# nHydrate Templates

### 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.**

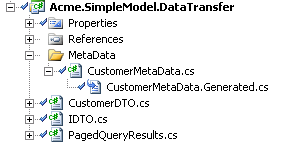
|  |
| --- |
| **copy $(TargetDir)$(TargetName).\* "C:\Program Files\Widgetsphere\CodeTools\\*.\*"** |

### 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.

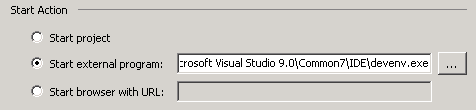
|  |
| --- |
| /// <summary>  /// This template will generate the content for the [TableName]MetaData.cs file. This file  /// will contain a public partial class [TableName]MetaData. This file is  /// provided as a way to allow programmers to extend the generated class.  /// It will be generated once 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.