N-Tier Entity Framework  
User Guide

|  |  |
| --- | --- |
| Version | 1.1 |
| Date | 02.07.2013 |
| Status | Ever evolving |
| Author | Christof Senn |

**Version History**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description |
| 0.1 | 05.11.2010 | Christof Senn | Initial draft (framework version 0.1) |
| 0.2 | 08.11.2010 | Christof Senn | .NET Client Profile support (framework v0.2) |
| 0.3 | 03.12.2010 | Christof Senn | Added description of client info service parameter; transaction management, validation support; Silverlight support (framework v0.3) |
| 0.4 | 10.01.2011 | Christof Senn | Minor corrections (framework v0.5) |
| 0.5 | 24.01.2011 | Christof Senn | Added description of sample applications (framework v0.5) |
| 0.6 | 06.10.2011 | Christof Senn | Added description of dynamic properties, dynamic validators, client profile (framework v0.8 and v0.9) |
| 0.7 | 31.10.2011 | Christof Senn | Created chapter architecture guide; added appendix with references and index; improved layout |
| 0.8 | 15.11.2011 | Christof Senn | Improved overview; added description of added value; inserted layer diagram in architecture section; added sections for design principles and patterns |
| 0.9 | 21.02.2013 | Christof Senn | Corrections and updates |
| 1.0 | 03.03.2013 | Christof Senn | Public release (framework v1.0) |
| 1.1 | 02.07.2013 | Christof Senn | Updates for framework version 1.1 |

**License**

This document is provided “as-is”. Information and views expressed in this document may change without notice.

This document does not provide you with any legal rights to any intellectual property in any Trivadis product. You may copy, modify, and use this document for internal, reference purposes.

Copyright © 2013 Trivadis. All rights reserved.

Trivadis is a trademark of the Trivadis Group. All other trademarks are properties of their respective owners.

The *N-Tier Entity Framework* is licensed under the Apache License, Version 2.0 (the "License"); you may not use the framework or any related work except in compliance with the License.   
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

[Introduction 1](#_Toc361088081)

[1. Overview 2](#_Toc361088082)

[1.1 Why Entity Framework 2](#_Toc361088083)

[1.2 Why N-Tier Entity Framework 2](#_Toc361088084)

[1.3 Added Value 3](#_Toc361088085)

[1.4 Features 3](#_Toc361088086)

[1.5 Current Restrictions 4](#_Toc361088087)

[2. Installation 5](#_Toc361088088)

[2.1 Prerequisites 5](#_Toc361088089)

[2.2 Installation from within Visual Studio 5](#_Toc361088090)

[2.3 Manual Installation 6](#_Toc361088091)

[Architecture Guide 9](#_Toc361088092)

[3. Architecture Guide 10](#_Toc361088093)

[3.1 Distributed Domain Model 11](#_Toc361088094)

[3.1.1 Entity vs. pure DTO 11](#_Toc361088095)

[3.1.2 Remote Façade 11](#_Toc361088096)

[3.2 Design Principles 12](#_Toc361088097)

[3.2.1 Single Responsibility Principle (SRP) 12](#_Toc361088098)

[3.2.2 Open Close Principle (OCP) 12](#_Toc361088099)

[3.2.3 Liskov Substitution Principle (LSP) 12](#_Toc361088100)

[3.2.4 Interface Segregation Principle (ISP) 13](#_Toc361088101)

[3.2.5 Dependency Inversion Principle (DIP) 13](#_Toc361088102)

[3.2.6 Don’t repeat yourself (DRY) 13](#_Toc361088103)

[3.2.7 You ain't gonna need it (YAGNI) 13](#_Toc361088104)

[3.3 Design Patterns 14](#_Toc361088105)

[3.3.1 Unit of Work 14](#_Toc361088106)

[3.3.2 Repository 14](#_Toc361088107)

[3.3.3 Directed References 15](#_Toc361088108)

[3.3.4 Aggregate and Aggregate Root 15](#_Toc361088109)

[3.4 Design Decisions 15](#_Toc361088110)

[3.4.1 How many Entity Models do I need? 15](#_Toc361088111)

[3.4.2 Where do I put my Domain Logic? 15](#_Toc361088112)

[3.5 Framework Comparison 16](#_Toc361088113)

[3.5.1 EF and WCF 16](#_Toc361088114)

[3.5.2 WCF Data Service 17](#_Toc361088115)

[3.5.3 WCF RIA Services 17](#_Toc361088116)

[3.5.4 DevForce 17](#_Toc361088117)

[Development Guide 19](#_Toc361088118)

[4. Development Guide 20](#_Toc361088119)

[4.1 Creating a new N-Tier Entity Solution 20](#_Toc361088120)

[4.2 Adding an Entity Model 21](#_Toc361088121)

[4.3 Code Generation 24](#_Toc361088122)

[4.3.1 Generated Files 24](#_Toc361088123)

[4.3.2 Metadata Files for Custom Data Annotations 25](#_Toc361088124)

[4.3.3 Code Generation Templates and Customization 25](#_Toc361088125)

[4.4 WCF Services 26](#_Toc361088126)

[4.4.1 Passing Custom Data 26](#_Toc361088127)

[4.4.2 Factory Method for Data Repository 27](#_Toc361088128)

[4.4.3 Transaction Management 27](#_Toc361088129)

[4.4.4 Custom Processing 27](#_Toc361088130)

[4.4.5 Interceptors 28](#_Toc361088131)

[4.4.6 Data Validation 30](#_Toc361088132)

[4.4.7 WCF Service Hosting 31](#_Toc361088133)

[4.5 Data Validation 31](#_Toc361088134)

[4.6 .NET Client Profile Support 32](#_Toc361088135)

[4.7 Silverlight Support 32](#_Toc361088136)

[4.7.1 WCF RIA Services 33](#_Toc361088137)

[4.7.2 Native Silverlight Library 35](#_Toc361088138)

[4.8 Testing 38](#_Toc361088139)

[4.9 Consuming an N-Tier Entity Framework Application 39](#_Toc361088140)

[4.9.1 Integrating into MVVM and MVC 39](#_Toc361088141)

[4.9.2 CollectionView and DataLoader for Rapid Application Development 39](#_Toc361088142)

[4.10 Implementing the Aggregate Pattern 40](#_Toc361088143)

[5. Known Issues 41](#_Toc361088144)

[6. Sample Application 42](#_Toc361088145)

[Appendix 43](#_Toc361088146)

[References 44](#_Toc361088147)

[Index 45](#_Toc361088148)

Introduction

# Overview

The N-Tier Entity Framework is based on Microsoft Entity Framework and provides support for n-tier software solutions.

## Why Entity Framework

The Entity Framework (EF) is Microsoft’s data access layer (DAL) strategy that supersedes DataSet by providing an O/R mapper that allows for a conceptual data model to be mapped to a physical data model. The ability to translate data structures from a physical into a conceptual model and back makes EF very powerful and allows for a natural view in an object oriented (OO) fashion on the data.

What’s more, it works seamlessly with a wide range of other Microsoft frameworks and technologies like:

* WCF RIA Services
* WCF Data Services
* DataBinding
* Data Validation (DataAnnotation)

The Entity Framework fully supports Linq, is simple to use and doesn’t imply a steep learning curve.

## Why N-Tier Entity Framework

The Microsoft Entity Framework provides generators for distributable entities which may easily be exposed through WCF (e.g. custom WCF service or WCF Data Service). However, large portions of EF’s comfort are lost when distributing entities over tiers since only the data and possibly metadata is distributed over tiers. Logic contained in EF is not available on client side. Change tracking, Linq support etc. needs to be implemented by the developer in case it is required. These tasks may be both complex and time consuming.  
That is where the N-Tier Entity Framework comes in. It provides an *Entity Framework* like API on the client of an n-tier solution while guiding for a clean client-server design.

Designing a proper n-tier architecture is not a trivial task. The *N-Tier Entity Framework* generates server, client, and shared components based on a conceptual entity model. This way the framework creates a light weight and flexible basis for different kinds of applications. Generated parts may easily be extended and enriched with custom code and specific business logic.

**The N-Tier Entity Framework aims to provide the richness of EF in a distributed architecture while keeping its simplicity.**

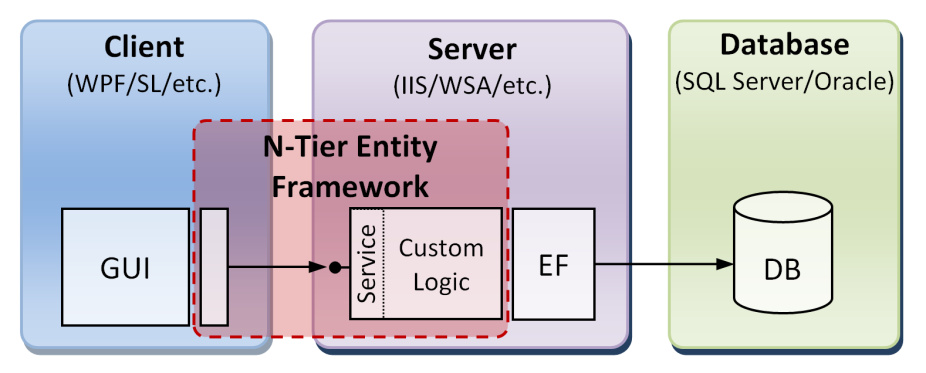


Figure 1: N-Tier Entity Framework Context Diagram

## Added Value

The N-Tier Entity Framework is an architectural framework for Microsoft .NET that supports development of robust and well-designed domain layer for line of business (LOB) software solutions of any size ­– regardless whether building a 2-tier or n-tier system; regardless whether building a Windows or a web application; regardless whether using WPF, Silverlight, ASP.NET, ASP.NET MVC or any other .NET technology.

The framework is simple to use and provides rich out of the box functionality that allows for rapid development of standard scenarios with minimal effort. Still, it is flexible and highly customizable to support development of customized solutions for advanced and complex use cases.

Hence, your gain is a low learning curve and a short time to market of your application compared with other frameworks or custom made solutions.

The N-Tier Entity Framework is designed for:

* Simplicity
* Flexibility / customizability
* Scalability
* Maintainability
* Testability

## Features

The following list provides a rough overview of the features provided by the N-Tier Entity Framework:

#### General:

* Client/service-architecture with clear distinction between client, server, and common parts
* Code of both sides, client and server, provides for extension and customization
* Full Visual Studio integration
* Visual Studio project and item templates
* Code generation based on entity model (edmx) using T4 templates
* Query objects and query parameters (end-to-end query)
* Data validation based on DataAnnotation and IDataErrorInfo
* Transfer of changed data only (minimal bandwidth consumption)

#### Client:

* Remote (WCF) or local (in-process) setup by deployment, based on MEF or other dependency injection framework
* Change tracking provided by DataContext in client component
* Dynamic properties and dynamic data annotations on entities
* DataContext provides observable data collections (entity sets), acting as a local data cache
* Linq support on client providing IQueriable interface
* Client side merge options for subsequent data requests
* Reset of client changes
* Support of inline count in data requests (select count(\*) from table)
* Support of synchronous and asynchronous query and save operations (providing callbacks, events, and tasks)
* CollectionView and DataLoader for automatic server side paging, filtering, and sorting for rapid application development (RAD) with WPF and Silverlight
* Seamless integration with WPF, Silverlight, and any other .NET client

#### Server:

* Stateless data services
* Query and update interceptors on server (concept copied from WCF Data Services)
* Support of server side data validation and manipulation
* Partial methods to inject custom logic into service methods
* Mock-up support on repository and entity sets
* Transaction support (WS-AtomicTransaction)
* User- and role-based security or claims based security support (WS-Security, WS-Federation, WS-Trust)

## Current Restrictions

The N-Tier Entity Framework is currently available for:

* Visual Studio 2010 Professional and above
* .NET Framework 4 (Client Profile) and above
* Silverlight 4 and above
* C#

# Installation

Installing the N-Tier Entity Framework is fairly easy since it is shipped as a Visual Studio Extension (vsix) and is available through the Visual Studio Gallery. However, in order to work with the framework you are required to install the following prerequisites as described by the following section.

## Prerequisites

The N-Tier Entity Framework requires Visual Studio 2010 Professional or above in order to install.

#### Required:

**NuGet Package Manager** is required and may be installed as Visual Studio Extension via Extension Manager.

N-Tier Entity Framework code generation makes use of the **T4 Toolbox** which may be installed as Visual Studio Extension via Extension Manager.

For Visual Studio 2010 T4 Toolbox is not available as Visual Studio Extension and needs to be installed manually. Download the installation file (msi) from <http://t4toolbox.codeplex.com/>. The Framework has been tested using alpha-version 10.10.13.1 of the T4 Toolbox. However, earlier versions might work as well.

To work with Silverlight the **Silverlight SDK** needs to be installed.

#### Optional:

For editing T4 templates it is recommended installing tangible T4 Editor for VS2010 from tangible engineering. This Visual Studio extension provides intellisense and syntax-highlighting which Visual Studio does not provide in T4 editor out of the box: [http://t4-editor.tangible-engineering.com/](http://t4-editor.tangible-engineering.com/T4-Editor-Visual-T4-Editing.html)   
However, tangible T4 Editor is not required for N-Tier Entity Framework to work.

## Installation from within Visual Studio

The N-Tier Entity Framework may be downloaded and installed from within Visual Studio using the Visual Studio Extension Manager found under Tools > Extension Manager…

Selecting Online Gallery from the left hand side menu and entering the search term ‘n-tier entity framework’ in the top right corner should result in listing the framework for download and installation.

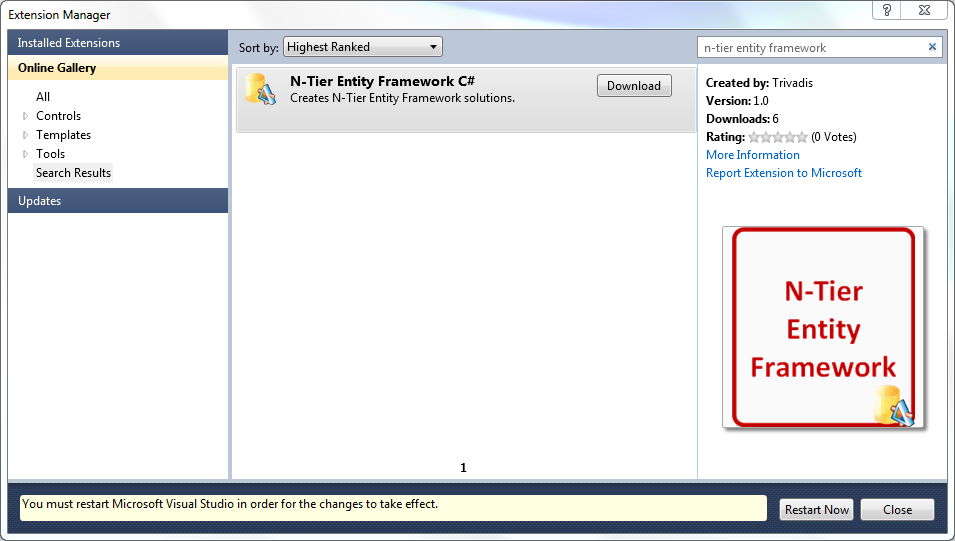


Figure 2: Visual Studio Extension Manager providing integration with Online Gallery

Visual Studio needs to be restarted after installation before the N-Tier Entity Framework is ready to be used.

## Manual Installation

Alternatively the N-Tier Entity Framework VSIX installation package may be downloaded manually from the Visual Studio Gallery (<http://visualstudiogallery.msdn.microsoft.com/>) or may be built locally based on the source published on CodePlex (<http://ntieref.codeplex.com/>).

Executing the vsix file (N-Tier Entity Framework.vsix) and clicking install will do the installation.

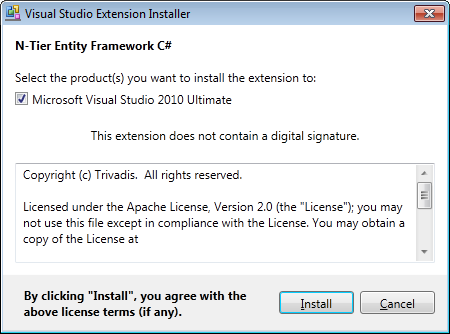


Figure 3: Visual Studio Extension Installer

Note:   
For Visual Studio 2010 extensions are installed in user profile rather than at machine level. Therefore each user has to do the installation on his own profile in case of shared development machines.   
For Visual Studio 2012 and above the *N-Tier Entity Framework* is installed at machine level.

Visual Studio Extensions may be disabled or uninstalled within Visual Studio Extension Manager, found in Visual Studio via Tools > Extension Manager…

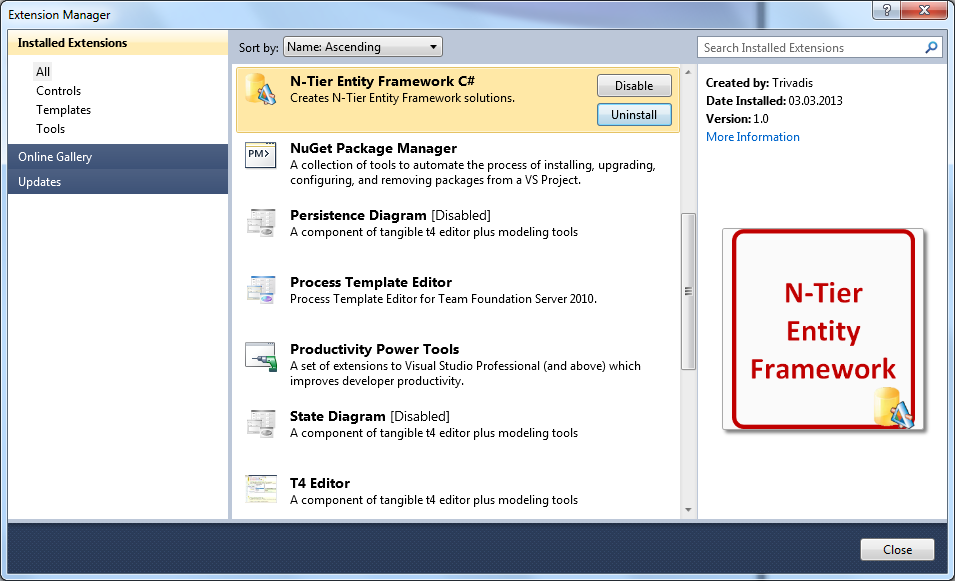


Figure 4: Visual Studio Extension Manager

Architecture Guide

# Architecture Guide

The N-Tier Entity Framework may be seen as not simply a data access layer (DAL) for cross-tier data exchange but rather an application framework for a distributed domain model.

While the framework serves well the purpose of a DAL in applications with little domain logic, e.g. applications structured with the use of the transaction script pattern (Fowler et al. 2002), its strengths pays off when applying the domain model pattern (Fowler et al. 2002), the central pattern in domain driven design (DDD) (Evans 2003), especially for n-tier applications.

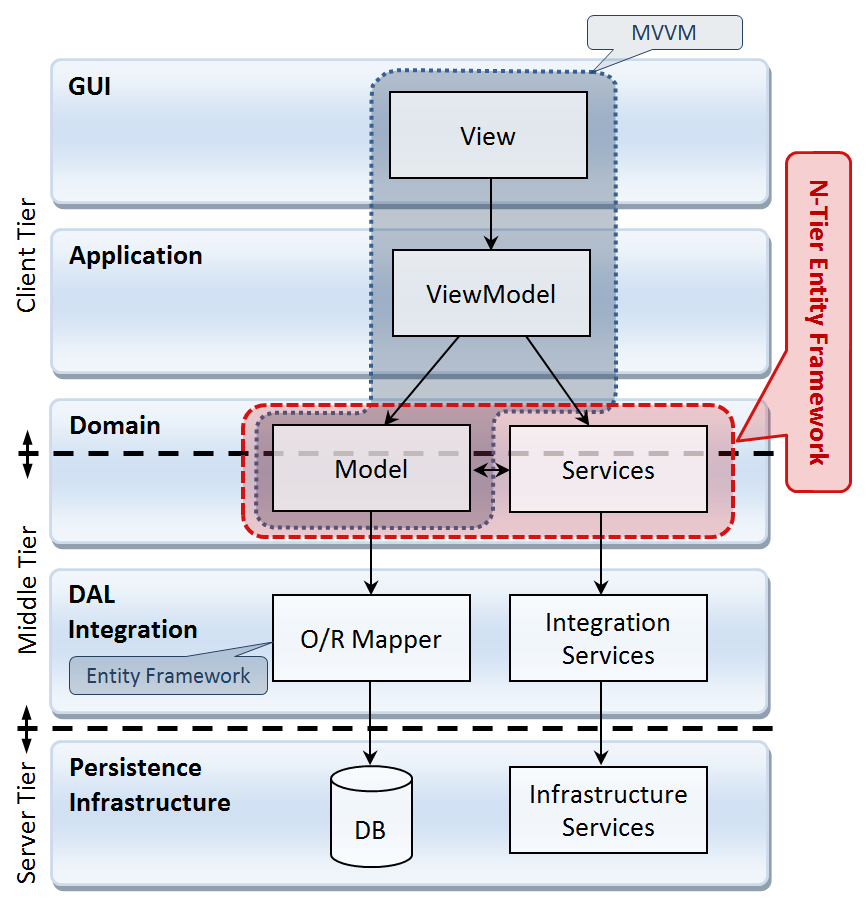


Figure 5: Layered Architecture with Cross-Tier Domain

In solutions build with the N-Tier Entity Framework client and middle tier share a common data model and data transfer between these tiers is organized using entities and value objects (Fowler et al. 2002) which are generated based on the data model. The framework suits best if your design is fine with reading and writing based on the same data structure. If your application generally searches to write completely different data than it uses in its read operations you may be served better using some other framework applying a more dedicated approach like the command query responsibility segregation (CQRS) pattern (Fowler 2011).

You find a discussion about this cross-tier domain approach in following section.

Design patterns and design principles are guidelines and represent a set of common approaches on solving software problems. Both, the framework itself and solutions you build based on the framework should follow these principles.

Principles and patterns are not meant to be adhered dogmatically; and eventually software should be designed for business value. However, as patterns and principles represent proven ways to solve common problems every software designer should at least consider them and apply them whenever appropriate.

This chapter discusses important principles and patterns in regards to the N-Tier Entity Framework.

## Distributed Domain Model

Common architectural models usually imply the physical gap between two tiers to be located in-between two logical layers – between application layer and domain layer or between domain layer and data access layer for example. However, this theoretical definition is rather an abstract simplification and does often not reflect reality. The argument to reach looser coupling by deploying the domain model solely on the middle tier, preventing domain object to be used on the client does not hold true because coupling gets even tighter as aspects of the domain inevitably need to be implemented within the application layer.

The N-Tier Entity Framework follows the approach of a distributed domain model. This architectural pattern accounts for the fact that in most n-tier applications the domain logic is neither exclusively placed on the middle tier nor on the client tier. In practice there are always facets which are more specific to the client application and facets which are better hosted on the mid-tier on a centralized server. What’s more, there are important aspects which are shared among the two tiers.

Applications created with the N-Tier Entity Framework reflect these three aspects with distinct client, server, and common components. The common modules contain the entities, value objects, and services used on both sides, client and server. Client and server components contain mainly services and repositories which are of specific use at their side.

The split of domain logic into client, server, and common does not only make sense when these parts are also actually deployed on different physical tier. It is also an important design decision to account for, that a software solution should be prepared to be potentially distributed at some later stage.

### Entity vs. pure DTO

Data transfer objects (DTO) (Fowler et al. 2002) are simple data containers, specifically designed for data transfer between client and middle tier. The N-Tier Entity Framework doesn’t use specific DTOs for data transfer but rather used its entities and value objects as they represent the model’s basic data structure on both, client and server. Even though these objects are not considered as pure DTOs they server the purpose of data transfer perfectly well containing all information to be transferred from middle tier to clients and back.

Entities generated by the framework include information about modified properties and therefore payload me be reduced to minimum when sending changes from client to server. On the other hand they support adding validation information to be send back from sever to client upon validation failures within an update procedures.

### Remote Façade

The N-Tier Entity Framework generates a WCF service for each entity model, providing a *GetXYZ* service method for each entity type contained in the model. These data query service methods accept a query object parameter which holds query information to filter, sort, and page data results. In addition the query objects may also define a list of other related entity types to be included within the result. On top of this, a query result may also contain the total number of records contained in the database. Unlike multiple query methods there is only one service method per entity model to allow submission of changes. The *SubmitChanges* service method takes a change set object which includes a list for every entity type of the entity model. This service operation works as a unit of work and takes care of all insert, update, and deletes at the same time.

The generated WCF services may be extended with custom service methods as required by the specific use cases. However, the service methods should be kept coarse gained. It is important to keep in mind that the service acts as a remote façade (Fowler et al. 2002) between two physical tiers. This is important even if the application is not physically deployed on different tiers at first, as this may still happen at a later stage.

## Design Principles

Design principles describe fundamental ideas and are basic guidelines in software design.

The SOLID principles are the first five principles in object-oriented design (OOD) (Martin 2005). The following sections describe these and other principles in regards to the N-Tier Entity Framework.

### Single Responsibility Principle (SRP)

*A class should have one, and only one, reason to change.*

The “S” in SOLID requests for high cohesion at class level. Important framework types are Entity, EntitySet, DataContext (client repository), Repository (server repository), DataService etc. Each of these types serve well defined purpose. In case you extend the functionality of these classes you should take care to prevent violation of these purposes.

For more information see <http://en.wikipedia.org/wiki/Single_responsibility_principle>.

### Open Close Principle (OCP)

*You should be able to extend a classes behavior, without modifying it.*

The “O” in SOLID advocates extensible classes which allow for concrete or modified behavior in derived types. The framework provides numerous interfaces and abstract classes which generated parts derive from. By providing partial classes to those generated parts you are able to override protected methods in order to implement custom requirements without the need to modify the frameworks base classes.

For more information see <http://en.wikipedia.org/wiki/Open/closed_principle>.

### Liskov Substitution Principle (LSP)

*Derived classes must be substitutable for their base classes.*

The “L” in SOLID is commonly the most difficult principle in this series to understand. While at a first glance it seems to be natural that in place of a base class one of its derived types may be used in any place, it actually is not. When deriving a class one has to be very careful to make sure to not violate the base class’s invariants, which means not to strengthen any preconditions, and not to weaken and postconditions.

For more information see <http://en.wikipedia.org/wiki/Liskov_substitution_principle>.

### Interface Segregation Principle (ISP)

*Make fine grained interfaces that are client specific.*

The “I” in SOLID targets high cohesion and loos coupling by reducing interfaces to their minimum. Similar to the single responsibility principle for classes an interface should serve a single responsibility or purpose. If too many concerns get packet into a single interface, the interface becomes hard to implement and implementing classes are difficult to maintain.

For more information see <http://en.wikipedia.org/wiki/Interface_segregation_principle>.

### Dependency Inversion Principle (DIP)

*Depend on abstractions, not on concretions.*

The “D” in SOLID suggests relying on abstractions instead of concrete implementations in order to achieve looser coupling. Decoupling and reusability is improved by separating high-level and low-level components into separate components. Abstract types instead of concrete implementations are used when holding references to each other. Different patterns like plugin, service locator, dependency injection, adapter, and factory help to implement concrete solution following this principle.

The framework applies this principle in many circumstances by providing interfaces and abstract types in its base library as well as it generated code.

For more information see <http://en.wikipedia.org/wiki/Dependency_inversion_principle>.

### Don’t repeat yourself (DRY)

The DRY principle suggest in its purest form for any given aspect to have only one single place of truth. When or wherever an aspect is reused, it must not be re-implemented but has to rely on the single place of truth.

Applying this principle the framework reaches a high degree of automaton during development by generating server, client, and common components based on a centrally defined entity model. Furthermore, it’s possible to either generate the entity model from an existing database or alternatively generate the database schema based on the entity model.

What’s more, this principle is basically the main motivation for the cross-tier distributed domain model approach. Having a distributed domain model allows to keep the application layer free of domain logic which is another important aspect considering DRY and an important concern in DDD.

For more information see <http://en.wikipedia.org/wiki/Don't_repeat_yourself>.

### You ain't gonna need it (YAGNI)

The YAGNI principle represents kind of a contrast to other principles and suggests to only add functionality if it is required. This principle has to be balanced against others to reach an optimal cost-benefit ratio. As this principle in its purest form doesn’t take mid-term and long-term architectural considerations into account, it should be used with caution when taken into consideration. While this principle may often not be applied entirely, it is still a reminder that every design decision should be taken according to requirements. Adherence to principles and implementation of patterns should not be done to fulfill a self-purpose but to provide business value.

For more information see <http://en.wikipedia.org/wiki/You_ain't_gonna_need_it>.

## Design Patterns

Patterns are solution to common problems in software development. Many patterns exist and the more you have in your toolbox the better. However, it is important to keep in mind that patterns should not fulfill a self-purpose but help you in solving your problems which in the end are the functional and non-functional requirements of your system.

### Unit of Work

The unit of work pattern allows the execution of multiple tasks at a given time within one transaction, either succeeding or failing as a whole, comparable to database transactions.

The N-Tier Entity Framework applies this pattern for saving changes of the data model. A client application uses a data context object (repository) to retrieve, modify, add, and delete entities of a specific model. At any given point the client may decide to save all pending changes invoking the “SaveChanges” method of the data context. All pending changes are committed to the server and applied to the data source. Should any of the operations involved fail, the operation fails as a whole, letting the client resolve the issue and retry again.   
If required, the framework supports the use of common .NET transaction scopes on client side. On server side the framework ensures the existence of a transaction for saving changes. Extension points in the server component allow the implementation of customized transaction management on server side as well.

The unit of work pattern also finds its use in other aspects like transactional editing of entities. Entities expose methods to begin, cancel, and commit editing allowing changes to be either accepted or rejected in a transactional fashion.

### Repository

The repository pattern is implemented on both, client and server. It represents the single point of entry for its consumers to access and query the model. Furthermore, the repository is responsible to track changes applied to the model and commit them as a unit of work.

The server side repository represents an additional abstraction layer on top of the data mapper. It supports the objective of achieving a clear separation and one-way dependency between the domain and data mapping/data access layer. This is especially useful for testing purpose, allowing the real repository implementation to be replaced with a corresponding mock-up returning hard coded in-memory collections.

At client side the repository encapsulates the (web) service access logic for any interaction with the server. Furthermore it acts as local data cache, keeping loaded data in memory and exposing it without needing a re-query to the server.

### Directed References

References between entities are not only a fundamental concept in OO but also facilitate much of the expressiveness of a domain model. Including every possible reference, however, pollutes the model and decreases its expressiveness.

References should only be established where they make sense from the use cases perspective. Often they are only used in one direction. For instance it might be required for an order entity to hold a list of references to the respective order items while at the same time it is not interesting to be able to navigate from any given order item to its order entity. Here a directed reference should be established in the entity model to only allow navigation from “order” to “order items”.

The Entity Framework and the N-Tier Entity Framework, both support undirected as well as directed references between entities in the data model.

### Aggregate and Aggregate Root

While some objects within a domain model may exist on their own, like products and customers, there are others which are absolutely meaningless without being part of another object, like order items belonging to an order. Such logical groups of entities and value objects, inseparably belonging together, are called aggregates.

An aggregate has a logical boundary where only one of the objects contained is directly accessible from outside of the aggregate. This globally accessible object is called aggregate root. All other objects within the aggregate may not be accessed or referenced directly. Access only works by navigating to them starting at the aggregate root.

While the Entity Framework does not support the concept of aggregates, the N-Tier Entity Framework extends EF’s entity model (edmx) allowing for denoting entities to be aggregate roots and non-roots.

## Design Decisions

In this section you find a description of the most common design decisions to be taken by the software architect, whether or not using the N-Tier Entity Framework.

### How many Entity Models do I need?

One of the first questions arising when creating entity data models is how many entity models I need within my domain and how many entities should I pack into one single model. This question is rather unrelated to technology and must be answered considering the specific used cases and their requirements.

While for some applications one whole entity model is best, considering one entity model per use case or user story respectively is a generally good start. Depending on the complexity of the use cases, additional models may be added to perform individual tasks. On the other hand, a common model may be used by multiple use cases if they share much in common.

### Where do I put my Domain Logic?

Another important question arising is where to actually put the domain logic. As explained, the N-Tier Entity Framework does not try to restrict domain logic to be contained only on client side or only on server side. In fact logic may be implemented on both sides and it is up to the software architect to decide where to put which assets. In addition to this some aspects are required on both sides and therefore implemented within shared components.

Generally, domain logic should be put on the middle tier whenever possible. This tier may be run and managed centrally and is therefore easier to maintain and more reliable than client applications.

Aspects of the domain logic have to be placed on client side when roundtrips to the server would become inefficient or too expensive in any way. An example is factories creating object graphs and setting default values.

The domain model consisting of entities and value objects is used on both, client and server tier, and is therefore contained in a shared assembly.

It’s an overhead for the client to make a roundtrip to the server for simple data validation and therefore needs validation logic to be contained on its side as well as on server side. Hence, most validation logic goes into the shared assembly. For this purpose the N-Tier Entity Framework generates metadata classes for each entity which contain validation logic in form of data annotation attributes. The entities themselves contain logic to allow automatic validation when setting their properties, using validation attributes from metadata.

## Framework Comparison

As there is not just one problem there is also not only one single solution. Multiple frameworks exist, all of them serving a well-defined purpose. First you need to know your requirements – then you are able to decide which solution serves you best.

The following sections provide a short description and comparison of frameworks which are worth to be compared to the N-Tier Entity Framework.

### EF and WCF

The Entity Framework supports generation of self-tracking entities (STE) which are decoupled from their data context and may be exposed through custom WCF services. STEs are capable to internally track changes off their properties and ship this information forth and back between client and middle tier. This makes it easy to implement a basic client server architecture using EF on server side.

However, any functionality provided by EF is not available on client side. Although changes within entities are tracked by themselves, added and removed entities must be tracked by the client. If it’s not desired to ship the whole object tree forth and back within every request to the server, the client must also track which entities actually have changed to reduce to amount of data sent over the wire.

Furthermore, well known featured and patterns like repository, unit of work, LINQ etc. are to be implemented be the client if required. Such solutions soon become complex and are time consuming to maintain.

All these limitations are overcome automatically when using the N-Tier Entity Framework, without losing the ability and flexibility to implement custom logic and behavior on both, client and server side.

### WCF Data Service

While WCF Data Services are good for building RESTful service applications they are rather limited to exposing read-only data or when connecting rich internet applications (RIA) to a data backend using XML or JSON formatted data for exchange.

The limitation to the HTTP protocol as well as limited support for security and the lack of features such as transactional integrity support make WCF Data Service impractical for building typical enterprise applications.

These limitations are all overcome automatically when using the N-Tier Entity Framework as basic WCF is used between client and server.

### WCF RIA Services

The approach of WCR RIA Services is to bridge the physical gap between a Silverlight client and its domain layer which runs on a remote tier. Besides some technical limitations this framework has its main drawback of being only applicable to Silverlight clients.

In comparison the N-Tier Entity Framework targets any type of .NET client. It is even imaginable to generate client parts in other languages letting e.g. a Java client take part in the system.

### DevForce

DevForce from IdeaBlade™ shares a similar high-level approach to n-tier application development based on Microsoft Entity Framework as the N-Tier Entity Framework does. Its strengths lies in rapid application development by bringing an enhanced Entity Framework like API onto the client tier, supporting several client technologies and taking care of the physical gap between client and server tier.

The difference between the two frameworks lies in their architectural approach. DevForce allows for client side programming and does not really consider server side domain logic whereas the N-Tier Entity Framework provides, beyond its out of the box functionality for rapid application development, for a solid basis to build proper client-server architectures focused on domain-driven design with a high degree of flexibility and customizability.

DevForce is kind of a cross-tier data access layer framework whereas the N-Tier Entity Framework may serve the same purpose, but is more of a cross-tier domain layer framework.

Development Guide

# Development Guide

This chapter explains step-by-step how to build an n-tier solution using the N-Tier Entity Framework while discussing specific aspects at the same time.

## Creating a new N-Tier Entity Solution

A new n-tier entity solution is created in Visual Studio by creating a new project (File > New > Project) and selecting the “N-Tier Entity Framework Application” project template under Windows category.

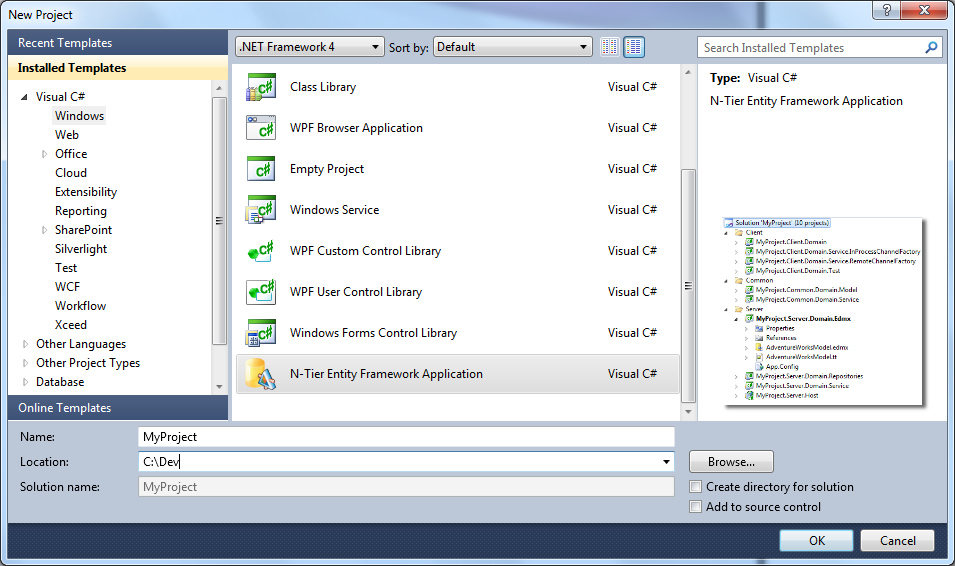


Figure 6: New Project Dialog with “N-Tier Entity Framework Application” Project Template

Note:   
As Visual Studio implies a limitation of the file name length of individual project files upon creating a new solution, which is likely to get exceeded by the lengthy project names created by the N-Tier Entity Framework, it is best creating a new solution in a root path (e.g. D:\) and move the whole solution into your project’s source code folder afterwards.   
For more information about this issue read the corresponding section under chapter “5 Known Issues”.

The “N-Tier Entity Framework Application” project template creates a new solution containing ten projects:

|  |  |
| --- | --- |
| Project | Description |
| <project>.Client.Domain | Main client component containing data context used as entry point for data access |
| <project>.Client.Domain.Service .InProcessChannelFactory | Contains concrete channel factories for in-process configuration of n-tier application – i.e. server parts are executed in client process. |
| <project>.Client.Domain.Service .RemoteChannelFactory | Contains concrete channel factories for remote WCF based configuration of n-tier application – i.e. client and server via WCF. |
| <project>.Client.Domain.Test | Console application used as simple test utility for quick manual testing. |
| <project>.Common.Domain.Model | Contains definition of entities and any types transferred between client and server via WCF services.  This component is shipped to both client and server, serving as shared assembly for WCF communication. |
| <project>.Common.Domain.Service | Contains definitions of WCF services and is shipped to both client and server. |
| <project>.Server.Domain.Edmx | This project is intended to contain entity models (edmx) and T4 templates for code generation. |
| <project>.Server.Domain.Repositories | This project contains the repositories serving as entry points to the entity framework on server side. |
| <project>.Server.Domain.Service | This project contains all WCF service implementations. |
| <project>.Server.Host | ASP.NET application project which may be used to host the WCF services. |

The names of the projects/components correspond to their namespaces. The projects may be divided into the three main categories client, server, and common. Components in the client namespace are to be deployed on client side, server components reside on the server whereas components in the common namespace are deployed to both, client and server.

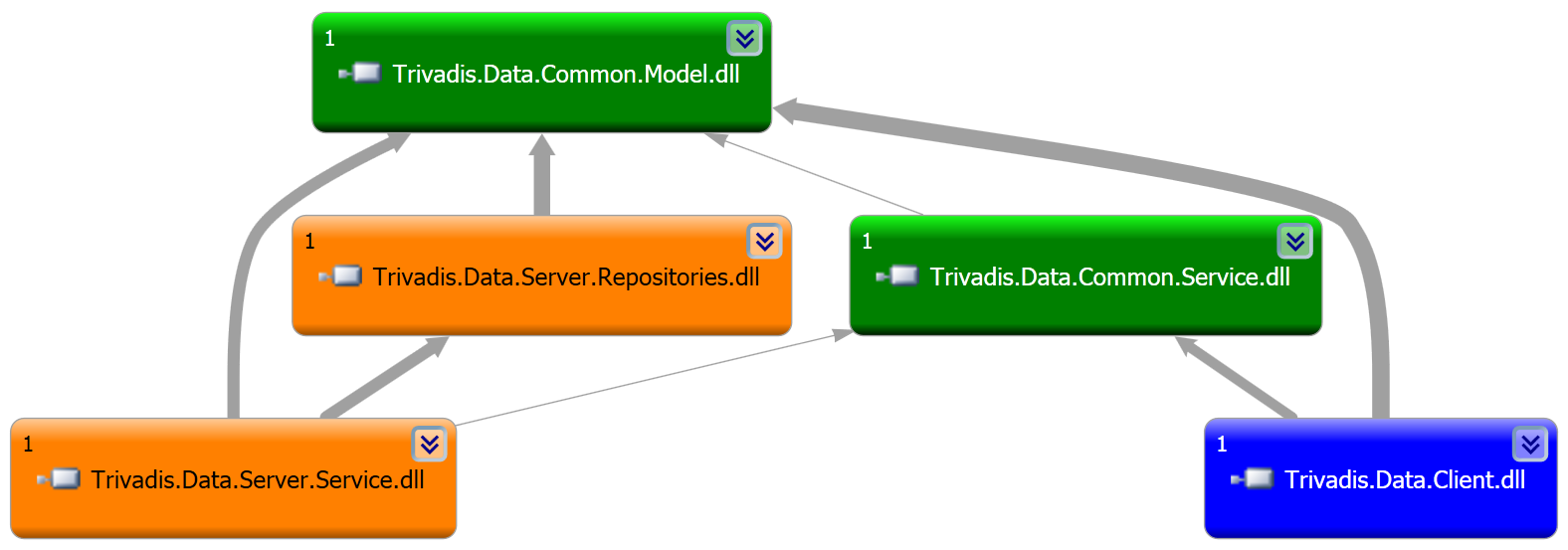


Figure 7: Main Components and their Dependencies

The solution created so far contains references to framework libraries but it does not contain any code yet. Entity models have to be added, in order to turn the solution into a data serving application.

## Adding an Entity Model

Entity models are to be added to the edmx project (e.g. *MyProject*.Server.Domain.Edmx). It is possible to either add existing entity models (edmx) or to create new models using add new item in Visual Studio and creating new ADO.NET Entity Data Models:

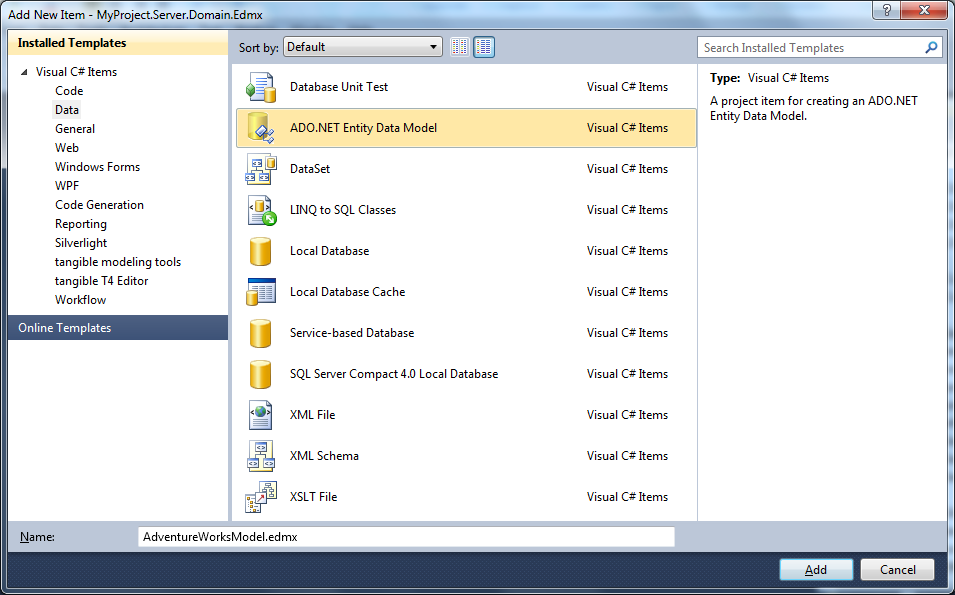


Figure 8: New Item Dialog with "ADO.NET Entity Data Model" Item Template

It does not make any difference how the entity model is created, which data base and data provider is being used. This is under the hood of Entity Framework and does not affect the N-Tier Entity Framework.

For the purpose of this how-to guide an entity model is created based on the “AdventureWorks” sample data base. The sample data base may be downloaded from codeplex: <http://msftdbprodsamples.codeplex.com/>.

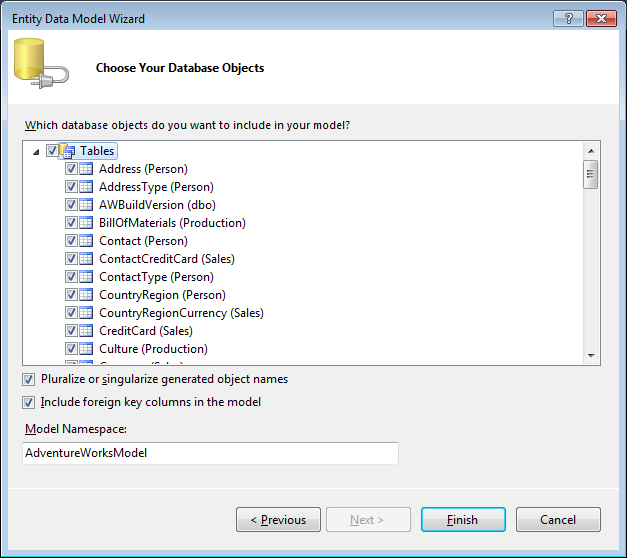


Figure 9: Entity Data Model Wizard

Once the entity model exists a code generation template has to be added in order to generate all required code based on the entity model. For this purpose a Visual Studio item template exists.

The T4 template may be added either by using “add new item” or by simply right-clicking on the entity designer and selecting “Add Code Generation Item” from the context menu.

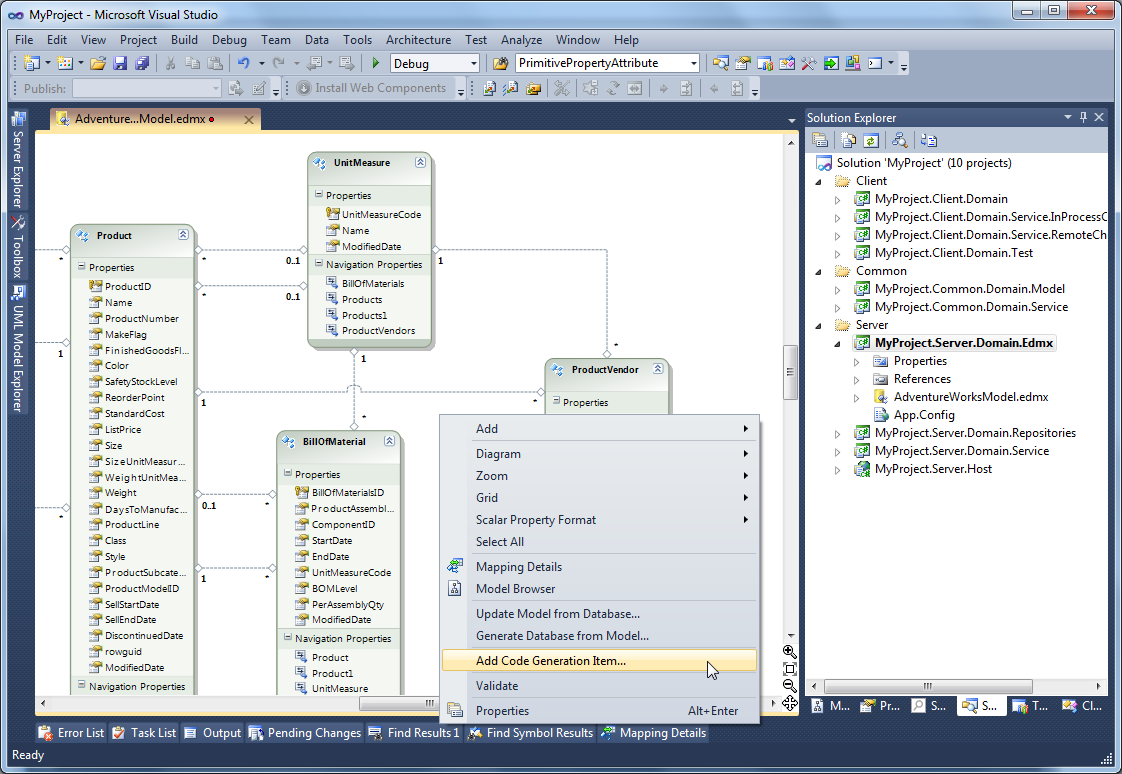


Figure 10: Add Code Generation Item using Context Menu of Entity Designer

From the “add new item” dialog select the “N-Tier Entity Generator” item template found under data. As a convention the T4 template file should have the same name as the entity model edmx file.

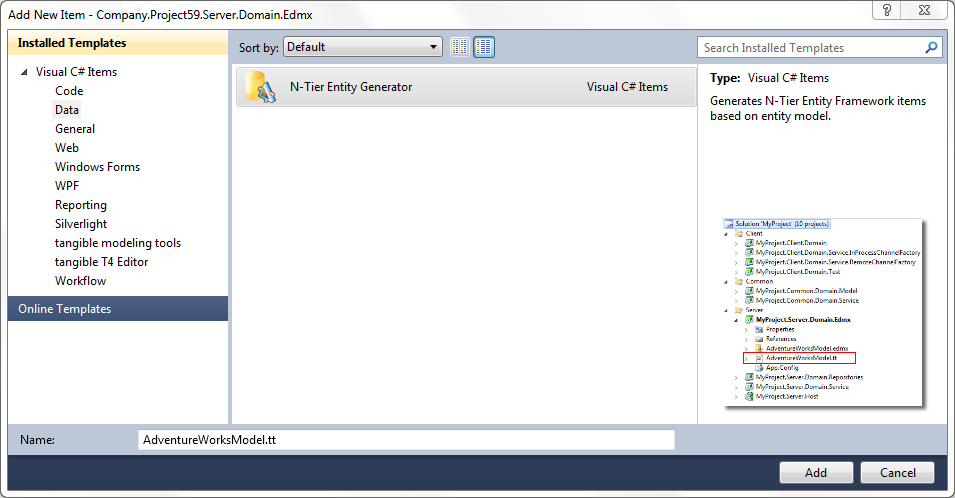


Figure 11: New Item Dialog with "N-Tier Entity Generator" Item Template

Adding this item launches code generation. Note that each time you save changes in your entity model, code is automatically regenerated. To run code generation explicitly you may right-click the code generation template and select “Run Custom Tool”.

## Code Generation

The *N-Tier Entity Framework* bases its code generation on entity data models (edmx). Whether or not you are eventually using *Entity Framework* or another data access technology for interaction with your data sources, you are encouraged to define your data models via edmx.

### Generated Files

The code files generated by the “N-Tier Entity Generator” (added besides the entity model as described in the previous chapter) have the suffix “.g.cs” in their filenames for better housekeeping. “g” stands for “generated”. These files get replaced upon code regeneration and therefore should not be edited manually.

Most generated classes are defined as partial and therefore may easily be extended. By convention code files for manual extensions should be named the same as their generated equivalents using the suffix “.Part.cs” in place of “.g.cs”.

Note:   
Define partial classes in extra cs files to extend generated types rather than editing generated code. Call these files .part.cs.

If you are working with Visual Studio 2010 you may find a log file containing the list of generated files under the T4 template. For Visual Studio 2012 and above this information is stored in the project file, rather than in as separate log file (Note: this behavior is given by the *T4 Toolbox* rather than the *N-Tier Entity Framework*). You should not remove or edit these files since they are used for housekeeping by the code generator.

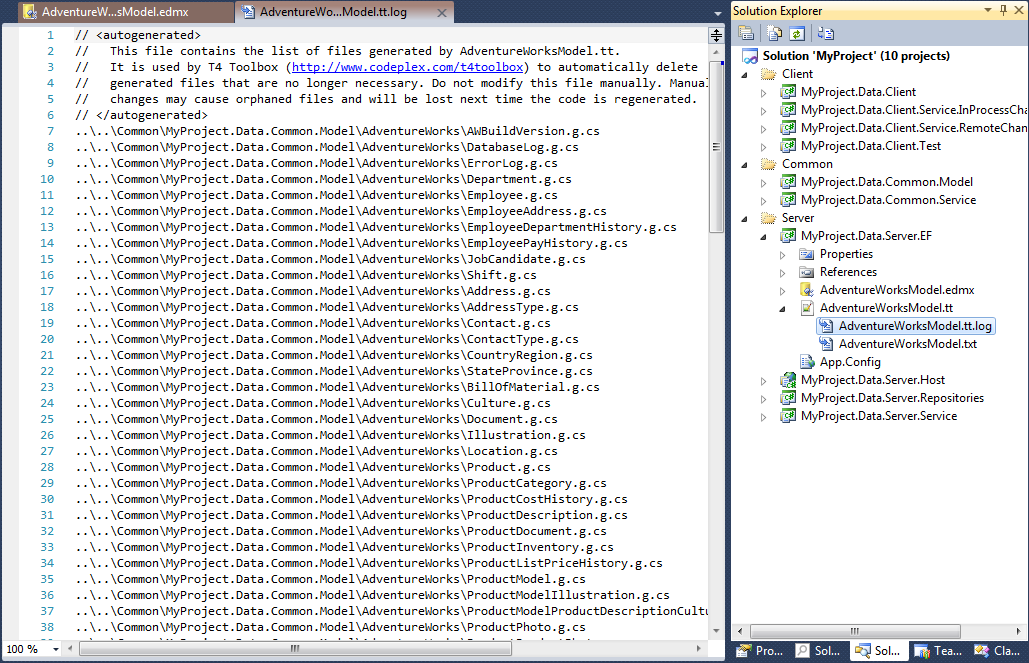


Figure 12: Code Generation Log File

### Metadata Files for Custom Data Annotations

There is one category of generated files which do not get removed nor regenerated unless they are moved or renamed manually by the developer. These files are generated into the “Common.Domain.Model” project and have the suffix “.metadata.cs”. These files are meant to be edited by the developer and therefore are not touched by the generator once they exist.

A metadata file is generated for each entity and serves the purpose of holding data validation metadata in the form of DataAnnotation attributes.

Note:   
You may define DataAnnotation attributes on entities and their properties by editing the corresponding .metadata.cs file.

### Code Generation Templates and Customization

N-Tier Entity Framework’s code generation templates make use of the include functionality to re-use code generation logic provided by the framework. These include files get installed with the framework and are automatically copied into a *T4* directory under each relevant project, either when the project or a specific project item is created. This way code generation logic and templates are placed within the project structure and is easy to be included within a version control system as part of your code.

If customization of code generation logic or templates is required, the T4 files included in the project may be modified as required. There are different types of T4 files provided by the *N-Tier Entity Framework*, each playing its specific role in code generation:

|  |  |
| --- | --- |
| N-Tier Entity Framework ̶ T4 File Types | Description |
| NTierEF.\*TT.CS.ttinclude | Text template for generation of a specific type of classes |
| NTierEF.\*Generator.CS.ttinclude | Generation logic coordinating generation of different types of classes in different projects |
| NTierEF.\*Utils.ttinclude | Utility functionality used by code generators |
| NTierEF.T4Context.ttinclude | Integration for different versions of the T4 Toolbox |
| \*.tt | Code generation item - for code generation of a specific entity data model (edmx) |

## WCF Services

For each entity model a WCF service gets generated containing one “Get” service operation for each entity type contained in the model and one single “SubmittChanges” service operation. There is only one submit changes method to take care of all inserts, updates, and deletes as one unit of work and allow the Entity Framework to take care of relations, dependencies and corresponding update orders.

### Passing Custom Data

In order to allow programmers to pass custom information from client to server, e.g. specific context information, an instance of type ClientInfo may be passed to every service method. The ClientInfo class provides an indexer to allow to store and retrieve objects associated with a key.

Objects set as content of the ClientInfo must be serializable for transportation between client and server. In practice these objects have to be of native .NET types like strings, numeric or date-time values or you have to implement them as classes as part of the “Common.Domain.Model” project and mark them with the DataContractAttribute attribute as the DataContractSerializer is used as the default serializer for WCF.

At client side there are several points where an instance of ClientInfo may be set to be included in service requests. Setting ClientInfo at context level permits the object to be included for all query and save operations, while setting it on a specific entity sets only let the object to be passed to query operations of the specific entity type. At the same time it is possible to set the object for single queries and save operations as shown by the following samples:

// create and populate client info  
var clientInfo = new ClientInfo();  
clientInfo["PropertyName"] = "PropertyValue";  
   
var ctx = new UserAdminDataContext();  
   
// setting client info at context level  
ctx.ClientInfo = clientInfo;  
   
// setting client info at entity set level  
ctx.Users.ClientInfo = clientInfo;  
   
// setting client info for single query  
var users = ctx.Users.AsQueryable()  
    .SetClientInfo(clientInfo)  
    .ToList();  
   
// setting client info for single save operation  
ctx.SaveChanges(clientInfo: clientInfo);

At server side the ClientInfo instance gets passed to repository factory methods, custom processing methods, and interceptors which all are extension points for custom logic. Please read the following sections for further information about these topics.

### Factory Method for Data Repository

The generated WCF service class includes a partial method CreateRepository which may be implemented in a partial class to intercept the creation of the repository instance which is called upon each service request.

partial void CreateRepository(ClientInfo clientInfo, ref IRepository repository);

### Transaction Management

Every save operation is enclosed in a transaction which may be managed by the client specifying a transaction scope:

using (TransactionScope ts = new TransactionScope())  
{  
    ctx.SaveChanges();  
    ts.Complete();  
}

In most circumstances it is not required by the client to manage transactions, leaving this task to the server.

When running the solution remotely, WCF implicitly creates a transaction for the SubmitChanges service method. However, this is not the case when using the in-process configuration, running client and server within a single process on the same machine.

In order to get similar behavior in both configuration modes, remote and in-process, the service base class implements the two virtual methods OnTransactionInit and OnTransactionEnd which are called for every save operation to create and complete a transaction scope in the absence of an existing transaction.

These two methods may be overridden in a partial class of the generated WCF service class in order to participate in transaction management. In this way developers may change transaction options like isolation level by creating an inner transaction and specifying the settings required.

### Custom Processing

There are partial methods named PreProcessing, PostProcessing and BeforeSaving which are called at the beginning and the end of each service operation and before actual saving takes place in the SubmitChanges service method.

These method stubs may be implemented in a partial class of the generated WCF service class in order to process and possibly manipulate service parameters and results before service operations take place and before results get delivered to the client.

Note:   
You may implement partial PreProcessing and BeforeSaving methods to inspect and possibly manipulate service parameters and to run custom data validation.   
Partial PostProcessing methods may be implemented in order to process and possibly manipulate data before it gets delivered to the client.

The following example shows an implementation of a PreProcessing method to demonstrate how custom validation may be implemented:

partial class UserAdminDataService  
{  
    partial void PreProcessing(ClientInfo clientInfo, ref UserAdminChangeSet changeSet,   
                               UserAdminEntities repository)  
    {  
        // retrieve smallest manager salary from database  
        var minManagerSalary = repository.Users.Where(u => u.Role == "Manager").Min(u => u.Salary);  
   
        foreach (var user in changeSet.Users)  
        {  
            // ensure user salary stays bellow smallest manager salary  
            if (user.Salary >= minManagerSalary)  
            {  
                // set error entry for individual entity  
                user.Errors.Add(new Error("Salary exceeds limit", "Salary"));  
            }  
        }  
    }  
}

### Interceptors

The concept of interceptors allows the definition of server enforced query filters for data requests as well as data manipulation logic for inserts, updates, and deletes.   
This concept has been copied from WCF Data Services.

Note:   
You may add query and change interceptors for server enforced query filters and custom data validation and manipulation on insert, update, and delete.

#### Query Interceptors

A query interceptor is a method returning a where predicate to allow for server enforced where clauses which are context agnostic and get executed on the database. A query interceptor is implemented for a specific entity type and gets called each time the corresponding entity is queried.

Here is a sample of query interceptors. The first query interceptor returns a predicate which only allows the administrator to see all products. Other users may only see products of the “Bikes” category. The second query interceptor takes a parameter of type ClientInfo to evaluate a value passed from the client application:

[QueryInterceptor(typeof(Product))]  
public Expression<Func<Product, bool>> ProductFilterByUserIdentity()  
{  
    IIdentity id = ServiceSecurityContext.Current.PrimaryIdentity;  
   
    return product => id.Name == "trivadis.com\\Administrator" ||   
                      product.ProductSubcategory.ProductCategory.Name == "Bikes";  
}  
   
[QueryInterceptor(typeof(Product))]  
public Expression<Func<Product, bool>> ProductFilterWithClientInfoParameter(ClientInfo  clientInfo)  
{  
    // silly demo sample: only return data if the client and server value are equal  
    string clientValue = (string)clientInfo["PropertyName"];  
    return product => "my server value" == clientValue;  
}

#### Change Interceptors

A change interceptor is a method taking an instance of the specific entity type and the update operation as parameters. This method is called for each entity which gets inserted, updated, or deleted. This allows the developer to validate and or manipulate data before submitting it to the database.

Validation errors may be added to the entities errors collection, which prevents committing the changes of the corresponding entity to the database.

Here is a sample of a change interceptor for product entities setting the required guid in case it is missing for inserts. Then the modification timestamp is set on all inserts and updates and lastly it validates start and end date, and adds an error entry in case validation fails:

[ChangeInterceptor(typeof(Product))]  
public void OnChangeProducts(Product product, UpdateOperations operation, ClientInfo  clientInfo)  
{  
    // set row guid  
    if (operation == UpdateOperations.Add)  
    {  
        if (product.rowguid == Guid.Empty)  
        {  
            product.rowguid = Guid.NewGuid();  
        }  
    }  
   
    // set modification timestamp  
    if (operation == UpdateOperations.Add || operation == UpdateOperations.Change)  
    {  
        product.ModifiedDate = DateTime.Now;  
    }  
   
    // validate sell start and end dates  
    if (operation == UpdateOperations.Add || operation == UpdateOperations.Change)  
    {  
        if (product.SellStartDate > product.SellEndDate)  
        {  
            product.Errors.Add(  
                new Error("Sell start date must be before sell end date.",   
                    "SellStartDate", "SellEndDate"));  
        }  
    }  
}

Note:   
Both query and change interceptors may be implemented with or without the ClientInfo argument in their parameter list.

For more information about the concept of interceptors see <http://msdn.microsoft.com/en-us/library/dd744842.aspx>. In order to support strong typing the *N-Tier Entity Framework* does not use the same interceptor attributes as in *WCF Data Services*, however the concept is the same.

### Data Validation

There is no automatic data validation being executed on the server since by default only changed data is transferred which prevents most validators to run.

However, you may enforce data validation accessing the IsValid property on each entity, using the IDataErrorInfo interface of each entity for specific properties, or running your own custom validation logic by implementing change interceptors.

To ensure properties required for data validation are always transferred to the server, these properties may be annotated with the IncludeOnUpdate and IncludeOnDelete attributes in the corresponding entity’s metadata class.

In order to pass validation results back to the client each entity exposes an Errors property where instances to type Error may be added. The Error class provides two overloaded constructors where one takes an error type which may be ErrorType.Error or ErrorType.Warning. The error message string is a mandatory parameter and both of the constructors optionally take an arbitrary list of member names.

// set error entry for individual entity, changes of this entity will not be applied to database  
entity.Errors.Add(new Error("Error message", "Property1"));  
  
// set error entry of type warning, changes of this entity will still be applied to database  
entity.Errors.Add(new Error(ErrorType.Warning, "Error message", "Property1", "Property2"));

The error type specified determines further processing. Entities having an error entry with ErrorType.Error in their errors collection are skipped from being executed while entries with ErrorType.Warning do not affect processing. In fact, submission of any changes to the database is prevented if any of the entities contained in the change set contains an entry with ErrorType.Error. The submission of changes functions as one unit of work, either succeeding or failing as a whole.

Note:   
In case of an error in any entity contained in the change set the save operation fails as a whole.

In case of any errors returned from the server, an exception of type MyProject.Client.Domain.ServerValidationException is thrown on the client, after the SubmitChanges operation completes. This allows a client application to examine the errors collection of every entity and correct issues before attempting another submission of changes to the server.

Optionally the transaction may be aborted within a service operation, calling the Rollback method on the current transaction. However, this prevents from sending error information back to the client as the WCF service connection terminates and a System.ServiceModel.ProtocolException is thrown on the client.

// abort entire transaction  
Transaction.Current.Rollback();

Note:   
In order to prevent the insert, update, or delete operation to take place you need to explicitly add an Error entry to the entity’s Errors collection. Error entries may be analyzed at client side to learn about the reason of the error. Only Error entries with ErrorType.Error affect server side processing, letting the whole submission fail as one unit of work.

For more information about how to add validation logic, read chapter “4.3.2 Metadata Files for Custom Data Annotations” and “4.5 Data Validation”.

### WCF Service Hosting

Hosting the WCF services is fairly simple using the provided ASP.NET application (Server.Host). This web application already contains the required svc files for the generated services. All you have to do is to define the WCF service configurations in the Web.config. In addition to this you also need to specify the connection strings for all database connections the Entity Framework requires.

For convenience the Web.config already contains a connection string and a WCF service configuration for the AdventureWorks sample. You just need to uncomment it and possibly adapt the connection string for your environment to make the sample work.

Note:   
Do not forget to define WCF service configuration and connection string for each of your entity models.

## Data Validation

Data validation is performed within entities, based on data annotation. For this purpose, properties may be annotated with DataValidationAttribute attributes as described in chapter “4.3.2 Metadata Files for Custom Data Annotations”.

While static data annotation is the preferred way to define validation logic, since this way validation information is automatically available on client and server side using shared assembly, DataValidationAttribute attributes may also be registered dynamically at runtime allowing for dynamic injection of validation logic. For this purpose, each entity type provides the static methods RegisterValidator and UnregisterValidator which allow validation attributes to be registered and unregistered respectively. Both methods take a string and a ValidationAttribute parameter. The string parameter must hold the name of the property to which the validator has to be assigned to. It is important to note that validators which are registered dynamically at client side are not automatically available at server side and vice versa.

Due to a limitation in Entity Framework, neither change tracking nor data validation is started automatically if an instance of an entity is created using the constructor. Both change tracking and validation are started automatically when an entity is attached or added to a data context. Change tracking and validation may also be started explicitly calling the extension method StartTracking.   
The limitation of *EF4* is that there is no way to know about when materialization of an entity starts. Hence, validation is disabled by default in order to allow loading data from the data source, even is the data possibly does not meet the validation requirements.

The simplest way to create a new entity at client side is to call the CreateNew factory method provided by any entity set. The new entity gets added to the entity set automatically, which makes sure both, validation and change tracking are enabled.

var entity = context.EntitySet.CreateNew();

Note:   
Due to a limitation in *Entity Framework* change tracking and validation is not enabled immediately after the creation of an instance of an entity. The entity has to be attached/added to a data context or the StartTracking extension method has to be called before modifying any data properties.   
  
All of this is accomplished automatically when using the CreateNew factory method of any entity set.

If enabled, validation is performed automatically when modifying a property of an entity. In case of a validation error, a corresponding ValidationException is thrown. Automatic validation may be suppressed setting the IsValidationEnabled property to false.

Validation may also be requested explicitly by using the *System.ComponentModel.IDataErrorInfo* interface, which is implemented by the *Entity* base class.

Note:   
Turn on automatic data validation to prevent invalid data being set into to model. Turn off automatic validation if you prefer to set user input into to model first and perform validation afterwards.

## .NET Client Profile Support

The N-Tier Entity Framework is targeting the .NET Client Profile by default.

For information about compatibility issues with client profile in conjunction with WCF RIA Services read chapter “4.7.1 WCF RIA Services”.

For more information about *.NET Framework Client Profile* see <http://msdn.microsoft.com/en-us/library/cc656912.aspx>.

## Silverlight Support

The N-Tier Entity Framework provides support for Silverlight in two ways. One way is to connect via WCF RIA Services while the second and preferred way is to connect a Silverlight application directly to the data service.

Either way, to develop N-Tier Entity Framework solutions for Silverlight the Silverlight 4 SDK is required to be installed.

### WCF RIA Services

A T4 code generation template is provided to generate domain services based on entity models. A domain service uses the N-Tier Entity Framework as any other .NET client does by accessing entity sets through an instance of a client data context.

The generated domain service classes may be edited manually to define default sorting, filters, additional service methods etc. Code does not get regenerated unless source files are removed manually.

#### Code Generation

There is no specific Visual Studio project template required. You may simply use the existing WCF RIA Services Class Library project template. To allow for default behavior on code generation you should name this project “<MyProject>.RIAServicesLibrary” (convention over configuration).

It is easy to use the provided code generator in both Silverlight web applications and WCF RIA Service class library web projects. You may add the T4 code generation template selecting the “N-Tier Entity Domain Service Generator” item template under Silverlight from the “add new item” dialog.

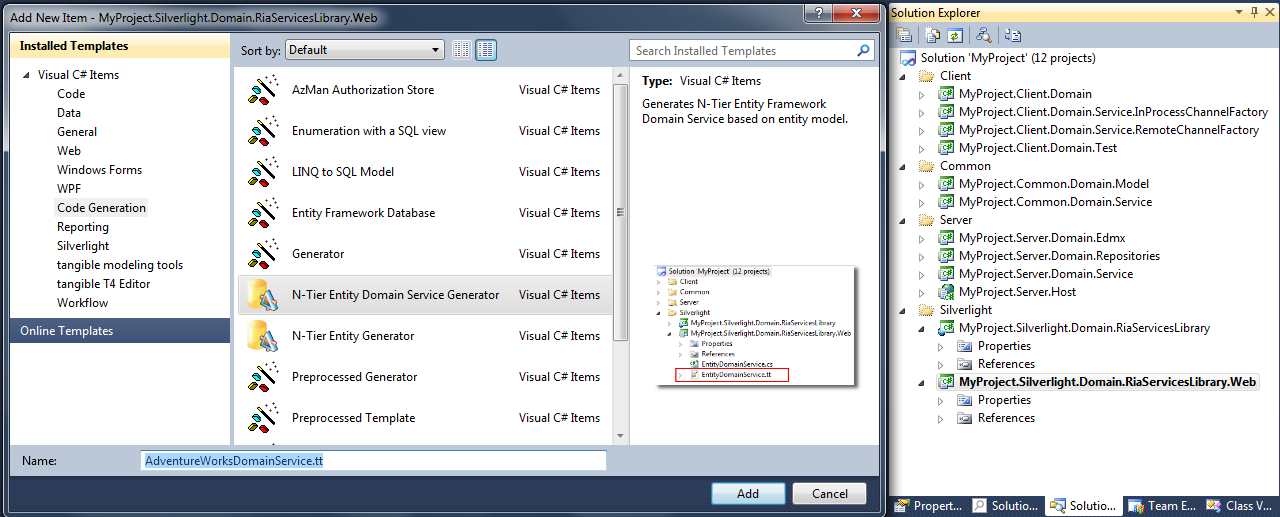


Figure 13: New Item Dialog with "N-Tier Entity Domain Service Generator" Item Template

The code generator expects the entity data model (edmx) file to be located at the same level in a solution folder called “Server”. The tt-file should have the same name as the edmx file, i.e. if the model file is called EntityModel.edmx you should name the tt-file either EntityModel.tt or EntityDomainService.tt. If both, location and name meet default settings, code gets generated automatically. Otherwise, you get an exception when adding the domain service generator. But don’t worry; all you have to do is to specify the edmx and project file. In this case you open the tt-file and specify project files and or edmx file manually as shown by the following sample:

<#@ template language="C#" hostspecific="true" debug="true" #>  
<#@ output extension="txt" #>  
<#@ include file="$(ProjectDir)\T4\NTierEF.DomainServiceGenerator.CS.ttinclude" #>  
<#  
 // <copyright file="EntityDomainService.tt" company="Trivadis">  
 // Copyright © Trivadis. All Rights Reserved.   
 // </copyright>  
   
 // Specify project files in case projects do not use default project names.   
 DomainServiceGenerator.ClientLibraryProject =   
 @"..\..\Client\MyProject.Client.Domain\MyProject.Client.Domain.csproj";  
 DomainServiceGenerator.DataModelProject =   
 @"..\..\Common\MyProject.Common.Domain.Model\MyProject.Common.Domain.Model.csproj";  
 DomainServiceGenerator.EdmxProject =   
 @"..\..\Server\MyProject.Server.Domain.Edmx\MyProject.Server.Domain.Edmx.csproj";  
   
 var generator = new DomainServiceGenerator(this,   
 @"..\..\Server\MyProject.Server.Domain.Edmx\AdventureWorksModel.edmx");  
 //generator.DomainServiceTemplate.GenerateModificationMethods = false; //default: true  
 generator.Run();  
#>  
Generates domain service.   
  
Note:   
 By default generated domain service is created only in case the code file does not exist.   
 In order to regenerate the corresponding file need to be renamed or removed manually.

#### Project References

Since the generated domain services are using the N-Tier Entity Framework classes as any other .NET client does you need to add corresponding references.

At design time the client library and model assemblies or projects need to be referenced. While at runtime, additionally the assemblies for channel factory and service contracts need to be available as well.

#### WCF RIA Service does not support client profile

It is important to note that for *WCF RIA Services* is not possible to target *.NET Framework Client Profile,* since required framework types, like the *System.ServiceModel.DomainServices.Server.ExcludeAttribute*, are not defined for client profile. As a result, the NTierEF.Common.Domain.dll and NTierEF.Server.Domain.dll assemblies need to be compiled targeting “.NET Framework 4” instead of “.NET Framework 4 Client Profile”.

For the very same reason you need to remove the CLIENT\_PROFILE compilation symbol in the project settings of your “Common.Domain.Model” project and target “.NET Framework 4”.

#### WCF RIA Service Specifics

For Silverlight code generation, which is taken care of by WCF RIA Services, general rules do apply. E.g. in order to generate navigation properties in the Silverlight code, the corresponding properties need to be annotated with the IncludeAttribute and the AssociationAttribute attributes in the model. This is done best in the metadata class of the specific entity as shown in the following example:

[MetadataType(typeof(OrderMetadata))]  
public partial class Order  
{  
}  
  
public sealed class OrderMetadata  
{  
    public int ID;  
      
    [System.ServiceModel.DomainServices.Server.Include]  
    [System.ComponentModel.DataAnnotations.Association("OrderItems", "ID", " OrderID")]  
    public TrackableCollection<OrderItem> OrderItems;  
}

### Native Silverlight Library

The N-Tier Entity Framework may be used from Silverlight directly. Since Silverlight applications may not reference normal .NET assemblies few additional steps are required.

#### Creating a Silverlight Class Library

In order to access the WCF services exposed by the N-Tier Entity Framework from Silverlight, a Silverlight class library containing all code from client and common namespaces is required.

A specific Visual Studio project template is provided to create an N-Tier Entity Framework class library for Silverlight.

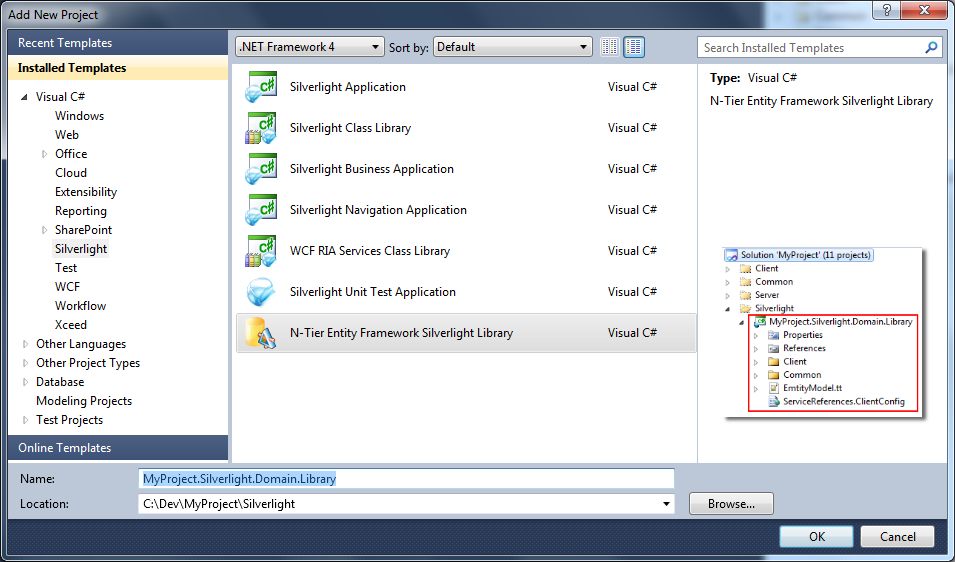


Figure 14: New Project Dialog with “N-Tier Entity Framework Silverlight Library” Project Template

It is recommended to use the existing solution and place the project in a folder called “Silverlight” at the same level as Client, Common and Server folders. It is important to pay attention to physical folder structure since other projects in the solution need to be references by the code generator.

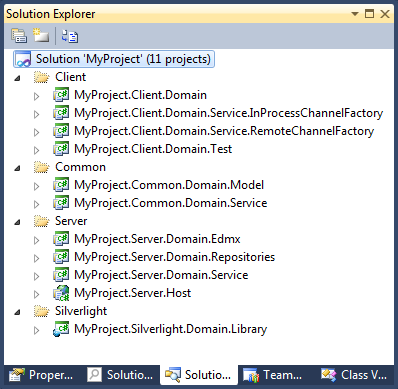


Figure 15: Recommended Folder and Project Structure

Note:   
You may specify the location of the project in the new project dialog: <solution directory>\Silverlight

Another recommendation for the Silverlight class library is to use the same base name as for the rest of the solution. This way the default settings of the code generator work fine and the least amount of configuration is required by the developer.

When creating the Silverlight class library a popup window appears. In this window the base namespace needs to be specified. Verify that namespaces displayed in the wizard match the namespaces of the client, model and service contract projects.

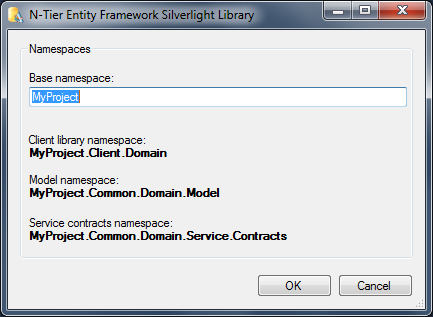


Figure 16: Configuration Wizard for the Silverlight Class library

#### Code Generation

The Silverlight class library contains an EntityModel.tt file for code generation. This file generates the code based on the edmx file. Rename the tt-file to match the name of the edmx file.

If you are using default naming and folder structure the code gets generated automatically. Otherwise, you need to open the tt-file and specify project files and or edmx file manually as shown by the commented lines:

<#@ template language="C#" hostspecific="true" debug="true" #>  
<#@ output extension="txt" #>  
<#@ include file="$(ProjectDir)\T4\NTierEF.SL.NTierEntityGenerator.CS.ttinclude" #>  
<#  
 // <copyright file="EntityModel.tt" company="Trivadis">  
 // Copyright © Trivadis. All Rights Reserved.  
 // </copyright>  
   
 // Specify project files in case projects do not use default project names.   
 NTierEntityGenerator.ClientLibraryProject =   
 @"..\..\Client\MyProject.Client.Domain\MyProject.Client.Domain.csproj";  
 NTierEntityGenerator.DataModelProject =   
 @"..\..\Common\MyProject.Common.Domain.Model\MyProject.Common.Domain.Model.csproj";  
 NTierEntityGenerator.EdmxProject =   
 @"..\..\Server\MyProject.Server.Domain.Edmx\MyProject.Server.Domain.Edmx.csproj";  
   
 var generator = new NTierEntityGenerator(this,   
 @"..\..\Server\MyProject.Server.Domain.Edmx\AdventureWorksModel.edmx");  
 generator.Run();  
#>  
See EntityModel.tt.log for list of generated files.

To generate code for multiple entity models you may copy the EntityModel.tt file or add new ones selecting the “N-Tier Entity Silverlight Generator” using the “add new item” dialog.

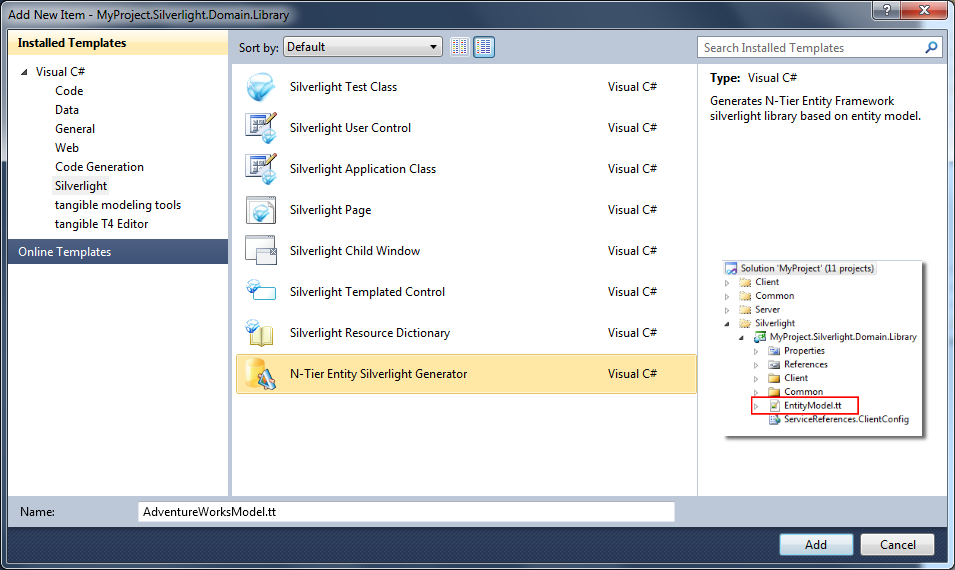


Figure 17: New Item Dialog with "N-Tier Entity Silverlight Generator" Item Template

Important: Since you may have specified metadata (e.g. data validation logic) for your entities the code generation relies on the model assembly (MyProject.Common.Domain.Model) to extract and include corresponding attributes in the generated code for Silverlight. Therefore you are required to build the model project (e.g. MyProject.Common.Domain.Model) before running Silverlight code generation.

It’s worth to note that even it would have been easier to include the logic defined in the common.domain.model’s metatdata files via linked items, it wasn’t possible for data annotation since the *System.ComponentModel.DataAnnotations.MetadataTypeAttribute* doesn’t exists for Silverlight. However, there is nothing preventing you from using linked items to share other custom code you might want apply on both platforms, .NET and Silverlight.

Note:   
Before running code generation for Silverlight you need to build the common.domain.model project.

#### Using the Library

In order to use the Silverlight class library you may just add a reference to it from your Silverlight application project. For the WCF service client to work you need to include a corresponding service configuration.

A sample configuration file is included in the Silverlight class library for your convenience. You may just copy the “ServiceReferences.ClientConfig” file to your Silverlight application project and change the content specific to your environment.

The n-tier entity data service may be hosted either at your Silverlight web host or in a separate web application as you prefer. In case you host the service in a different location to your Silverlight application, make sure to include a clientaccesspolicy.xml and a crossdomain.xml file at the root of the web application hosting the data service. This is required by Silverlight in order requests may be sent to that location. The web host application of the n-tier entity solution includes these two files as a reference. They are configured to allow request from any location and therefore should be changed for production.

## Testing

Once you have added an entity model and an N-Tier Entity Generator, and you have defined WCF service configuration and connection string in the Web.config of the WCF service hosting application you are ready to run your data service for the first time.

A simple and quick way to test your solution is by using the test project (Client.Test) included in the solution.

Before you can do so, you need to specify the WCF service client endpoint configuration in the app.config of your test application. For the AdventureWorks sample there already is a working configuration you just need to uncomment.

Note:   
The client application needs a WCF service client endpoint configuration corresponding to the WCF service on server side.

In the main method of the test console application you may now add code to test the N-Tier Entity Framework application.

Such a test could possibly look like this:

using NTier.Client.Domain;  
using MyProject.Common.Domain.Model.AdventureWorks;  
   
namespace MyProject.Client.Domain.Test  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // create context  
            var ctx = new AdventureWorksDataContext();  
   
            // create query  
            var query = from p in ctx.Products**.AsQueryable()**  
                        where p.ProductID == 523  
                        select p;  
   
            // execute query  
            var product = query.First();  
   
            // modify data  
            product.Color = "Gold";  
            product.ListPrice = 123.99m;  
   
            // save changes  
            ctx.SaveChanges();  
        }  
    }  
}

Attention:   
Pay attention to the AsQueryable() method which is called to make the data request being executed remotely against the WCF service. If you do not call the AsQueryable() method only data already being stored locally in data context is being returned.

## Consuming an N-Tier Entity Framework Application

The usage of N-Tier Entity Framework applications is fairly similar to the usage of normal Entity Framework application.

An instance of a data context is used as the entry point to a given entity model to access data, add, delete and manipulate entities and their properties, and saving changes back to the server.

However, there are two main distinctions to make:

1. **No lazy loading**: there is nothing such as automatic lazy loading of data. Data that has to be loaded needs to be specified explicitly in a query:  
   e.g. context.Products  
   .AsQueryable()  
   .**Include("ProductSubcategory.ProductCategory")**
2. All data loaded through a data context is automatically saved (referenced) in that data context. Accessing an EntitySet property of a data context (e.g. context.Products) does return this locally saved data. In order **to make a data request to be executed against the server** querying data form database **the method AsQueryable() has to be called**:  
   e.g. context.Products.**AsQueryable()**

### Integrating into MVVM and MVC

TODO

### CollectionView and DataLoader for Rapid Application Development

TODO

## Implementing the Aggregate Pattern

TODO

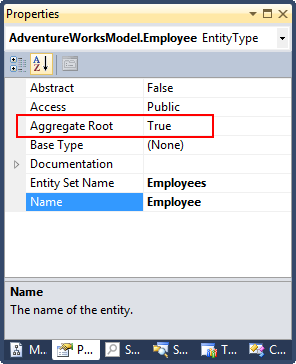


Figure 18: Property window of an entity showing the 'Aggregate Root' option

# Known Issues

#### The filename or extension is too long

If you get this error message while creating a new *N-Tier Entity Framework* solution you need to either select a different location for the solution with a shorter path, or choose a shorter name for the solution.

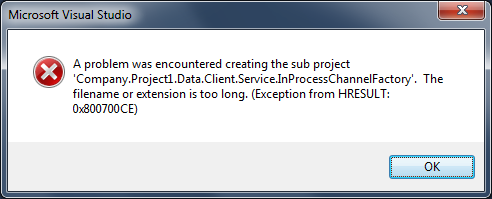


Figure 19: Message shown by Visual Studio due to limitation of file name length

This is due to a limitation of file name length *Visual Studio* adds at the creation of new projects. You may still try to re-locate your solution to a longer path after creation.

# Sample Application

There is a sample application available, developers may use as a reference. The sample was build using the N-Tier Entity Framework and includes a WPF, a Silverlight, as well as an ASP.NET MVC application as frontend.

The sample application is called “Product Manager” and includes four Visual Studio solutions:

#### ProductManager

The first solution “*ProductManager”* contains the *N-Tier Entity Framework* based projects and servers as the central data/application server.

#### ProductManager.Silverlight

The “*ProductManager.Silverlight”* solution contains a Silverlight application consuming the product data service, displaying a paged list of products. Data is requested asynchronously from the server while performing paging, filtering and sorting in the database.

#### ProductManager.WPF

The “*ProductManager.WPF”* solution contains a WPF application consuming the product data service, displaying a master-detail-view of products; allowing products to be changed, added, and removed.

The WPF application is build using the WPF Application Framework (WAF). WAF is based on Microsoft’s Managed Extensibility Framework (MEF) and is designed to support the implementation of the Model-View-ViewModel (MVVM) pattern. WAF may be downloaded from codeplex: <http://waf.codeplex.com>

#### ProductManager.Web

The “*ProductManager.Web”* solution contains an ASP.NET MVC 2 application displaying a paged list of products, allowing the list to be sorted by the user. Products may be added, edited, and removed using the corresponding action links.

Even though you might only be interested in one of the UI technologies demonstrated in the sample applications, you may also want to have a look at the other projects included, as they are all showing different ways of interaction with the *N-Tier Entity Framework*.

All that is required for the sample applications to build and run is Visual Studio 2010 (or later), SQL Server Express 2008 (or later), and Silverlight 4 SDK (or later) for the Silverlight sample. Other required DLLs as well as a reduced version of the AdventureWorks database are contained in the solutions.

Appendix

References

Evans, Eric J. 2003. *Domain-Driven Design. Tackling Complexity in the Heart of Software.* Addison-Wesley.

Fowler, Martin. 2011. “CQRS.” Retrieved September 19, 2011 (http://martinfowler.com/bliki/CQRS.html).

Fowler, Martin, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Liebhart, Daniel, Guido Schmutz, Marcel Lattmann, Markus Heinisch, Michael Königs, Mischa Kölliker, Perry Pakull, and Peter Welkenbach. 2006. *Architecture Blueprints. Ein Leitfaden zur Konstruktion von Softwaresystemen mit Java Spring,.NET, ADF, Forms und SOA.* 2nd ed. Hanser.

Martin, Robert C. 2005. “The Principles of OOD.” Retrieved November 22, 2011 (http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

Starke, Gernot. 2009. *Effektive Software-Architekturen. Ein praktischer Leitfaden.* Hanser.

Index

C

Code generation

code generation item, 23, 33, 36

domain model generator, 24

generated files log, 24

n-tier entity generator, 24

Silverlight generator, 37

WCF RIA Service generator, 33

D

Data validation, 31

injecting logic at runtime, 31

metadata files, 25

on server, 30

Domain model

cross-tier domain. *See* distributed domain model

distributed domain model, 11

E

Entity Framework

entity model, 21

S

Server processing

interceptors, 28

partial methods, 27

transaction management, 27

T

T4

tangible T4 Editor, 5

V

Visual Studio

extension manager, 6

project template, 20

W

WCF

WCF Data Services, 17

WCF RIA Services, 17