in the same container OrderDetailCollection orderDetailCollection = order.ParentCollection.SubDomain.GetCollection<OrderDetailCollection>();  OrderDetail orderDetail = orderDetailCollection.NewItem(); orderDetail.OrderId = order.OrderId;  orderDetail.Prop1 = 0;  orderDetail.Prop2 = 2;  orderDetail.Prop3 = 750;  //etc...  orderDetailCollection.AddItem(orderDetail);  // Save changes in the container to the database.  subdomain.Persist(); |

## Dependency Walking

Walking relationships is also where the nHydrate platform shines. Based on the model entities and relationships you can walk the object graph. EF provides the same functionality of course but you do have to specifically tell it to load the objects.

|  |
| --- |
| using (var model = new MyModel())  {  //Get the recordSet  var recordSet = (from x in model.UserAccount  where x.UserId > 100  orderby x.UserId  select x);  foreach (UserAccount userAccount in recordSet)  {  userAccount.Transaction.Load(); //Explicitly load  foreach (Transaction t in userAccount.Transaction.ToList())  {  System.Diagnostics.Debug.Write("");  }  }  } |

When using nHydrate, walking relationship can be done with a little simpler syntax.

|  |
| --- |
| //Get the recordSet  UserAccountCollection recordSet = UserAccountCollection.RunSelect(x =>  x.UserId > 100);  foreach (UserAccount userAccount in recordSet)  {  foreach (Transaction t in userAccount.TransactionList)  {  System.Diagnostics.Debug.Write("");  }  } |

Both allow you to load an object and then just walk up and down the relation graph to lazy load the hierarchy. All stored procedures needed to provide this fuctionality are generated as part of the framework. Everything is truly transparent. There is nothing to setup as all walking is determined by the entities in the model and the relationships between them. If you change your model and regenerate all the new walking code is generated for you. All of the stored procedure plumbing is added to the installation project automatically and available when you upgrade your database.

## Associated Projects

nHydrate gives you many projects to construct an application right out of the gate. Not only is there s a data access layer (DAL), which this article is addressing but also a data transfer layer (DTO) and other Inversion of Control (IoC) projects to build complex application hierarchies with no additional framework code needed. Currently there are ten projects that can be generated with the platform and these can be used to build application code immediately.

## Domain Layer Independence

The IoC patterned layer allows many different types of domains to be loaded. This makes it possible to develop web, windows, or mobile applications on the same code base. All development can be done against the DTO layer and backend can be swapped out for DAL (direct connection), WCF, or mocking layer. The default IoC proxies that comes with nHydrate mean you have a standard API to retrieve data. If a more specific or custom backend is needed, you can write a proprietary generator based on EF or nHibernate.

Although the current implementation of nHydrate is written on top of ADO.NET objects, it could be used to write a generated layer on top of EF if desired. This is a matter of designing a standard pattern and creating a template from it. The current code generation templates were constructed in this same way. This is the actual purpose of EF, to act as a foundation of other more specific, application frameworks. The goal for nHydrate moving forward is to provide multiple backend data access technologies interchangeable on a robust API.

## Miscellaneous

A small detail that is not wrong or right but is just convenient is the way pluralization is handled. The current nHydrate pluralization scheme simply adds the word “List” to related collections. This serves many purposes. First it is standard so when you have hundreds of entities you do not need to think about whether an object is plural or singular. You very quickly get used to the fact that all “Lists” can be enumerated. Secondly this alleviates the need for you to manually set the pluralization for all entities in your model. Thirdly, as you rename entities as happens in application development, you do not need to remember to change two properties each time to keep them in sync. Lastly, consistency is important. A Person table might be pluralized People, Persons, PersonList, PersonCollection, PeopleList, PeopleCollection, etc. If you have hundreds of entities in your model and a many developers, there is bound to be inconsistent naming if pluralization is ad hoc.

## Summary

There are many scenarios where EF will be the platform of choice but there is a learning curve to setup your application framework correctly and efficiently. This is not trivial. EF is great at database mapping but graphical design, but the default features are a sparce. There is a lot of code you will need to write to get a usable framework up and running. nHydrate tries to alleviate this issue by generating code with many of the convienences that you will need by default. EF is an extensible platform that allows you to build your own framework on top of it. This is a different methodology than that taken by nHydrate, which tries to get you coding sooner with a smaller learning curve. Both platforms address the repetitive database code that is the bane of all developers. However EF gives you a platform to build a framework and nHydrate gives you a framework already built.

# Part II Chapter 7 – Extend Generator Framework

The text so far has addressed how to use what nHydrate generates out of the box. However nHydrate is more than a canned set of templates. You can extend the generative framework by writing you own generators of extending those that already exist. You can create a new assembly that can be plugged directly into nHydrate and show up in the generate menu. This of course presupposes that you are creating a generator for a new project. However you can simply create a new file generator for an existing project.

### Overview

Before getting started with this walkthrough you should spend some time getting to know nHydrate. A good starting point is the article <http://www.codeproject.com/KB/dotnet/NHydrateStep-by-Step.aspx>.

nHydrate is a general purpose generation platform that comes with many out-of-the-box templates for creating simple and complex .NET applications. As shipped it can create data access layers (DAL), data transfer layers (DTO), and implement an inversion of control pattern (IoC). This functionality can get you up and running building an application very quickly. However you may want to extend the standard templates and create your own extensions. This is an easy feat and will be addressed in this text.

This text will address how to extend the existing DTO layer of the nHydrate templates; however you can also create your own project as well. The DTO project already exists. We will just add another generator to create an extra file type. These newly generated files will be added to the DTO project that nHydrate creates when the user requests to generate this project type. Not covered in this article but still useful is creating a generator that creates a completely new project in the VS.NET solution explorer.

In this text we will extend the DTO generator. The extension will include creating two templates that generate new classes into the DTO project. The templates will work together to produce a single class for each table defined in the model. We are using two templates to demonstrate a common pattern by which we generate out a single class into two files using the partial class methodology. One file is generated to handle custom class extensions while the other file is generated and should not be customized.

### Create a Project

To get started we will create a new class library project Widgetsphere.GeneratorWalkthrough. The project will need to have references to.

* **Widgetsphere.Generator.Common** - This holds the interfaces and attributes that make up the core generation
* **Widgetsphere.Generator** – Contains the model classes as well as base classes that make creating a new template easier.
* **Widgetsphere.Generator.DataTransfer** – Since we are extending the DTO generator in this example we will be creating new classes in the DTO project. We will use this reference to DTO objects to do this.

**Click on the project properties and highlight the build events. To make it easy to deploy, set the post-build event as follows. Keep in mind that this is the default installation location for a 32-bit machine. If you are using a 64-bit machine or installed in a custom folder, you will need to use the correct path.**

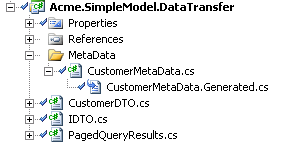
|  |
| --- |
| **copy $(TargetDir)$(TargetName).\* "C:\Program Files\Widgetsphere\CodeTools\\*.\*"** |

### Create Generator and Template Class

By convention we separate the file creation from the file content into two classes. The file creation is specified in the generator class. This class has two major requirements. Inherit from ***BaseGenerator*** and apply the ***GeneratorItemAttribute*** to the class.

|  |
| --- |
| /// <summary>  /// This generator will generate the [TableName]MetaData.cs file. This file  /// contains a public partial class [TableName]MetaData. For the generated  /// file the class will be empty. This file is provided as a way to allow programmers to  /// extend the generated class. It will be generated once *and never over written after that*  */// point.*  /// </summary>  [GeneratorItemAttribute("MetaData Class Extension", typeof(DataTransferProjectGenerator))]  public class MetaDataInfoClassExtensionGenerator : BaseGenerator  /// <summary>  /// This generator will generate the [TableName]MetaData.generated.cs file. This file  /// contains a public partial class [TableName]MetaData. This generated  /// file will contain the core implementation of the [TableName]MetaData. It will be  /// overwritten each time a generation occurs. It should not be extended by programmers.  /// </summary>  [GeneratorItemAttribute("MetaData Class", typeof(MetaDataInfoClassExtensionGenerator))]  public class MetaDataInfoClassGenerator : BaseGenerator |

Let us first discuss the ***GeneratorItemAttribute***. The generator engine searches for all classes that expose this attribute. Once it has found this class it prepares loads it into a list of classes that should be called during the generation process. The first parameter in the attribute is simply the name that should be applied to this generator. The second is a little more complicated. It is used to identify a dependent generator. The generation architecture uses this information to call the generators in the appropriate order. In this example you will notice that ***DataTransferProjectGenerator*** is the predecessor to ***MetaDataInfoClassExtensionGenerator*** and ***MetaDataInfoClassExtensionGenerator*** is the predecessor to ***MetaDataInfoClassGenerator***. The reason for this is very apparent when you see the generated results. You must have a visual studio project ***Acme.DataTransfer*** before you have an item associated with it like **CustomerMetaData.cs** and you must have the item **CustomerMetaData.cs** before you can create a sub item **CustomerMetaData.Generated.cs.**



The ***BaseGenerator*** is an abstract implementation of the interface ***IProjectItemGenerator*** this interface is used by the core generator engine to add and overwrite files through the VSCodeTools Add-In. ***BaseGenerator*** was created to hide the complexities of implementing this class. When overriding this class, there are two abstract properties *FileCount* and *ProjectName* and one abstract method *Generate* that must be implemented. The properties are very straight forward. The first *FileCount* is provided to give the generator a heads up on how many project items the generator can expect to be creating. This is provided to the user interface for presentation at various points in the generation process. The second is *ProjectName* which is used to identify the project that the files should be placed in. Finally there is the *Generate* method, which is called by the generation engine. As part of the implementation of this method the user is expected to raise the ***ProjectItemGenerated*** and ***ProjectItemGenerationComplete*** events. The generation engine is monitoring these events and uses the event arguments to identify what to create within the appropriate Visual Studio project.

|  |
| --- |
| /// <summary>  /// Method called by the core generator. This method is used to  /// manage what items are to be generated by the system.  /// </summary>  public override void Generate()  {  //Walk through all the tables in the model.  //Generate classes for those that are specified to be generated.  foreach (Table table in \_model.Database.Tables.Where(x => x.Generated).OrderBy(x => x.Name))  {  //Construct the template class passing it the current model file and table  MetadataInfoClassExtensionTemplate template =   new MetadataInfoClassExtensionTemplate(\_model, table);    //Create a string that represents the project relative  //output location of the file (Ex: \MetaData\Customer.cs)  string projectRelativeFileName = RELATIVE\_OUTPUT\_LOCATION + template.FileName;    //Get the generated file content.  string fileContent = template.FileContent;    //Publish the Project Item Generated event for the generation engine to handle  ProjectItemGeneratedEventArgs eventArgs =  new ProjectItemGeneratedEventArgs(projectRelativeFileName,   fileContent, ProjectName, this, false);  OnProjectItemGenerated(this, eventArgs);  }  //Publish the generation completed event to identify to the   //engine all files have been generated.  ProjectItemGenerationCompleteEventArgs gcEventArgs =   new ProjectItemGenerationCompleteEventArgs(this);  OnGenerationComplete(this, gcEventArgs);  } |

### Create Template Class

The file content is generated from the template class. This class will inherit from ***BaseDataTransferTemplate***. By convention generation project will contain a template base class that aids other developers with methods that make creating a new template easier.

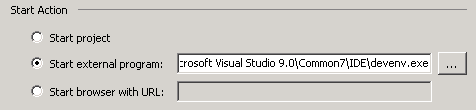
|  |
| --- |
| /// <summary>  /// This template will generate the content for the [TableName]MetaData.cs file. This file  /// will contain a public partial class [TableName]MetaData. This file is  /// provided as a way to allow programmers to extend the generated class.  /// It will be generated once and never over written after that point.  /// </summary>  public class MetadataInfoClassExtensionTemplate : BaseDataTransferTemplate  /// <summary>  /// This template will generate the content for the[EntityName]MetaData.generated.cs file.  /// This file contains a public partial class [EntityName]MetaData. The file will contain  /// the core implementation of the [EntityName]MetaData. It will be overwritten each time  /// a generation occurs. It should not be extended by programmers.  /// </summary>  public class MetaDataInfoClassTemplate : BaseDataTransferTemplate |

We generate this class to produce the *FileName*, *FileContent* and when necessary the *ParentItemName* for all files that need to be added to the project.

|  |
| --- |
| /// <summary>  /// Returns the name of the file. That is to be added to the project  /// </summary>  public override string FileName  {  get { return string.Format("{0}MetaData.Generated.cs", \_currentTable.PascalName); }  }  /// <summary>  /// Returns the a parent item name. This is only necessary when the  /// parent item is another file in the project  /// </summary>  public string ParentItemName  {  get { return string.Format("{0}MetaData.cs", \_currentTable.PascalName); }  }  /// <summary>  /// Returns the generated contents of the file as a string  /// </summary>  public override string FileContent  {  get  {  GenerateContent();  return sb.ToString();  }  } |

### Lets Test

To test, we will setup the Widgetsphere.GeneratorWalkthrough project to start another instance of Visual Studio. In the project properties, highlight the Debug tab. Next select the “start external program” option and enter the path to the Visual Studio application.



*IMPORTANT: It is very important to realize that Visual Studio will lock the files under the nHydrate install folder as soon as a model is opened. Once these files are locked you will not be able to build this project. For this reason it is suggested that only one instance of Visual Studio that has opened the generation extension project is running. Once you start the debug a second instance will be started. This instance will be used to open a model file and start the generation process.*

Now that the debug information is setup we can start debugging. Press F5 and the new instance of Visual Studio will start up. In the example, there is another solution TestCustomTemplate.sln, this solution contains a very simple model file that you can generate from. Open the model file and start a generation. Put a breakpoint on the *Generate* of the *MetaDataInfoClassExtensionGenerator* class. Run the Generate All command making sure that the DTO project is selected for generation. Once a generation is started, the breakpoint is hit and you can walk through the code.

After generation, you can see that there is a DTO project. This is created by the standard DTO project generator that comes with nHydrate. However you now see that there is a MetaData folder in the project. In the folder, you can see that there is a set of files for each database table in the model. There is a user-modifiable class named *[Table]MetaData.cs* and a dependent generated file named [Table]MetaData.Generated.cs. The user can make modification in the former file and the changes will not be overwritten. However the latter file is maintained by the generator and will be overwritten on every generation. A user should never modify this file since the changes will not be preserved. You will also notice that in the generated file there is a method that returns all database field names and a property for each of the table’s properties.

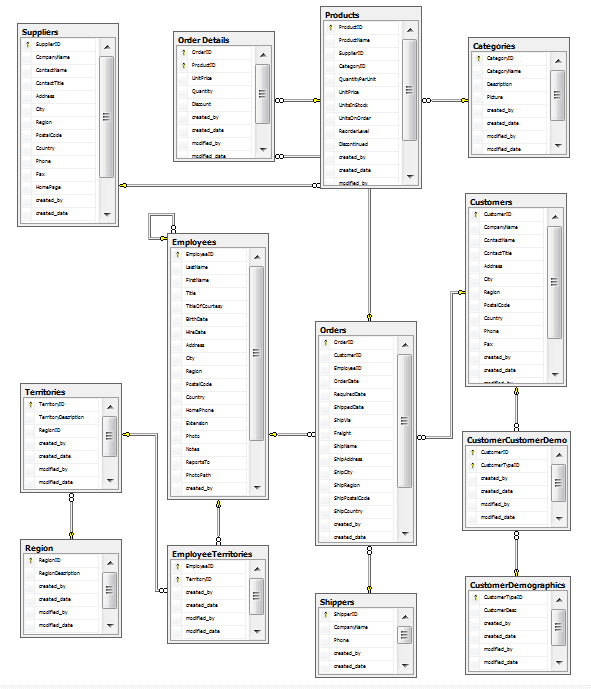
Using this technology is truly powerful in that you can extend existing generators and customize the nHydrate framework to create anything you needed for your custom application.

# Part III Chapter 8 – Acme Sales Example

Just like a picture is worth a thousand words an example is worth a thousand pages. It is easier to look at what others have done than to read a detailed description. The Acme Sales application can be downloaded from nHydrate CodePlex site under downloads.

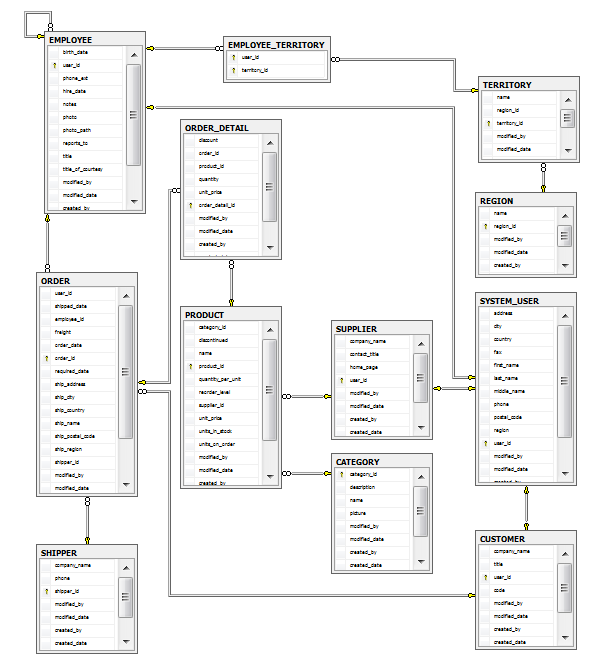
The Acme Sales application is a port of the old Northwind database we are all so familiar with. Northwind was redesigned (and renamed Acme Sales) to show off some more advanced nHydrate functionality and to be a little more real-world in design.

The original database schema has a string primary key for customers. It also has an intermediary table that is really a type table. In addition there are some tables that really should be inside of an inheritance hierarchy. These and other functionality points have been changed or extended. The following screen shots show the original Northwind database and the new Acme Sales database.



The new database schema has all tables assigned an auto incrementing integer as a primary key. A new table “SYSTEM\_USER” has been added to be a base type for all users. In the original design Customer, Supplier, and Employee were very similar. They are after all users or at the very least people/entities. The base table holds all common properties and the other “derived” tables hold the properties specific to each implementation. Of course database tables cannot inherit from each other, but since we are using a model driven architecture this conceptualization is handled by the framework.

This inheritance hierarchy defines a one-to-one relation between base and child table. The primary key of one table maps to the primary key of the other. This is all handled by the model. In the model, I create the base entity and add all common properties to it. I then create child entity and add the properties specific to its implementation. I do not need to add the common properties like first and last name to the child table. To “derive” a child entity from a base, simply set the child entity’s parent table property. This signifies to the model that we are inheriting. The verification will check to ensure all necessary rules are met for this functionality to work properly. This is important since this is a specialization the database table structure.



# Conceive, Model, Generate!