# 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.