nHydrate 5

Getting Started

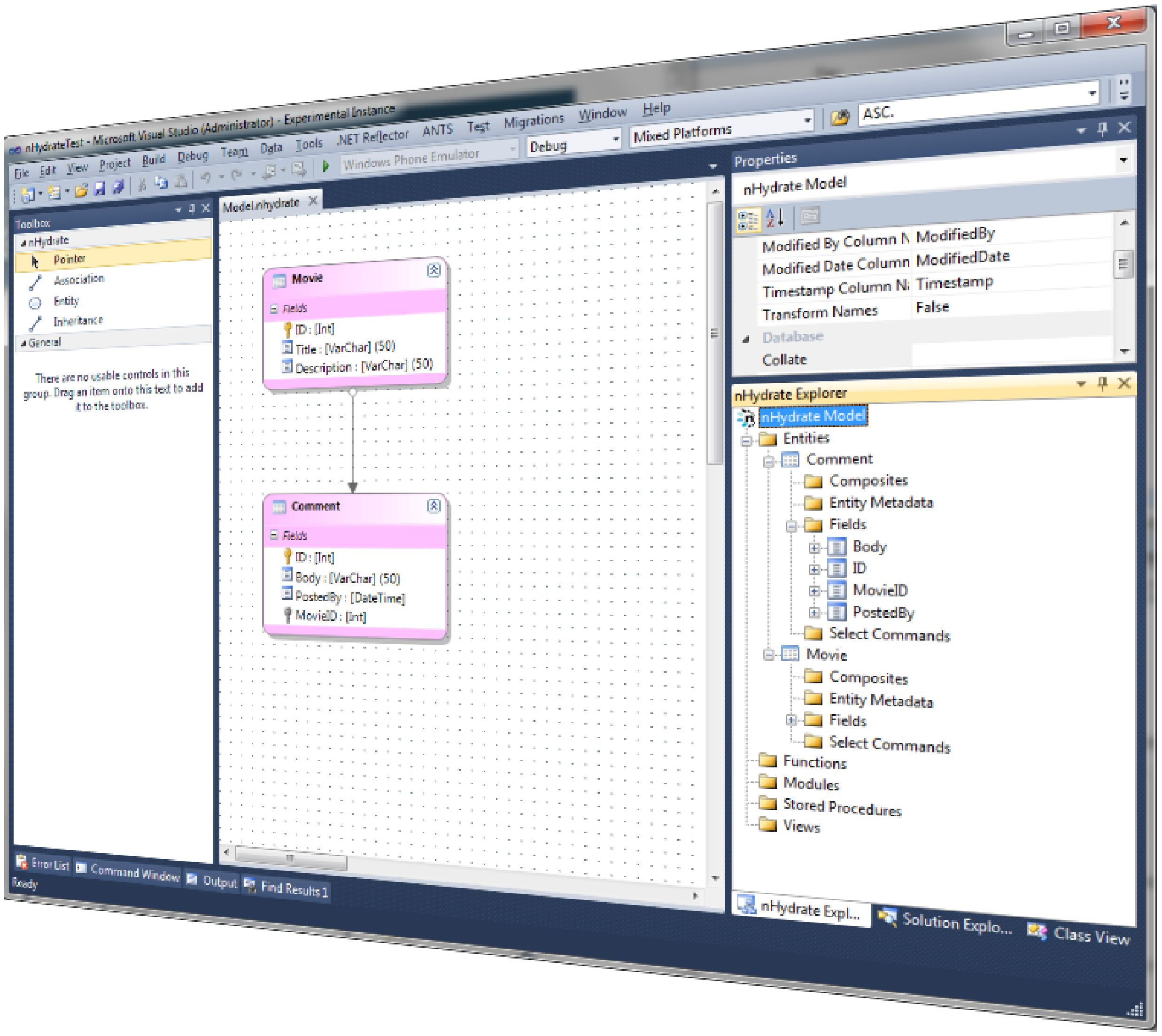


Table of Contents

Getting Started with nHydrate

Welcome to the Quick Start walkthrough for nHydrate. This document will introduce you to the core concepts, classes and techniques of nHydrate. We will look at how to create a domain model, load your domain data and make changes to that data and save it back to your database.

In this walk-through, we will be using the nHydrate Visual Studio Designer which takes care of most routine tasks for you so you will need to have installed nHydrate prior to embarking on this document.

About nHydrate

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

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

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

Objects and Databases

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

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

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

nHydrate as an object-relational mapper

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

Getting Started

Let’s get started with nHydrate by:

- Building up a simple domain model
- Running some queries
- Making changes to entities

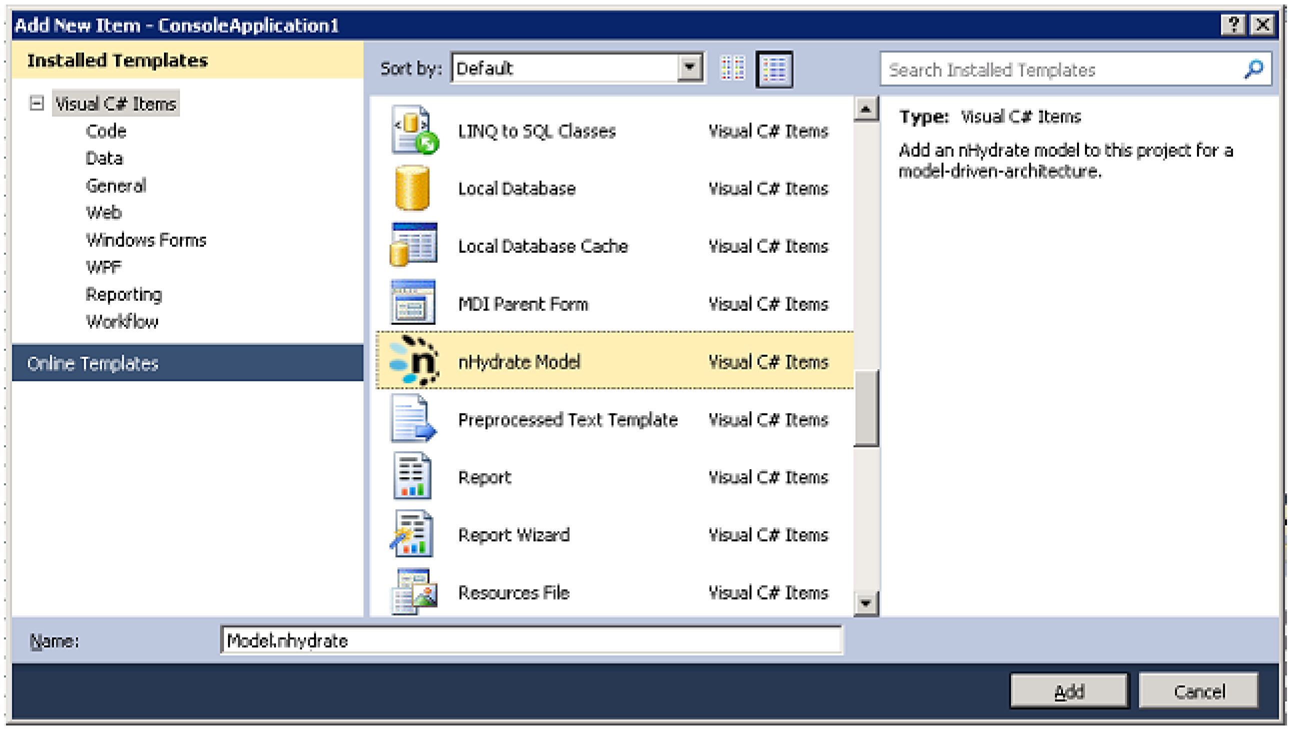
This example will assume you are building using the C# language and are targeting a .NET 4.0 based solution. This is necessary because we are using Entity Framework 4.0 and this is a minimal framework required.

Building Your First Domain Model

To create your first domain model, start by creating a new project within Visual Studio. For the purposes of this example we will create a ConsoleApplication to house the model, but you can add it to any project. It is important that you click the check box “Create directory for solution”. This will add the console application to its own folder. The reason for this is that the model is going to generate new projects in this solution. The projects will be created off of the solution folder.

The next step is to create a nHydrate model. To do this, Add a New Item to the project and then select nHydrate Model from the list of available item templates. Enter an appropriate name to describe the domain for this model, or if you are only likely to use a single domain model for the entire application the standard is to name this Model.

In this example, we will create our model using the name Model, which will generate a new file called Model.nhydrate within our project.



Sort by: Default

LINQ to SQL Classes	Visual C# Items
Local Database	Visual C# Items
Local Database Cache	Visual C# Items
MDI Parent Form	Visual C# Items
nHydrate Model	Visual C# Items
Preprocessed Text Template	Visual C# Items
Report	Visual C# Items
Report Wizard	Visual C# Items
Resources File	Visual C# Items

Type: Visual C# Items

Add an nHydrate model to this project for a model-driven-architecture.

Name: Model.nhydrate

Modelling

nHydrate supports two ways in which you can elaborate your domain model. You can either start by describing your domain model prior to creating your database (this is known as Model First), or by starting with an existing database and using that to create your initial model (this is known as Database First).

It is recommened that you start with a model first and let the nHydrate generated installer project create your database. Of course this may not be possible if you are upgrading a legacy system. You can import an existing database as a starting point and from that point on manage your database with the nHydrate installer.

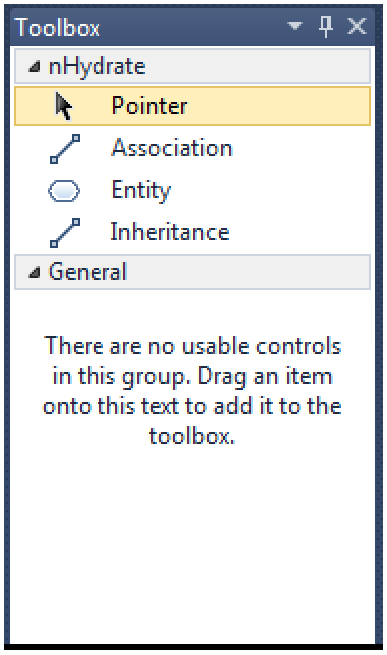
For the purposes of this example, we will be designing a sample which has 2 entities, a Movie and a Comment. There is a relationship between Movie and Comment in that a movie may have one or more comments about it.

We will model this in nHydrate using a Model First approach which means we will use the nHydrate designer to describe our model and then commit this to a database. You could equally achieve the same goal by using the Database First approach as well.

Designing the Model using the Visual Studio Designer

Designing your domain model using the Model First approach is simple using the nHydrate Design Surface. To get started, double click on the Model.nhydrate file which opens up the design surface in Visual Studio.

Expanding the Toolbox pane gives you access to the objects with which you can use to model. We will start by dragging on 2 entity shapes, one for each of the domain entities we described above.

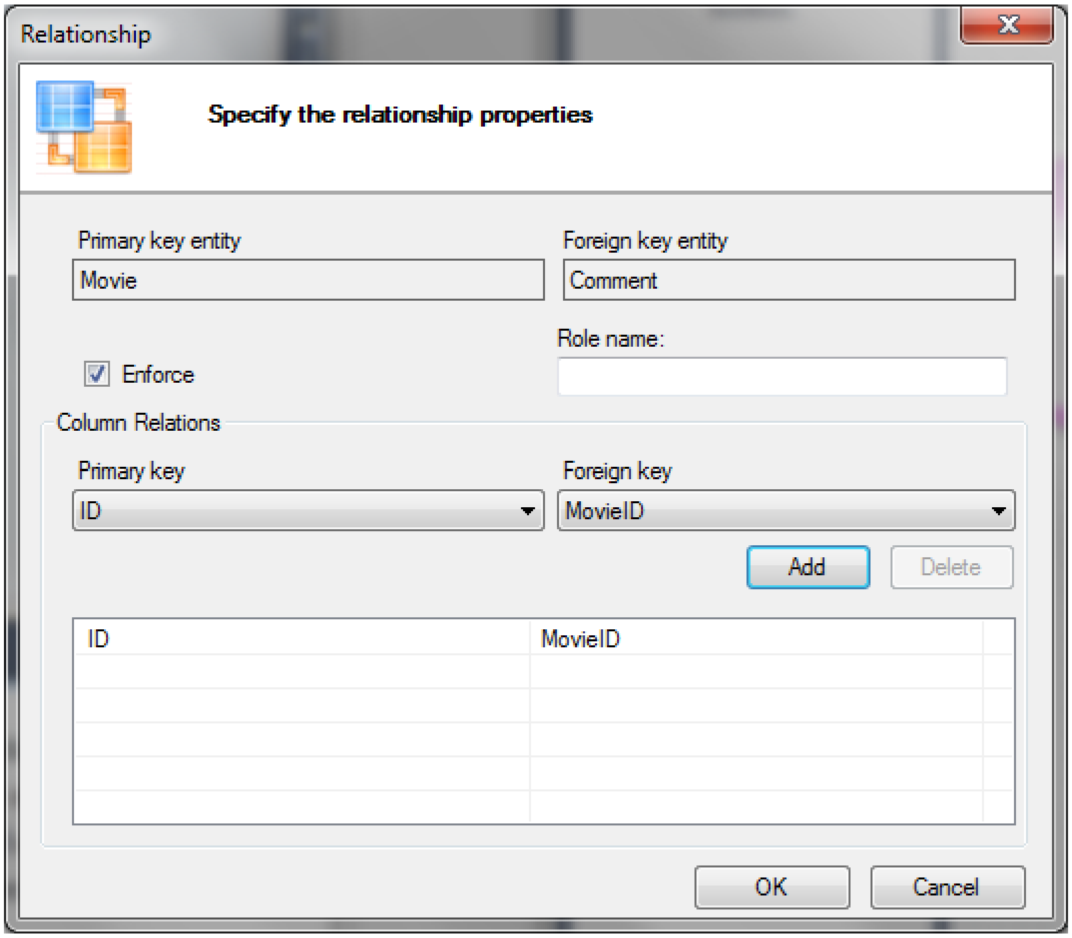


There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

After you drag these entities on the canvas, you can their properties using the standard property dialog window that all .NET components use.

The next step is to add the fields to the entities. To Movie I have added Title and Description. They both default to varchar of length 50 and I have left them like this. Notice that Movie already has a primary key defined as ID that is a database identity. I have left the default name and settings here as well. On the Comment entity, in addition to the primary key already being added I add the Body and PostedBy fields as well. I want to create a link between these two entities so I add a new field MovieID to Comment. This will be the foreign key that links Movie to Comment.

Now we need to add the actual relation. Drag an association shape from the toolbox on to Movie and then link it to Comment, this will create a relationship between these entities. A relationship properties screen will appear allowing you to setup the fields that define this relation.



Primary key entity: Movie

Foreign key entity: Comment

Role name:

☑ Enforce

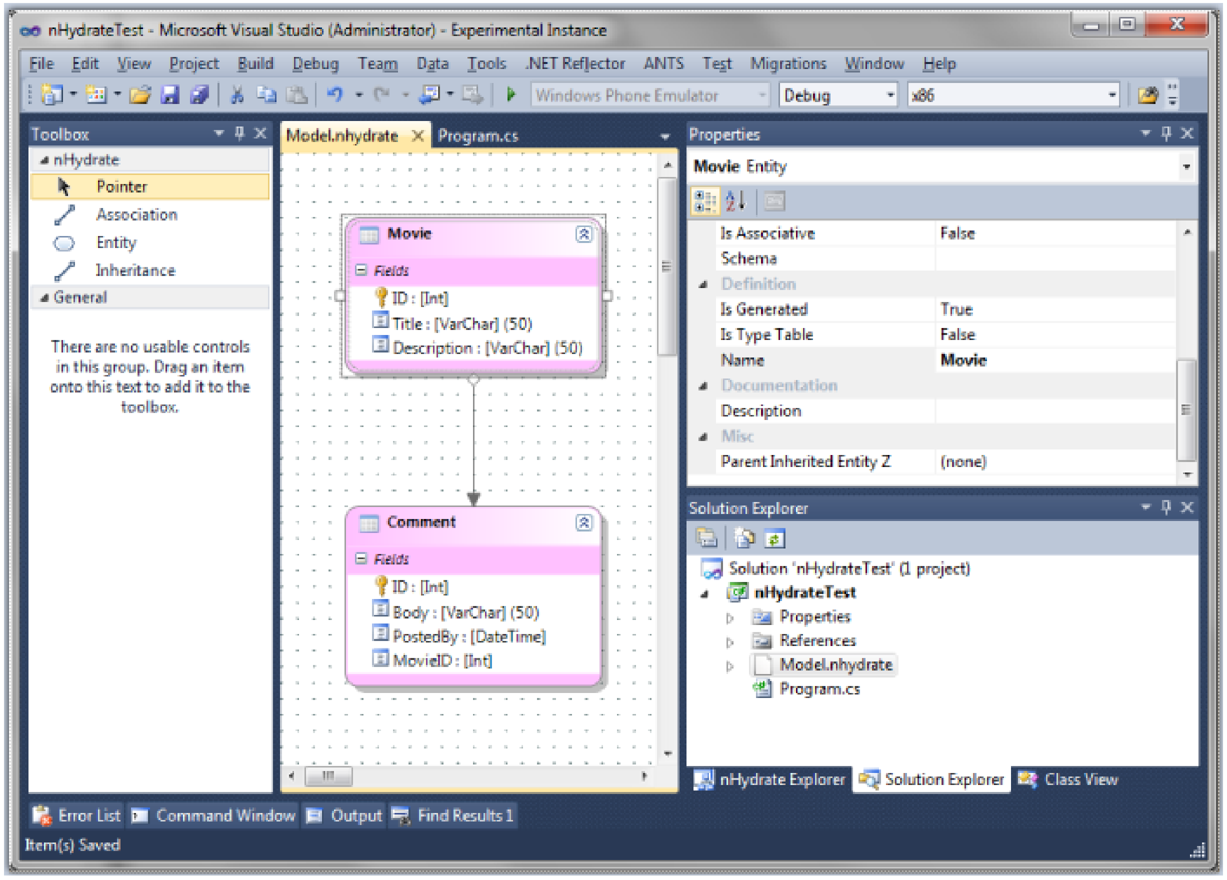
Column Relations

Primary key: ID

Foreign key: MovieID

ID	MovieID

After we have created the the two entities and defined the relation, the canvas should look like the following image.

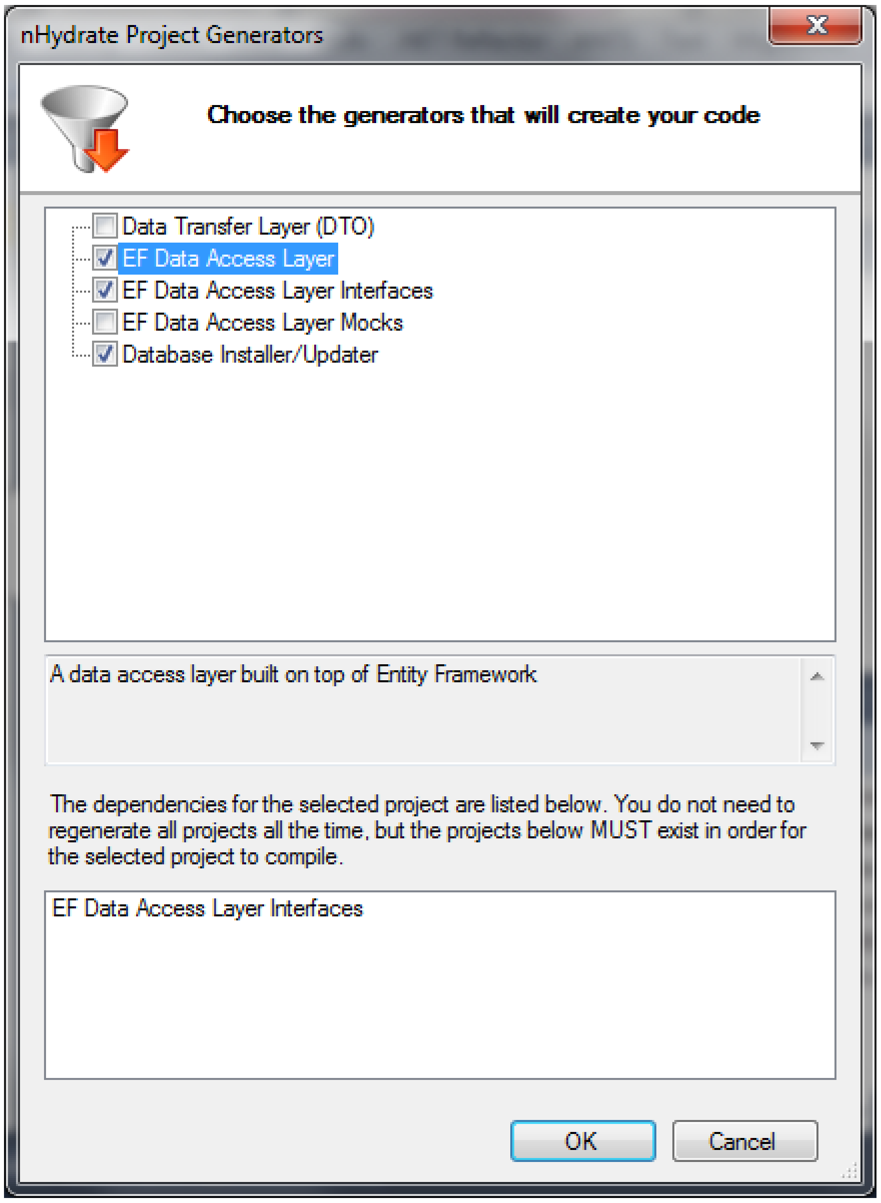


Is Associative	False
Schema	
Definition	
Is Generated	True
Is Type Table	False
Name	**Movie**
Documentation	
Description	
Misc	
Parent Inherited Entity Z	(none)

Now that we have defined a model, we can generate some code. There are some non-essential projects that you can create like mocks and data transfer objects; however for this example, we will generate the three bare minimal projects to manage your database and access these objects in code. These projects are the Entity Framework data access layer (EFDAL), the interface project that defines the entity objects, and the installer.

Before we proceed we need to set two more properties. Click on the canvas and you see the properties for the model. There is a company name and project name. These are used to define namespaces of the generated projects so they are required. I have set the company name to Acme and the project name to MovieTest. This will create a namespace in the EFDAL of Acme.MovieTest.EFDAL.

To generate, right click on the canvas free space to get a pop-up menu. Open the Model menu and choose the Generate option. You will see the following dialog.


nHydrate Project Generators
Choose the generators that will create your code

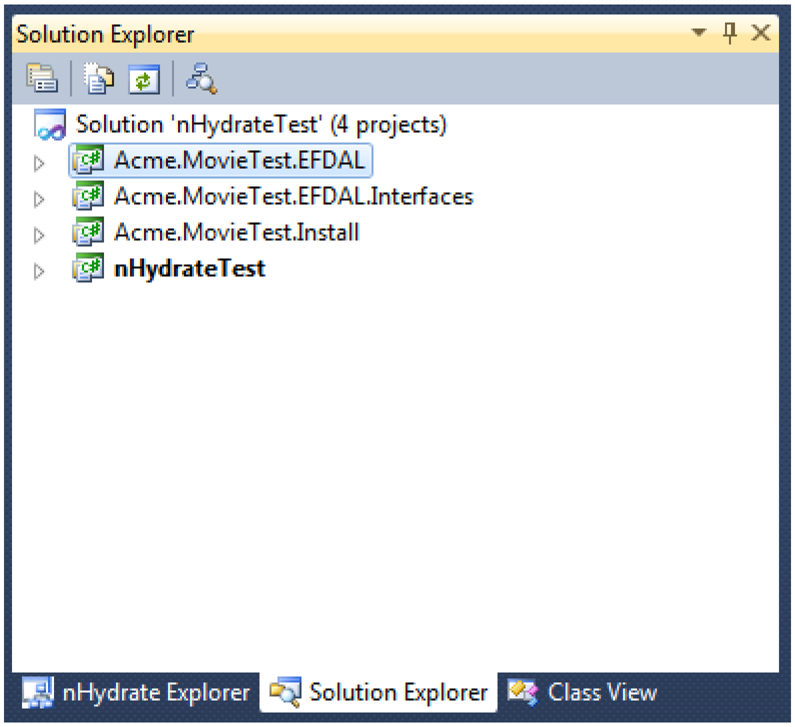
- [ ] Data Transfer Layer (DTO)
- [✓] EF Data Access Layer
- [✓] EF Data Access Layer Interfaces
- [ ] EF Data Access Layer Mocks
- [✓] Database Installer/Updater

A data access layer built on top of Entity Framework

The dependencies for the selected project are listed below. You do not need to regenerate all projects all the time, but the projects below MUST exist in order for the selected project to compile.

EF Data Access Layer Interfaces

After you generation is complete there will be three new projects in your solution. The solution explorer will look like the following image.

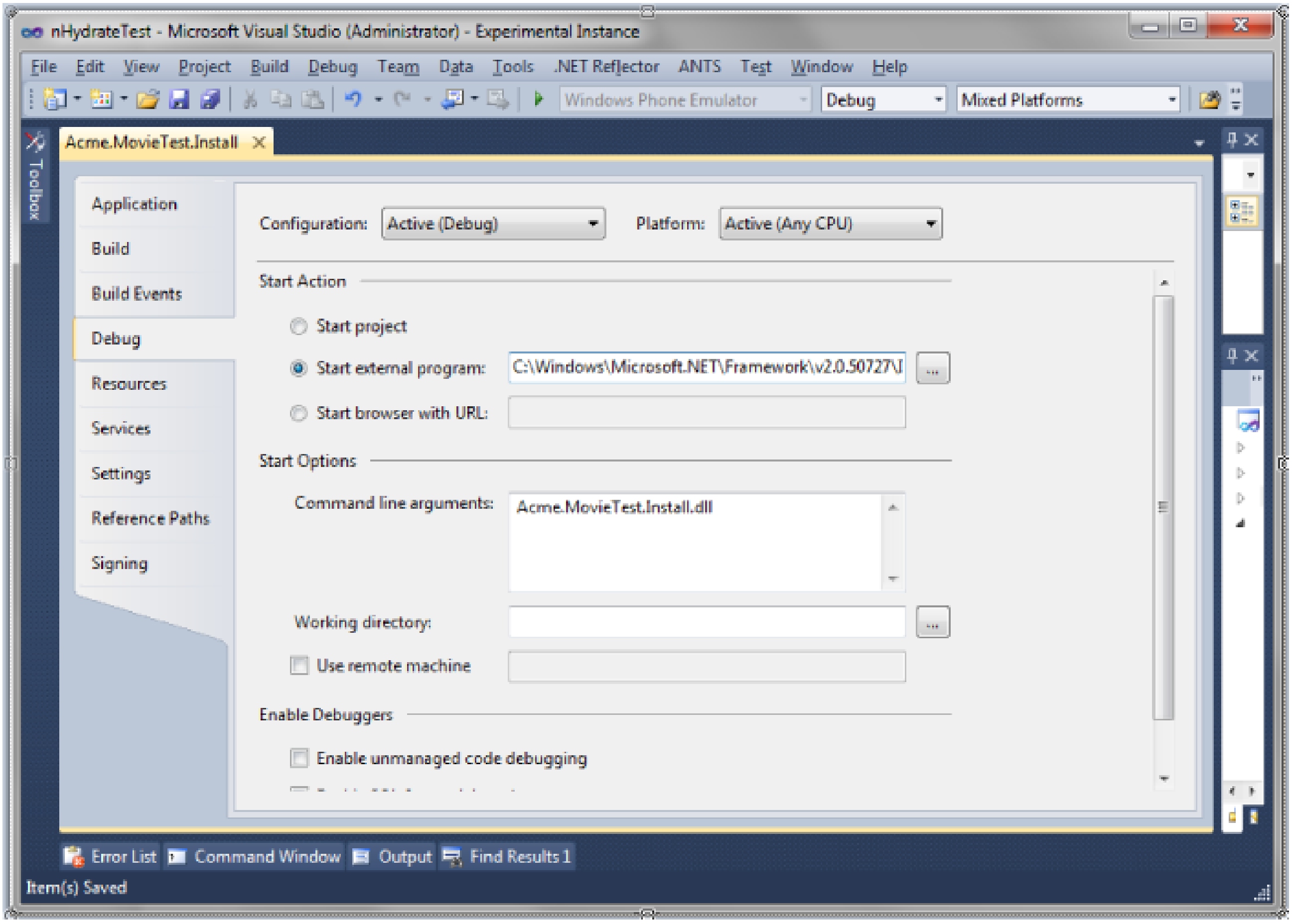


Solution 'nHydrateTest' (4 projects)
Acme.MovieTest.EFDAL
Acme.MovieTest.EFDAL.Interfaces
Acme.MovieTest.Install
nHydrateTest

Now to create a new database with this schema directly from Visual Studio, simply setup the installer project to run with the .NET tool InstallUtil.exe. Open the property window of the installer project and setup the debugging section. Setup the external program to be this application which on my machine is located at the following location.

“C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe”.

Afterwards setup the command line parameters to be output assembly which is Acme.MovieTest.Install.dll.



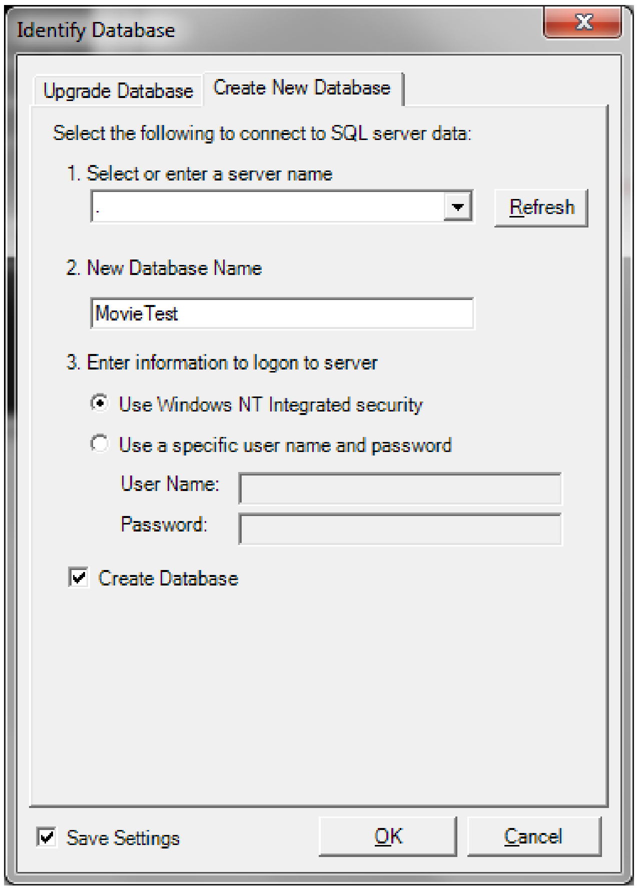
Configuration: Active (Debug)   Platform: Active (Any CPU)

Start external program: C:\Windows\Microsoft.NET\Framework\v2.0.50727\I

Command line arguments: Acme.MovieTest.Install.dll

Now all you need to do is right-click the project and click the Debug|Start New Instance menu. This action will run the InstallUtil application with the installer as target assembly and you will get a user interface to create or upgrade a database.

The following graphic shows the default UI for database interaction. If there is no existing database, you can use the Create tab to make one. If you are upgrading a database, then use the Upgrade tab.



Select the following to connect to SQL server data:

1. Select or enter a server name: .

2. New Database Name: MovieTest

3. Enter information to logon to server

   [✓ Use Windows NT Integrated security]

   Use a specific user name and password

   User Name:

   Password:

[✓ Create Database]

[✓ Save Settings]

This project can be used in your own custom application installation to upgrade a production database. The model tracks changes as you make them and it creates incremental upgrade scripts upon generation. After generation you may upgrade your database. It does not matter how long it has been since your last upgrade. The installer tracks the last database version and only runs the upgrade scripts necessary to bring the existing database version up to the latest level.

Code Generation

By using the nHydrate Designer you are actually generating code for the classes represented by your models. All nHydrate domain models are ultimately represented in C# code. The design surface provides the productivity boost and convenience in creating these, however you can craft your own custom code as extension of the generated code.

If you wish to extend these classes with custom behaviour or additional fields and properties, simply open the generated partial class. In the EFDAL project, there is an Entity folder. This houses all entities defined in your model. Notice that there is a partial gen-once file and a sub-file with the same name except that it is suffixed with the Generated keyword. This is a gen-always file. Never make changes in file that ends with generated as it will be re-created on each generation. In the partial class, you may add any extension functionality you wish. The changes will not be overwritten.

Selecting Data

Now that we have a a generated API, we can now interact with the database. First setup the solution dependencies. Right-click on the EFDAL project and select the Project Dependencies menu. Once open, check the interfaces assembly to notify the solution that it is a dependency. Now select the console application and setup the EFDAL project as a dependency. Press OK and build the solution.

In the console application add a reference to Widgetsphere.Core.dll as well as the EFDAL and interfaces assemblies in the Bin folder. Ensure that your console project is a Full 4.0 framework application and not a Client Profile application in the project properties sheet. Now we can create some Entity Framework database access code.

First we must setup the connection string. With Entity Framework this can be tricky, because they are quite complicated. I have chosen to use the App.Config file but you can set the connection string in code as well.

Sample connection string configuration block in the App.Config

```
<?xml version="1.0"?>
<configuration>

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

</configuration>
```

Querying

From this point forward the API is mostly Entity Framework. However there are many special features not addressed in this document. I will stick mostly to standard Entity Framework. The following code creates an Entity Framework context and selects all Movies from the database.

Query all movies

```
using (var context = new MovieTestEntities())
{
   var movieList = context.Movie.ToList();
   foreach (var item in movieList)
```

```
    {
            System.Diagnostics.Debug.WriteLine(item.Description);
    }
}
```

Updating

Of course, you can update objects as well. All entities can be updated unless they are marked Immutable. Immutable objects are a special case mainly for well-defined type tables or some other read-only data that never changes. The modifier are all private on these objects so a develop can never update these objects through the API. However all other entities can be modified.

The following code selects the first movie from the database and appends some text to the Description field. After the change the context's SaveChanges method is called to persist this change back to the database.

Update a movie

```
using (var context = new MovieTestEntities())
{
    var movie = context.Movie.FirstOrDefault();
    if (movie != null)
    {
            movie.Description += " and more...";
            context.SaveChanges();
    }
}
```

Deleting

Deleting objects is very standard as well. Simply call the context's DeleteItem method with a loaded entity object.

Delete a movie

```
using (var context = new MovieTestEntities())
{
    var movie = context.Movie.FirstOrDefault();
    if (movie != null)
    {
            context.DeleteItem(movie);
            context.SaveChanges();
    }
}
```

Of course this seems ridiculous to load an object from the database just to issue back a deletion. Once of the non-standard features that is not present in standard Entity Framework is issuing delete commands with LINQ without loading the objects in memory. This is also nice, because with it you could remove thousands or millions of objects with no adverse memory usage by your application.

Remove multiple rows without loading
Movie.DeleteData(x => x.Description.Contains("some data"));

There is a static method named DeleteData on each Entity type. You can issue a LINQ statement that is converted to a SQL statement with no object loading.

Summary

There are many features not covered in this document. The nHydrate system allows you to create Entity Framework APIs with a lot of additional functionality. The meta data defined in the model is used for binding, validation, and other advanced functionality. The visual modeler makes it easy to define and manage a complete data model.

Conceive, Model, Generate!