

PLINQO for LINQ to SQL

Archive of the CodeSmith Tools documentation for PLINQO for LINQ to SQL, as published on the CodeSmith Confluence site. It is kept for reference and is no longer updated. Current documentation is at www.codesmithtools.com/docs.

33 pages.

Contents

[Home](#)

[Getting Started](#)

[What's New](#)

[Features](#)

[Documentation](#)

[Feature Tour](#)

[Business Logic Features](#)

[Auditing](#)

[Using the Rules Engine](#)

[Rules Example Code](#)

[Data Access Features](#)

[Entity Enhancements](#)

[Enum Generation](#)

[Many to Many Relationships](#)

[Query Extensions](#)

[Performance Features](#)

[Caching](#)

[Future Queries](#)

[LINQ to SQL Profiler](#)

[Performance Improvements](#)

[Project Features](#)

[Customization & Refactoring](#)

[Data Services & WCF](#)

[The Quick Start](#)

[Using the Templates](#)

[DBML Template](#)

[Entities Template](#)

[Managers Template](#)

[Query Extensions Template](#)

[Quick Start Template](#)

[Downloads](#)

[Contributions](#)

[Testimonials](#)

Home

Looking for [PLINQO for Entity Framework](#), or [PLINQO for NHibernate](#)?

It's still LINQ to SQL, but better!



PLINQO is a **replace and enhance**
alternative to LINQ to SQL.

PLINQO is available in C# and VB, and is compatible with any project using .NET 3.5 or greater.



It's still LINQ to SQL, but better!

PLINQO is a **Replace and Enhance alternative to LINQ to SQL.**

PLINQO is available in C# and VB, and is compatible with any project using .NET 3.5 or greater.

Overview

PLINQO, which stands for Professional LINQ to Objects, is a collection of CodeSmith templates that are meant to replace and extend the LINQ to SQL designers that are included with Visual Studio.

What is PLINQO for LINQ to SQL?

PLINQO opens the LINQ TO SQL black box giving you the ability to control your source code while adding many new features and enhancements. In the time that LINQ to SQL has been available, we have been identifying ways to make LINQ to SQL better. We have compiled all of those cool tips and tricks including new features into a set of CodeSmith templates.

PLINQO's generated output is high quality, easy to maintain code that creates a solid foundation and will jump start any project on the road to success. PLINQO takes full advantage of many unique CodeSmith features including merge strategies, CodeSmith projects and Visual Studio integration.

Why should you make PLINQO your ORM of choice? Excellent question! Our [Why PLINQO?](#) article can tell you the many reasons.

Documentation

All of our documentation is provided in a friendly wiki format.

Please use the navigation tree on the left to locate the subject that you are interested in.

Additional Resources

[Official Website](#)

[Feature Tour](#)

[Issue Tracker](#)

Getting Started

Getting started with PLINQO for Linq-to-SQL is an easy three step process.

Step 1: Download CodeSmith Generator

Download and install the latest version of CodeSmith Generator.

PLINQO requires the latest version of **CodeSmith Generator**.

If you have not previously registered with CodeSmith Tools or do not currently own a license of CodeSmith Generator, you will be required to **register** on the CodeSmith Tools website to get a trial key before using CodeSmith Generator.

Step 2: Find PLINQO

1. Open **Template Explorer**
2. Navigate to Frameworks\PLINQO in the Generator Samples folder.

Step 3: Run the QuickStart

1. **Double click the quickstart.cst template**
2. Select a database

3. Click generate!

Clicking generate will automatically open Visual Studio so that you can quickly get started using PLINQO!

Once the Visual Studio Solution has opened, build the project to generate PLINQO!

What's New

For a full change log of all the new features and bug fixes for each version of PLINQO be sure to look at the [PLINQO Release Blog](#).

PLINQO 5.2

- Added support for .Net 4.5 and Visual Studio 2012/2013.
- Fixed a bug where an exception would be thrown when trying to add an association that contains a column that was is not a supported db type (E.G., spatial types).
- Fixed a bug where Visual Studio Solutions wouldn't open automatically.
- Fixed a bug where a VB Compiler error would occur when generating functions with Xml Serialization enabled.
- Fixed a issue where VB access modifiers could be incorrectly generated.
- Fixed a bug where ToPlural could return a lowercased resolved mapped name.
- Fixed a query extensions compiler error that would occur for non nullable XML columns (Operator '=' is not defined for types 'System.Xml.Linq.XElement' and 'System.Xml.Linq.XElement'.)

PLINQO 5.1

- Added support for Generator 6.x.
- Add MemCached Provider to PLINQO.
- Added IncludeManyToMany option.
- Updated Entities to support Nullable Composite Key assignment.
- Updated RuleManager and added Remove method.
- Added additional ClearCache methods.
- Updated interfaces for many associations to be IEnumerable instead of IList, this allows for comparability with .NET 3.5
- Various Performance Improvements when generating.
- Database columns that have the type of Geography and geometry will now be skipped.
- Make sure that no lazy loading happens while doing a detach.
- Fixed a bug with Xml Comments not being escaped and two bugs where invalid syntax was used.
- Fixed a bug that could cause the CacheManager to remove the wrong cache entry.
- Fixed a bug with PLINQOS IncludeXmlSerialization support and Binary Columns.
- Fixed a Compiler Error with Visual Basic Function Support.
- Fixed a bug in the Visual Basic Query Extensions.
- Fixed a bug with the Generated Interfaces not importing the entity namespace. This effected complex column properties..
- Fixed a bug where existing Member Names starting with a digit where treated as valid names. If a memeber names starts with a digit, we will update it!
- Fixed a NullReferenceException in LinqToSqlDataContextProvider.GetDataContext(IQueryable query)

PLINQO 5.0.4

- Fix bug with GetHashKey and Contains not being unique
- Fixed a bug where the DataContext was not being named correctly.
- Updated PLINQO Interfaces to include associations
- Fix issue with Transaction and GetCommand

PLINQO 5.0.3

- Fix to remove unprocessed function types and columns
- Updated the syntax for invoking stored procedures.
- Fix issues with AuditLog by adding a Refresh method on AuditManager
- Add support for XmlReader to AuditLog
- Add support for table CRUD mapped to functions

PLINQO 5.0.1

- Added IncludeList property to dbml template to support explicit include
- Fixed keyword escape issues in functions
- Added Serializable attribute to entities
- Added QueryNamespace property to queries template so queries can be in a different namespace
- Make enums use full namespace to prevent naming issues
- Fixed issue with many to many association error "An item with the same key has already been added."
- Updated vb function to use correct ByRef or ByVal in parameters
- Fixed issue for cache on different databases
- Fixed include list to not prevent association generation
- Fixed issue with cache default profile
- Fix to make IsPrimaryKey sync correctly
- Fixed issue initializing PLINQO rules asynchronously

PLINQO 5.0

- Added Visual Studio 2010 and .NET 4 Support
- Updated QuickStart with latest Dynamic Data and DataServices
- Added SQLCacheDependency Support
- Added IQueryable Cache Extensions
- Added Include extension for advanced eager loading
- Improved AuditLog, now includes updates after SubmitChanges
- Fixed bug in CacheManager where default profile did not load properly
- Added override RuleManager.Run override that populates the TrackedObject.Original
- Fixed bug in AuditLog
- Fixed bugs in VB Quick Start

- Added DataContextName to QuickStart.cst
- Fixed Managers didn't have the same access modifier as entities
- Fixed bug in GetByKey function in VB
- Fixed invalid cast exception for Range attribute on short properties
- Fixed bug in ExecuteFutureQueries when no queries were loaded
- Fixed bug in Manager when column didn't exist
- Added logic to skip columns of type 'hierarchyid'
- Added work around for L2S Profiler not working with transactions
- Fixed bug where Metadata property types did not match up with Enums
- Added BeginTransaction with no overload

PLINQO 4.0.1

- Change to argument order of operators to make it more clear which is the left and right side of the operation. **(Breaking Change)**
- Add NextPage, PreviousPage, GotPage to PagedList
- Update CodeSmith.Data.chm to have correct email address for support
- Add methods GetOrSet to CacheManager and CacheProvider
- Add BeginTransaction to DataContextExtensions
- Add GeneratedCodeAttribute to Generated classes
- Fix issue with duplicate many to many names
- Add full namespace to Metadata class
- Fix bug in LinqCacheDataSource
- Add DataContextName to dbml template and Generator
- Change default Clean Expression to be less aggressive
- Added Current and Original entities to AuditEntity.
- Fixed a bug loading NRefactory causing the templates not to compile
- Add NotNull Rule and Attribute.
- Fix bug with Update and TimeSpan data type.
- Fix issue with column named "Key"
- Change ViewDirectory to ModelDirectory
- Change comments in Entity.Editable.cst
- Fix ToBinary
- Improve sql generated with null values

PLINQO 4.0

- Added future query support similar to NHibernate Futures
- Added ability to query by null to query extensions
- Added ability to query by a list of values to query extensions
- Added ability to query using comparison operators (ie. greater than, less than) to query extensions
- Added cache provider support
- Added support for using Ayende's LINQ to SQL Profiler

- Added Memcached cache provider
- Added ability to create named cache profiles in configuration
- Added ability to validate individual entities
- Refactored RuleManager to lazy load shared rules
- Combined extension method namespaces to make the extension methods more discoverable
- Added support for SQL Server user defined functions
- Added entity interfaces to help with mocking and other uses
- Updated ToPagedList() to use future queries for retrieving the total row count
- Added ToBinary(), FromBinary(), and FromXml() methods to help with entity serialization
- Added UTC support for the Now attribute and rule
- Added ClearCache() to remove queries from the cache
- Fixed several bugs with batch queries
- Many other bug fixes and small enhancements listed on the CodeSmith Google Code project site.

PLINQO 3.1

- Added query result caching.
- Added a new AlwaysAuditAttribute to make it possible to always audit a property even if it hasn't changed.
- Added partial change tracking methods for EntityRef associations.
- Added support for maintaining user defined associations.
- Added partial change tracking methods for EntityRef associations.
- Added shared version information for Assemblies.
- Added support for AuditPropertyFormat.
- Removed AddRule that is no longer used in manager templates.
- Updated manager templates classes to lazy load.
- Fixed many to many issue when IncludeDeleteOnNull is false.
- Fixed bug with a keyword in function parameter name.
- Fixed bug with transactions in ExecuteQuery.
- Fixed bug with nullable columns in queries.
- Fixed bug with nullable parameter in query.
- Fixed support for inheritance.

PLINQO 3.0

- Added Entity Detach.
- Added Entity Clone.
- Added support for Many-to-Many Relationships.
- Added auditing.
- Added Batch Updates and Deletes.
- Added support for multiple result sets.

PLINQO 2.x

- Added Enum table support.
- Added Many To Many association support.
- Added Detach method to Context.
- Added ICloneable on Business Entities.
- Updated Insert Class Merge Strategy to support merging Namespaces.
- Fix Issue with DeleteOnNull persisting when IncludeDeleteOnNull set to False.

PLINQO 2.0

- Added Quickstart template to allow quick generation of a Plinqo based web project.
- Added various Naming Convention properties to give better control over object naming standards.
- Added Dynamic Data support to generated entities.
- Added Metadata Internal class.
- LinqEntityBase now has an editable parent file.
- Manager produces rules based on MetaData Attributes
- Added Global.asax Template
- Entity Serialization adds functionality based on .Net framework version.
- Added Data Services support.

PLINQO 1.3

- Added GetByKey(IEntityKey) common method for manager template
- Made managers inherit from base classes in manager template
- Made primary key query use common name GetByKey in manager and query templates **(Breaking Change)**
- Fix namespace issues when context and entity namespaces aren't the same
- Fix QueryExtension with wrong context name
- Fix CompiledQuery issue when there are more then four columns
- Fix issue with compiled queries and load options
- Fix bugs when dbml file is loaded in designer
- Added support for adding the ConnectionString to the settings file
- Added Queries.cst to support query extension Methods
- Removed manager data context, now another partial of the DataContext **(Breaking Change)**
- Added support for unique indexes
- Added range rule for dates
- Update to .net 3.5 RTM

PLINQO 1.2

- Fix LinqDataSource issue by making all properties writable

- Added CompiledQuery support to the manager classes
- Added support for overriding the column data type

PLINQO 1.1

- Changed to use dbml schema as meta data
- Split templates up to allow more flexibility
- Added function, table inheritance, ignoring, and name cleaning support

Features

Features

The templates have the following features.

- Generate or update a LINQ to SQL dbml file from a database schema.
 - Includes all tables, stored procedures, functions, and views with the ability to exclude objects based on regex patterns.
 - Ability to automatically remove object prefix and suffixes (ie. tbl_ and usp_).
 - Ability to standardize object names produced from database tables.
 - Dbml file can still be customized with the normal Visual Studio designer.
 - Dbml file can be refreshed based on the current database schema without losing customizations. (*See Safe Attributes*)
- Generation of the LINQ to SQL DataContext class.
- Generation of the LINQ to SQL entity classes.
 - Generates one file per entity instead of one massive file.
 - Generates partial classes where custom code can be written and won't be overwritten.
 - Generates Metadata classes with attribute tags from DBML information.
 - Generated entity files are added to the project as code behind files to their corresponding custom entity files.
- Generation of entity manager classes.
 - Adds customizable business rules engine to enforce entity validation, business and security rules based on metadata tags.
 - Provides access to common queries based on primary keys, foreign keys, and indexes.
 - Common queries are exposed as IQueryable so they can be extended.
- Generation of query extension methods.
- All templates can be customized to meet your needs.
- Query Caching
- Performance Improvements
- Rules Engine

Advantages Over Designer

There are several advantages to using the PLINQO templates over the Visual Studio designer. Here is a list of the main reasons:

1. Remove the designer black box and allow for customization of the output while still retaining the ability to use the .dbml designer to make customizations inside of Visual Studio.
2. Ability to easily generate your entire .dbml file for a database and then the ability to regenerate that .dbml file as the schema changes. The regeneration preserves any customizations you may have made such as entity, property and relationship names. With the designer, if you make a database change, you

need to drop the entity and re-add it to get any new columns or data type changes, which would cause you to lose any customizations you may have made. Also, using the templates allows you to exclude unwanted tables, stored procedures and views using filter expressions and automatically strip / clean entity and property names of things like prefixes and suffixes that your database schema may be using (ie. `tbl_Customer` to `Customer`).

3. A business rules engine that allows you to enforce things like property length rules, required field rules, regex data validation rules as well as several other built in rules including authorization rules using metadata attributes. The `SubmitChanges` method on the data context object will automatically run the rules against any entities in your change set. If all rules are not met, a `BrokenRulesException` will be thrown that contains a list of the broken rules.
4. A manager class is generated for each entity that encapsulates all actions taken on an entity. Known common actions like retrieving entities by primary key, indexes, and foreign keys are generated. Any custom actions can be added and will be preserved during regeneration. While LINQ makes it easy to sprinkle your data access logic throughout your entire application, we still believe its poor design to do so and that is why we have included the manager classes.

Documentation

API Documentation

The CodeSmith.Data API documentation can be downloaded from [here](#).

If you are experiencing difficulties viewing the downloaded help file; please follow [this guide](#) to resolve this behavior.

Feature Tour

This [section](#) includes a full tour of [PLINQO's features](#).

Template Documentation

This [section](#) will cover how to use the various templates that ship with PLINQO.

Feature Tour

Business Logic Documentation

This [section](#) will cover some common business uses for PLINQO.

This section includes the following pages:

- [Auditing](#)
- [Rules Engine](#)

Data Access Documentation

This [section](#) will cover some Data Access features with PLINQO.

This section includes the following pages:

- [Entity Enhancements](#)
- [Enum Generation](#)
- [Many to Many Relationships](#)
- [Query Extensions](#)

Performance Documentation

This [section](#) will cover some performance improvements that PLINQO offers.

This section includes the following pages:

- [Caching](#)
- [Future Queries](#)
- [LINQ to SQL Profiler](#)
- [Performance Improvements](#)

Project Features Documentation

This [section](#) will cover some of the features that come with PLINQO.

This section includes the following pages:

- [Customization & Refactoring](#)
- [Data Services & WCF](#)
- [Quick Start](#)

Business Logic Features

The following pages go over some business uses that PLINQO provides.

- [Auditing](#)
- [Rules Engine](#)

Auditing

Auditing

Check out this PLINQO feature video for more information on Auditing!

What data changed, what was it and what is it now! Questions that are asked all the time. When these questions are asked, PLINQO is ready to help provide the answer. When `AuditingEnabled` is set to true on the data context, PLINQO will capture the change anytime changes are submitted to the database. PLINQO captures only the objects that are changed and only the properties in those objects that were changed everytime changes are submitted to the database. The before and after values are recorded. `Context.LastAudit` is where this information is held and there is a `ToXml()` method that makes it easy to turn the `AuditLog` into xml for easy storage. Here, we change the `User` and the `Task` objects, submit the changes and the xml for the audit is displayed. PLINQO auditing will log Inserts, Deletes and Updates and eliminates the work normally required when the decision is made that auditing of data changes is needed.

```
using (var context = new TrackerDataContext())

{
    Priority p = new Priority();
    p.Name = "High!";
    context.Priority.InsertOnSubmit(p);

    Task t = context.Task.GetByKey(1);
```

```

t.Details = "Startup Counterstrike. Used PLINQO. Project is done";

context.SubmitChanges();
AuditLog audit = context.LastAudit;
}

```

```

< audit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.codesmithtools.com/datacontext/audit/1.0" >
  < entity action="Insert" type="Tracker.Data.Priority" >
    < key name="Id" type="System.Int32" >
      < value xsi:type="xsd:int" >0< /value >
    < /key >
    < property name="Id" type="System.Int32" >
      < current xsi:type="xsd:int" >0< /current >
    < /property >
    < property name="Name" type="System.String" >
      < current xsi:type="xsd:string" >High!< /current >
    < /property >
    < property name="Order" type="System.Int32" >
      < current xsi:type="xsd:int" >0< /current >
    < /property >
  < /entity >
  < entity action="Update" type="Tracker.Data.Task" >
    < key name="Id" type="System.Int32" >
      < value xsi:type="xsd:int" >1< /value >
    < /key >
    < property name="Details" type="System.String" >
      < current xsi:type="xsd:string" >Startup CounterStrike. Used PLINQO. Project is done<
/current >
      < original xsi:type="xsd:string" >Work overtime to get project started< /original >
    < /property >
  < /entity >
< /audit >

```

Next: Using the Rules Engine

Using the Rules Engine

PLINQO Rules! Yes, PLINQO provides several different ways to lay down the law. Rules can be added programmatically or declaratively through attributes. Constraints like property length and required field rules can be enforced. Regular Expression data validation as well as several built in rules including authorization rules are possible with PLINQO. Before any data is saved, the rules are automatically executed against any entities in your change set. If any rules are broken, a `BrokenRulesException` will be thrown with a list of the rules that were broken and the entity will not be updated.

The PLINQO rule manager generates rules based on the schema and any custom rules can be added to the rules collection. The rules are enforced when any attempt to save changes is made. Custom rules are a snap to add and the `AddSharedRules` partial method on each entity is the place to add them. Only a few lines of code and a custom rule can be added. Here is a look.

A rule for the minimum length of `UserName` is added.

```
static partial void AddSharedRules()
{
    RuleManager.AddShared(new CustomRule("UserName", "UserName must be 5 characters.",
MinLengthUserName));
}

private static bool MinLengthUserName(string username)
{
    if (String.IsNullOrEmpty(username) || username.Length < 5)
```

```
        return false;
    else
        return true;
}
```

When any rules are broken, no data is updated and a nice list of the rules broken is returned.

```
using (var context = new TrackerDataContext())
{
    User user = new User();
    context.User.InsertOnSubmit(user);
    context.SubmitChanges();
}
```

Rules can be added declaratively as well by taking advantage of the metadata class mentioned previously. PLINQO supports the standard rules in the `System.ComponentModel.DataAnnotations` assembly as well as the rules defined by PLINQO.

As you can see there are many ways to enforce that the data that gets saved is data that follows the rules.

Next: Rules Example Code

Rules Example Code

The following sample code shows how one can take advantage of all the built-in Rules that are offered in PLINQO.

```
using System;
using System.ComponentModel.DataAnnotations;
using CodeSmith.Data.Attributes;
using CodeSmith.Data.Rules;
using CodeSmith.Data.Rules.Assign;
using CodeSmith.Data.Rules.Security;
using CodeSmith.Data.Rules.Validation;

namespace Sample.Data
{
    public partial class User
    {
        static partial void AddSharedRules()
        {
            //validation rules
            RuleManager.AddShared<User>(new CompareRule<int>("Age", 21,
ComparisonOperator.GreaterThanOrEqual));
            RuleManager.AddShared<User>(new CustomRule<string>("UserName", "UserName must be
unique.", UniqueUserName));
            RuleManager.AddShared<User>(new LengthRule("FirstName", 100));
            RuleManager.AddShared<User>(new RangeRule<int>("Age", 18, 21));
            RuleManager.AddShared<User>(new RegexRule("Email", @"^\w+([-+.']\w+)*@\w+
([-.]?\w+)*\.\w+([-.]?\w+)*"));
            RuleManager.AddShared<User>(new RequiredRule("Email"));

            //assignment rules. These rules are to help in assigning default values
            RuleManager.AddShared<User>(new GuidRule("UserId", EntityState.New));
            RuleManager.AddShared<User>(new NowRule("CreatedDate", EntityState.New));
            RuleManager.AddShared<User>(new NowRule("ModifiedDate", EntityState.Dirty));
            RuleManager.AddShared<User>(new UserNameRule("CreatedBy", EntityState.New));
            RuleManager.AddShared<User>(new UserNameRule("ModifiedBy", EntityState.Dirty));
            RuleManager.AddShared<User>(new IpAddressRule("IpAddress", EntityState.Dirty));
            RuleManager.AddShared<User>(new DefaultValueRule<int>("Score", 100,
EntityState.New));

            //security rules
            RuleManager.AddShared<User>(new CreateRule("Administrators", "Users"));
            RuleManager.AddShared<User>(new DeleteRule("Administrators"));
            RuleManager.AddShared<User>(new UpdateRule("Administrators"));
        }
    }
}
```

```

//This is called by the custom rule. The first argument is the property value.
private static bool UniqueUserName(string username)
{
    //check user name is unique;
    return true;
}

#region Metadata

private class Metadata
{
    // Only Attributes in the class will be preserved.

    [Guid(EntityState.New)]
    public Guid UserID { get; set; }

    [Required]
    public string UserName { get; set; }

    [IpAddress(EntityState.Dirty)]
    public string IpAddress { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.EmailAddress)]
    [RegularExpression(@"\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*"])]
    public string Email { get; set; }

    [StringLength(100)]
    public string LastName { get; set; }

    [Range(18, 21)]
    public int Age { get; set; }

    [Now(EntityState.New)]
    public DateTime CreatedDate { get; set; }

    [Now(EntityState.Dirty)]
    public DateTime ModifiedDate { get; set; }

    [UserName(EntityState.New)]
    public string CreatedBy { get; set; }

    [UserName(EntityState.Dirty)]
    public string ModifiedBy { get; set; }
}

#endregion

```

```
}  
}
```

Next: Entity Enhancements

Data Access Features

The following pages go over some data access features that PLINQO offers.

- [Entity Enhancements](#)
- [Enum Generation](#)
- [Many to Many Relationships](#)
- [Query Extensions](#)

Entity Enhancements

Entity Enhancements

If you want some more info about detaching, cloning, and serialization, take a look at this [quick overview](#)!

Detach

Inability to detach is what coders have complained about most when working with LINQ to SQL. Well, complain no more. PLINQO makes it easy to work with entities independent from a datacontext and attach to another when you are ready to save your changes. The following code is now possible thanks to PLINQO.

```
Task task = null;
using (var context = new TrackerDataContext())
{
    task = context.Task.FirstOrDefault(t => t.Id == 1);
    task.Detach();
}

task.StatusId = 1;

using (var context2 = new TrackerDataContext())
{
```

```
context2.Task.Attach(task, true);
context2.SubmitChanges();
}
```

The code shown here works without changes if your database supports row versioning. If row versioning is not supported in your database, the Update Check policy on each attribute in the dbml must be set to Never for detach to work.

Read more about how PLINQO implemented detach.

Clone

PLINQO takes advantage of the DataContractSerializer in WCF to provide a quality cloning solution for all entity objects. Simply call the Clone method on any entity generated by PLINQO and what you get is an object copied to its own memory location. Below is a quick sample of the clone method in action. A user object is retrieved from the database, cloned, one property is changed, a new user is saved to the database.

```
using (var context = new TrackerDataContext())
{
    var u = context.Manager.User.GetByKey(1);
    User clonedUser = u.Clone();
    clonedUser.Id = 0;
    context.User.InsertOnSubmit(clonedUser);
    context.SubmitChanges();
}
```

Serialization

Serialization of entities has had developers running in circles from the early days of LINQ to SQL until now. PLINQO eliminates this frustration by eliminating the possibility of circular references and taking advantage of WCF datacontract serialization. Here is a sample of serializing an object using PLINQO.

```
Task task = context.Task.GetByKey(1);
string xml = task.ToXml();
byte[] b = task.ToBinary();
```

Deserialization is also made simple with PLINQO

```
task = Task.FromXml(xml);  
task = Task.FromBinary(b);
```

Next: Enum Generation

Enum Generation

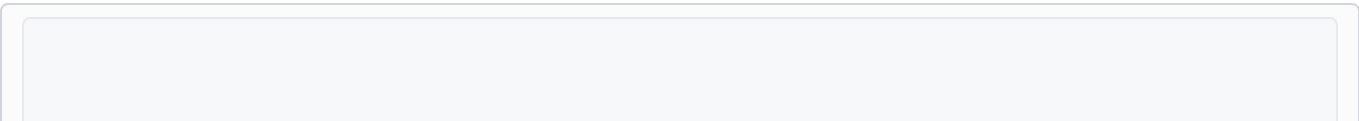
Enum Generation

Watch this video for a quick tutorial on Enum Generation!

PLINQO's enum generation can easily turn a table of data into an enum. Enum values are often stored and maintained in the database and in code resulting in the task of updating both locations when changes are needed. Making the same change twice just to keep things in sync is definitely a frustrating process. PLINQO makes it simple to define the list of Enum tables to generate and which fields contain the enum name and description. The result is enums are maintained in one spot and life is good.

PLINQO enum generation easily turned priority table into an enum.

Priority	
	Id
	Name
	[Order]
	CreatedDate
	ModifiedDate
	RowVersion



```
[DataContract]
public enum Priority : int
{
    [EnumMember]
    High = 1,
    [EnumMember]
    Normal = 2,
    [EnumMember]
    Low = 3,
}
```

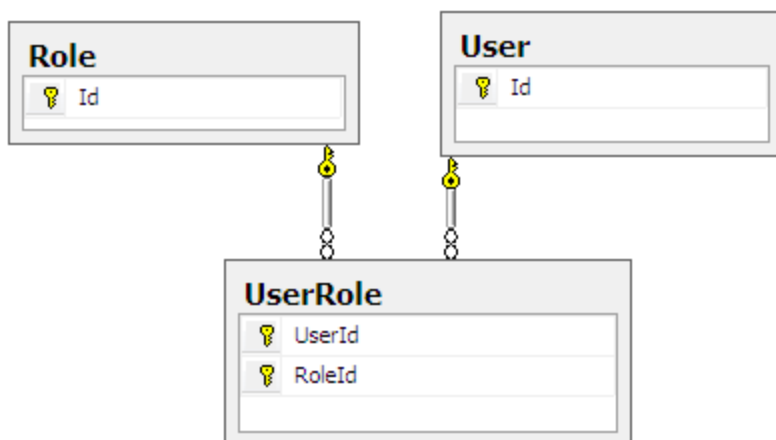
Next: Many to Many Relationships

Many to Many Relationships

Many to Many Relationships

This video covers the basics of many to many relationships in PLINQO. Check it out!

Almost every project requires the use of many to many relationships and LINQ to SQL does not support it. As you probably guessed, PLINQO does! Let's go through a quick sample of a many to many scenario in PLINQO. Below is a diagram of a many to many relationship between the User and Role entities.



The UserRole table before any many to many code is executed.

UserId	RoleId
NULL	NULL

The code here adds a User to a Role.

```
using (var context = new TrackerDataContext())
{
    User u = context.User.GetByKey(1);
    Role r = context.Role.GetByKey(1);
    u.RoleList.Add(r);
    context.SubmitChanges();
}
```

The result in the database after the code is executed.

UserId	RoleId
1	1

Read more about how PLINQO implemented many to many relationships.

Next: Query Extensions

Query Extensions

Query Extensions

For more information on query extensions, check out this PLINQO feature video.

Each of the query extension methods generated are composable meaning you can chain the methods together to build the desired functionality out of smaller parts and take full advantage of reuse.

```
task = context.Task.GetByKey(1);  
tasks = context.Task.ByAssignedId(1).ByStatusId(1);  
List taskList = tasks.ToList();
```

Comparison Operators

We have found that querying equality only solves a portion of your querying needs and that is not going to cut it. We want all of our querying needs met. To do that, the PLINQO query extensions use containment operators. For example, the database contains a person table and the requirement is to find all the record with an age greater than 21, there is a comparison operator for that.

```
db.User.ByAge(21, ComparisonOperator.GreaterThan);
```

The comparison operators provide all the options to filter data(Equals, GreaterThan, GreaterThanOrEquals, LessThan, LessThanOrEquals and NotEquals). PLINQO makes it a lot simpler and easier to meet all requirements when filtering data.

Next: [Caching](#)

Performance Features

The first thing most people ask about PLINQO is what kind of performance updates have we made since LINQ to SQL? The following pages will go over such updates.

- [Caching](#)
- [Future Queries](#)
- [LINQ to SQL Profiler](#)
- [Performance Improvements](#)

Caching

Caching

Watch this video for an overview of Caching!

A vital aspect to any framework is the option to cache data. Anytime the same data is retrieved over and over, precious time and cpu cycles are wasted. Sometimes simple caching works, sometimes you need more and sometimes how you want to cache depends on your situation. PLINQO understands that the caching needs of each application are completely different and that the application foundation is expected to handle all kinds of situations simply and effectively. PLINQO provides query result cache, cache profiles, cache groups, cache provider support and a cache manager making it easy to customize, configure and alter caching solutions that will meet the ever changing needs of your application. From configuring cache providers such as `HttpCacheProvider` or our very own `Memcached` provider, caching with PLINQO is simple and effective.

The Caching libraries are located inside of the Tracker Sample Applications that can be found [here](#)

Query Result Cache

PLINQO provides the option to cache any LINQ to SQL query result. PLINQO takes advantage of the `System.Web.Caching.Cache` class to store the results of the LINQ to SQL queries with the option to use a sliding or absolute expiration and set a priority on the items stored in cache.

To cache query results in PLINQO, use the `FromCache` extension method located in the `CodeSmith.Data.Linq` namespace. Below is a sample caching query results using all the different options provided by PLINQO.

Simply, execute the LINQ to SQL query as you normally would, then append the FromCache extension and tell PLINQO how the results should be cached.

```
//query is cached using the default settings
var tasks = context.Task.ByAssignedId(UserId).FromCache();

//query result is now cached 300 seconds
var approvedUsers = context.User.ByIsApproved(true).FromCache(300);
```

Link can also be cached using the same syntax of FromCache extension method. Each query is cached separately to provide more flexibility.

Cache Profiles

The way we want to cache often depends on the scenario. A slow changing dropdownlist has different caching needs than a task list for a user and possibly different provider needs. PLINQO's Caching Profiles are a nice, simple way to configure cache settings to be used to meet the needs of these types of scenarios. Below is a sample Cache Profile configuration:

```
< configuration >
  < configSections >
    < section name="cacheManager" type="CodeSmith.Data.Caching.CacheManagerSection,
CodeSmith.Data" / >
  < /configSections >
  < cacheManager defaultProvider="HttpCacheProvider" defaultProfile="Short" >
    < profiles >
      < add name="Short" description="Short cache" duration="0:5:0" / >
      < add name="ShortSliding" description="Short sliding cache" duration="0:5:0"
mode="Sliding" group="MyGroup" / >
      < add name="Long" description="Long cache" duration="1:0:0" / >
    < /profiles >
    < providers >
      < add name="MemcachedProvider" description="MemcachedProvider"
type="CodeSmith.Data.Memcached.MemcachedProvider, CodeSmith.Data.Memcached" / >
      < add name="HttpCacheProvider" description="HttpCacheProvider"
type="CodeSmith.Data.Caching.HttpCacheProvider, CodeSmith.Data" / >
    < /providers >
  < /cacheManager >
```

```
...
< /configuration >
```

You can also specify your memcached server information like this:

```
< configuration >
  < configSections >
    < section name="cacheManager" type="CodeSmith.Data.Caching.CacheManagerSection,
CodeSmith.Data" / >

    < sectionGroup name="enyim.com" >
      < section name="memcached" type="Enyim.Caching.Configuration.MemcachedClientSection,
Enyim.Caching" / >
    < /sectionGroup >
  < /configSections >
  < enyim.com >
    < memcached >
      < servers >
        < add address="127.0.0.1" port="11211" / >
        < add address="127.0.0.2" port="11211" / >
      < /servers >
      < socketPool minPoolSize="10" maxPoolSize="100" connectionTimeout="00:00:10"
deadTimeout="00:02:00" / >
    < /memcached >
  < /enyim.com >
  < cacheManager defaultProvider="HttpCacheProvider" defaultProfile="Short" >
    < profiles >
      < add name="Short" description="Short cache" duration="0:5:0" / >
      < add name="ShortSliding" description="Short sliding cache" duration="0:5:0"
mode="Sliding" group="MyGroup" / >
      < add name="Long" description="Long cache" duration="1:0:0" / >
    < /profiles >
    < providers >
      < add name="MemcachedProvider" description="MemcachedProvider"
type="CodeSmith.Data.Memcached.MemcachedProvider, CodeSmith.Data.Memcached" / >
      < add name="HttpCacheProvider" description="HttpCacheProvider"
type="CodeSmith.Data.Caching.HttpCacheProvider, CodeSmith.Data" / >
    < /providers >
  < /cacheManager >

  ...
```

```
< /configuration >
```

This configuration uses two providers, the default `HttpCacheProvider` and our very own `Memcached` provider. Also, we define a few named profiles. For short duration of cache, the `HttpCacheProvider` is used and anything being cached for a long period of time, the `Memcached` provider will be used. If no profile is specified when caching the data, the "Short" profile will be used. When storing items in cache, settings are a worry of the past, just tell PLINQO which profile to use.

```
var project = db.Project
    .ById(projectId.Value)
    .FromCacheFirstOrDefault("Short");
```

Cache Groups

How cool would it be to store items in cache as a group. Well, guess what! PLINQO is cool! When storing items in cache, just tell PLINQO the query result needs to belong to a group and specify the name. Invalidating cache is where the coolness of grouping really shows up. No coupling of cache and actions taken on that cache when they are in a group. Check out this example

```
public ActionResult MyTasks(int userId)
{
    // will be separate cache for each user id, group all with name MyTasks
    var tasks = db.Task
        .ByAssignedId(userId)
        .ByStatus(Status.InProgress)
        .FromCache(CacheManager.GetProfile().WithGroup("MyTasks"));

    return View(tasks);
}

public ActionResult UpdateTask(Task task)
{
    db.Task.Attach(task, true);
    db.SubmitChanges();
}
```

```
// since we made an update to the tasks table, we expire the MyTasks cache
CacheManager.InvalidateGroup("MyTasks");
}
```

When a task is updated, all the method needs to invalidate is the MyTasks group. This rocks because UpdateTask isn't required to have knowledge of any related cached items, it just knows there is a group of data items that need to be invalidated because this method was executed.

Cache Manager

Ok. Cache profiles and groups are nice, but how does PLINQO let me use these sweet features? Well, obviously, a manager will be brought in to do all the work. :) PLINQO's cache manager provides a simple, intuitive interface to store, retrieve and remove items from the cache.

```
CacheManager.Set("user-17", user, 30);
User user = CacheManager.Get("user-17");
CacheManager.Remove("user-17");
```

No clutter in this API. It is simple, intuitive and easy to use and is only focused on the important things that need to be done to manage cache.

Next: Future Queries

Future Queries

Future Queries

Check out this video about Future Queries for some more information!

With PLINQO, the future is now! Build up a list of queries for the data that you need and the first time any of the results are accessed, PLINQO will retrieve all the data in one round trip to the database server. Reducing the number of trips to the database is a great optimization that makes for smoother, faster running applications. Using this feature is as simple as appending `.Future()` to the end of your queries. To use the Future Queries please make sure that you are importing the `CodeSmith.Data.Linq` Namespace. Here is a quick sample:

```
// build up multiple queries
var q1 = db.User
    .ByEmailAddress("one@test.com")
    .Future();

var q2 = db.Task
    .Where(t => t.Summary == "Test")
    .Future();

// this triggers the loading of all the future queries
var users = q1.ToList();
```

No data is retrieved until `q1.ToList();` is executed. At that time, PLINQO knows to execute all the future queries automatically. The data is both batched and not retrieved until it is needed.

Not only can you queue up execution of queries for the future, the results can be cached as well. Again, PLINQO makes things easy. Here's a quick look at `FutureCache`.

```
// cache these results for 120 seconds
var q1 = db.User
    .ByEmailAddress("one@test.com")
    .FutureCache(120);

var q2 = db.Task
    .Where(t => t.Summary == "Test")
    .FutureCache(120);

// this triggers the loading of all the future queries
var users = q1.ToList();
```

Queuing up for the future has never been easier!

The Details

Future queries are created with the extension methods `Future()`, `FutureFirstOrDefault()`, or `FutureCount()`.

```
var db = new TrackerDataContext();

// build up queries
var q1 = db.User
    .ByEmailAddress("one@test.com")
    .Future();

var q2 = db.Task
    .Where(t => t.Summary == "Test")
    .Future();
```

```
// this triggers the loading of all the future queries
var users = q1.ToList();
```

In the example above, there are 2 queries built up, as soon as one of the queries is enumerated, it triggers the batch load of both queries.

```
var db = new TrackerDataContext();

// base query
var q = db.Task.ByPriority(Priority.Normal);
// get total count
var q1 = q.FutureCount();
// get page
var q2 = q.Skip(pageIndex).Take(pageSize).Future();

// triggers sql execute as a batch
int total = q1.Value;
var tasks = q2.ToList();
```

In this example, we have a common senerio where you want to page a list of tasks. In order for the GUI to setup the paging control, you need a total count. With Future, we can batch together the queries to get all the data in one database call.

Future Query Cache

Future queries can also be cached using the same syntax of FromCache extension method. Each query is cached separately to provide more flexibility.

```
var db = new TrackerDataContext();

// cache for 120 seconds
CacheSettings cache = new CacheSettings(120);

// build up queries
```

```
var q1 = db.User
    .ByEmailAddress("one@test.com")
    .FutureCache(cache);

var q2 = db.Task
    .Where(t => t.Summary == "Test")
    .FutureCache(cache);

// this triggers the loading of all the future queries
var users = q1.ToList();
```

How It Works

Future queries work by creating the appropriate `IFutureQuery` object that keeps the `IQueryable`. The `IFutureQuery` object is then stored in `IFutureContext.FutureQueries` list on the `DataContext` that created the query. The `DataContext` must implement `IFutureContext`. Then, when one of the `IFutureQuery` objects is enumerated, it calls back to `IFutureContext.ExecuteFutureQueries()` via the `LoadAction` delegate. `ExecuteFutureQueries` builds a batch query from all the stored `IFutureQuery` objects. Finally, all the `IFutureQuery` objects are updated with the results from the query.

Limitations

Does not support anonymous types.

Next: LINQ to SQL Profiler

LINQ to SQL Profiler

LINQ to SQL Profiler

The LINQ to SQL profile is quick and easy to use! Check out this video for additional information.

LINQ to SQL Profiler is a real-time visual profiler allowing you to gain valuable insight and perspective into an application's database interaction. It makes it easy to identify your applications hot spots. Not only does LINQ to SQL Profiler show you what is going on, but it will take it a step further in suggesting possible changes that will make your code more efficient. LINQ to SQL Profiler is a must on the tool belt of anyone using PLINQO.

To get started using LINQ to SQL Profiler, just follow these steps:

1. Download the trial from <http://l2sprof.com>
2. In your Visual Studio Project, add a reference to:

```
HibernatingRhinos.Profiler.Appender.dll
```

3. Add the following code in the startup portion of your application

```
LinqToSqlProfiler.SetupProfilingFor(TrackerDataContext.MappingCache);  
LinqToSqlProfiler.Initialize();
```

4. Start up LINQ to SQL Profiler
5. Start your application
6. Watch the action begin!

It's that simple to start profiling PLINQO!

Next: Performance Improvements

Performance Improvements

Performance Improvements

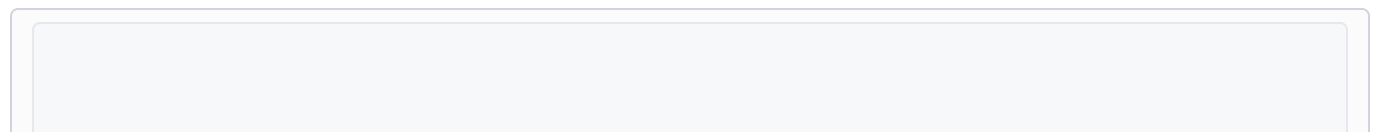
Check out this video to see some of our performance improvements in action!

Batch updates and deletes

Deleting and updating are major parts of any application. PLINQO now provides an optimal solution whether working with one entity LINQ to SQL requires a lot of work to delete or update an entity and no support to delete or update multiple entities at one time. For single deletes, the object must be retrieved and populated in order to delete and update. Two calls to the database to delete or update an entity when you already have the information you need! PLINQO eliminates the first call by sending deletes and updates directly eliminating the need to do the extra work. Thus, performance is easily improved, plus time is saved with fewer lines of code.

Multiple updates and deletes are also a downside for most ORMs and LINQ to SQL is no different. When deleting or updating multiple entities without PLINQO, multiple statements are issued when a single SQL statement will get the job done a lot more efficiently. PLINQO offers the ability to create multiple updates and deletes based on a filter. Below is a look at the different features available for deleting and updating.

Deleting



```
//Manager classes delete method
context.Manager.User.Delete(1);
//Query extension delete method
context.User.Delete(1);

//works for managers and queries
context.User.Delete(u => u.FirstName == "firstname");
```

Update

```
context.Task.Update(t => t.StatusId == 1, t2 => new Task {StatusId = 2});

IQueryable users = context.User.Where(u => u.FirstName == "firstname");
context.User.Update(users, u => new User {FirstName = "newfirstname"});
```

As you can see, there is no need for `SubmitChanges()`. The actions are taken immediately and much better performance will be seen thanks to PLINQO's batch deleting and updating capabilities.

Stored Procedures with multiple result sets

There are times when it is more efficient to batch queries into one stored procedure in order to optimize the number of round trips to the database. During generation, PLINQO recognizes all stored procedures that return multiple results and provides a nice way to handle the results.

Here is a look at how this can be done. The following stored procedure is created.

```
Create Procedure [dbo].[ReturnMultiplResultSets]
As
Select * From [User]
Select * From [Task]
```

Now we have a stored procedure returning multiple result sets and here is how PLINQO will handle it.

```
IMultipleResults results = context.ReturnMultiplResultSets();  
List users = results.GetResult().ToList();  
List tasks = results.GetResult().ToList();
```

It doesn't get any easier than this.

Batch Queries

When heading to the video game store. money in hand, ready to purchase Call of Duty, Madden 10 and Tiger Woods 10, I am sure you do not make 3 trips to the store to buy the games, but take care of it in one purchase. Making three trips would be incredibly inefficient. So why is this deemed acceptable by LINQ to SQL when pulling back data from the database. PLINQO finds this unacceptable! PLINQO supports batching queries making it possible to greatly optimize the time it takes completing operations. ExecuteQuery on the data context makes this possible. Here is an example that batches up Select * From User and Select * From Task into one trip to the database.

```
var q1 = from u in context.User select u;  
var q2 = from t in context.Task select t;  
IMultipleResults results = context.ExecuteQuery(q1, q2);  
List users = results.GetResult().ToList();  
List tasks = results.GetResult().ToList();
```

Easier Serialization

Serialization of entities has had developers running in circles from the early days of LINQ to SQL until now. PLINQO eliminates this frustration by eliminating the possibility of circular references and taking advantage of WCF datacontract serialization. Here is a sample of serializing an object using PLINQO.

```
Task task = context.Task.GetByKey(1);  
string xml = task.ToXml();  
byte[] b = task.ToBinary();
```

Deserialization is also made simple with PLINQO

```
task = Task.FromXml(xml);  
task = Task.FromBinary(b);
```

Next: Customization & Refactoring

Project Features

The following pages go over some major PLINQO features.

- [Customization & Refactoring](#)
- [Data Services & WCF](#)
- [Quick Start](#)

Customization & Refactoring

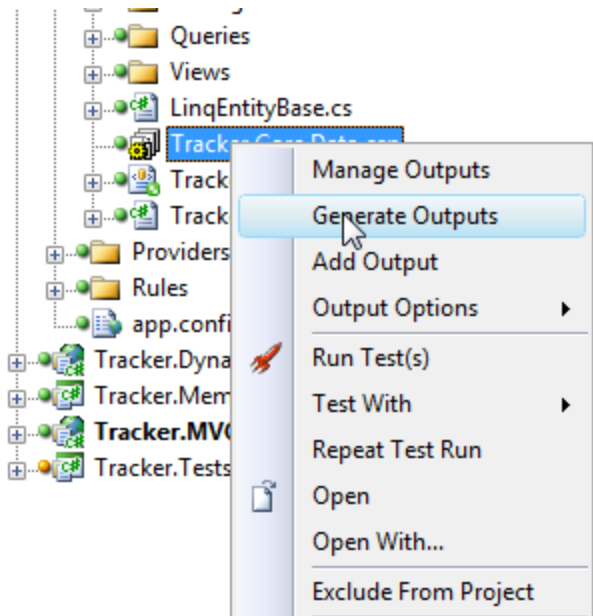
Customization & Refactoring

Please watch this video for an in-depth overview of Customization & Refactoring.

Unlike standard LINQ to SQL, PLINQO supports customizing and refactoring your DBML.

Easy Synchronization of Database Changes

Changes are an inevitable part of any project. The LINQ to SQL designer in Visual Studio provides a lot of support for manipulating entities, but does NOT make it easy when a refactor is required. The LINQ to SQL designer requires that the entity be dropped and recreated to generate the necessary updates. When the entity is dropped, any updates made for that entity are also lost and now must be re-created. This is time consuming, tedious and results in work being done over and over again. With PLINQO, make changes, right-click, generate, DONE! PLINQO makes it that easy. PLINQO supports syncing the database with the dbml while preserving any manual dbml updates.



Metadata Customization/Syncing

The System.ComponentModel.DataAnnotations assembly introduced in .NET 3.5 SP1 provides another opportunity for PLINQO to make life easier. PLINQO generates and maintains a metadata class inside each entity. This metadata class allows you to easily add extra knowledge about your domain objects and their properties. Each time PLINQO is re-generated, DBML and database changes are synced while preserving any custom changes using a new feature in CodeSmith called the insert class merge strategy. The process of maintaining metadata on each entity in the system now can be as simple as regenerating your metadata classes.

Here is a picture of a MetaData class that sits inside a User class.

```
[CodeSmith.Data.Audit.Audit]
private class Metadata
{
    // Only Attributes in the class will be preserved.

    public int Id { get; set; }

    [Required]
    public string UserName { get; set; }

    [Required]
    [DataType(System.ComponentModel.DataAnnotations.DataType.Password)]
    public string Password { get; set; }

    [DataType(System.ComponentModel.DataAnnotations.DataType.EmailAddress)]
    public string EmailAddress { get; set; }
}
```

```

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public System.Data.Linq.Binary Avatar { get; set; }

    [Now(EntityState.New)]
    [CodeSmith.Data.Audit.NotAudited]
    public System.DateTime CreatedDate { get; set; }

    [Now(EntityState.Dirty)]
    [CodeSmith.Data.Audit.NotAudited]
    public System.DateTime ModifiedDate { get; set; }

    public System.Data.Linq.Binary RowVersion { get; set; }

    public EntitySet AssignedTaskList { get; set; }

    public EntitySet CreatedTaskList { get; set; }

    public EntitySet UserRoleList { get; set; }
}

```

Having this pre-configured makes it obvious where to go when extra knowledge is needed about the model. ASP.NET Dynamic Data attributes such as what control should display the data, data types, how to validate fields and actions taken on properties are just a few of the attributes available for use within the metadata class.

Next: Data Services & WCF

Data Services & WCF

Data Services & WCF

Please watch this video for an in-depth overview of Data Services & WCF.

Data Services

LINQ to SQL does not support Data Services by default. However, all it takes to use data services with PLINQO is setting the `IncludeDataServices` property on the Entities template to true and let PLINQO take care of setting up your project to support Data Services.WCF

WCF

PLINQO takes care of all the tedious work of setting up the `DataContract` and `DataMember` attributes for Windows Communication Foundation (WCF). The `IncludeDataContract` option on the Entities template tells PLINQO to generate Data Contract and Data Member Attributes and makes using Windows Communication Foundation (WCF) with PLINQO painless by doing all the leg work.

Next: [The Quick Start](#)

The Quick Start

The Quick Start

The PLINQO Quick Start makes starting out IN PLINQO quick and easy! Check out this video to learn more.

A working project in seconds! Think about the time it normally takes to get a project up and running (creating the solution, add the projects, organize your code, etc). PLINQO has made this process as easy as selecting a database. Below is a picture of the CodeSmith Quick Start properties. As usual with CodeSmith, you are in control and can change any of the QuickStart properties to generate the project solution you want. Once you click generate, you can checkout all of the features of PLINQO by either playing around with the Dynamic Data interface, creating tests or moving toward completion of your project.

Only the SourceDatabase property is required.

CodeSmith - QuickStart.cst

1. Database

SourceDatabase	
----------------	--

2. Solution

Language	CSharp
LaunchVisualStudio	True
Location	
SolutionName	

3. Data Project

CopyTemplatesToFolder	False
DataProjectName	
QueryPattern	ManagerClasses

4. Interface Project

IncludeDataServices	True
InterfaceProjectName	
ProjectType	DynamicDataWebApp

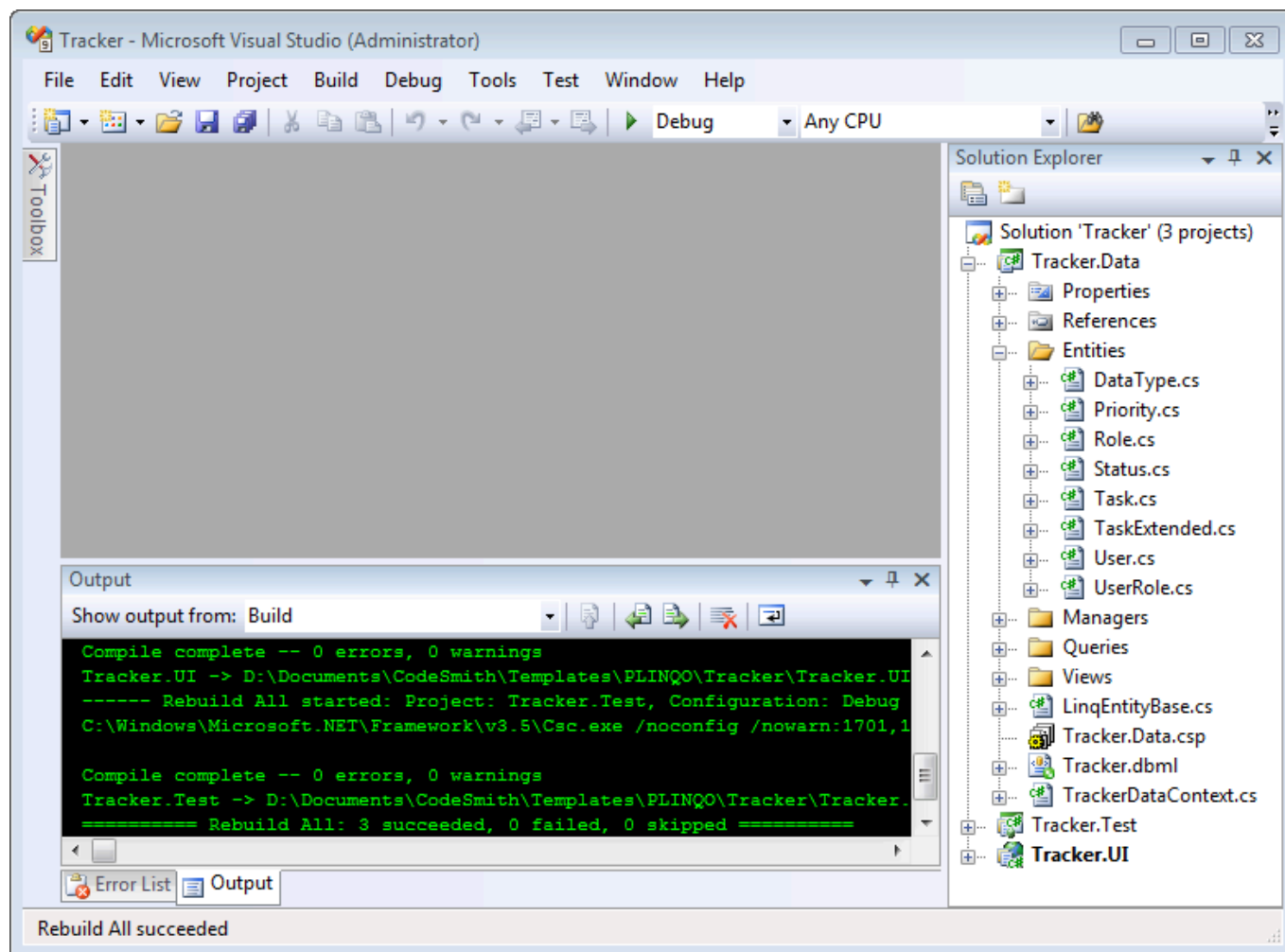
5. Test Project

IncludeTestProject	True
TestProjectName	

Copy Templates To Folder
Include a copy of the LinqToSql Templates in the project directory

>> Generate Cancel

Clicking generate produces the solution below. PLINQO even opens it for you!



Complete Email Client

LINQ to SQL does not put much thought into the user experience when database changes or customization is needed. You have to create your own classes when changes are needed. PLINQO is smart enough to know that the generated code is not the entire solution, but the rock solid foundation. Tremendous thought has been put into project structure and file organization. The larger a project gets, the more important this becomes. PLINQO creates partial classes and 2 files for each entity, manager and query extension. One file is an actively generated file and the other is specifically for customizing and extending the PLINQO architecture.

Using the Templates

The following section will cover how to use the various templates that ship with PLINQO.

Manual Generation

Use the following steps to get started using the PLINQO for Linq to Sql templates if you are not planning on using the QuickStart template.

1. Create a new Class Library project in Visual Studio.
2. Add a new CodeSmith project file to the Visual Studio project.
3. Add a new Output to the project file for the Dbml.cst template.
4. Add another Output to the project file for the Entities.cst template.
5. Optionally, Add an Output for the Managers.cst template.
6. Set the Sample.csp -> Output Options -> Add Outputs to Project to unchecked. The templates update the project for you.
7. Finally, Generate the Outputs.

DBML Template

The Dbml.cst template is used to create a LINQ to SQL dbml file. The file conforms to the Microsoft DbmlSchema.xsd schema. This is the same document that the LINQ to SQL designer uses. The generated dbml file from this template can also be edited from the LINQ to SQL designer.

The template will create a new file if it doesn't exist. If the file does exist, the template will read it in and update it. This allows you to make changes to the file and not have it overwrite if the template is re-ran. However, only some of the attributes are safe from overwriting. Here is a list of safe attributes. They will be listed as an xpath.

Safe Attributes to change in the Dbml file ...

- Database/@Class - The name of the DataContext class that will be generated.
- Database/@EntityNamespace - The namespace for the entity classes.
- Database/@ContextNamespace - The namespace for the DataContext class.
- Table/@Member - The property name for the table in the DataContext class.
- Type/@Name - The name of the entity class.
- Column/@Member - The property name for the column in the entity class.
- Column/@Storage - The private field LINQ to SQL will use to assign values to.
- Association/@Member - The property name for this association.
- Association/@Storage - The private field LINQ to SQL will use to assign values the association to.
- Association/@DeleteOnNull - Deletes row when association is set to null. Value will not be persisted if the CST's IncludeDeleteOnNull Property is set to false.
- Function/@Method - The name of the method for the database procedure.
- Parameter/@Parameter - The method argument name that maps to the database procedure parameter.

Warning: Be aware that the template will drop tables, columns and associations that it did not find in the database.

Properties on the Dbml.cst template:

The **EnumList**, **IgnoreList** and **IncludeList** properties use a list of **Regular Expressions** that match the full name of a Table, View, Command.

Here is a common Regex pattern that will match a table by name (E.G., ^owner.TableName\$). For Example, the following regex (^**dbo.Account**\$) will match the Account table.

We recommend using **Expresso** to build your regular expression lists.

Property	Description
----------	-------------

1.Database	
CleanExpression	List of regular expressions to clean table, view, column and procedure names. Any matched text found will be removed from the name.
EnumList	List of regular expressions to identify an Enum Table.
IgnoreList	List of regular expressions used to ignore tables, views and procedures when generating mapping.
IncludeList	List of regular expressions used to include tables, views and procedures when generating mapping. When this property is set the IgnoreList property will be ignored and only tables included in this list will be used.
IncludeFunctions	Include stored procedures and user functions in mapping.
IncludeViews	Include views in mapping.
SourceDatabase	The source database to generate the dbml file for.
2.Class	
ContextNamespace	The namespace to use for the context class file.
EntityBase	The base class for the entity.
EntityNamespace	The namespace to use for the entity class files.
IncludeDeleteOnNull	Templates will generated DeleteOnNull Attribute for Associations.
Naming Conventions	A Menu of options for naming objects in the dbml.
AssociationNaming	Add a list suffix or pluralize associations to an object.
EntityNaming	Generate entity objects in singular or plural form.
TableNaming	The table naming convention followed in the source database.
3.Enum	
NameColumn	List of regular expressions to identify an Enum Table's Name Column.
DescriptionColumn	List of regular expressions to identify an Enum Table's Description Column.
4.Mapping	
DbmlFile	The path to the dbml file to generate.

Entities Template

The entities template generates the entity classes needed by LINQ. The classes are generated from a dbml file. You can modify the names for classes and properties by editing the dbml file. See [DBML Template](#) for a list of safe attributes to change in the dbml file.

The template will generate 2 files for every Type in the dbml file. One file will be the generated partial class that can not be changed as it is overwritten when the template is re-ran. It will have the following file name... <entity>.Generated.cs

The second file is a partial class that can be modified as it will not be re-generated. You can implement the partial methods in this file. Some partial method stubs are created by default. This file will be named... <entity>.cs

If you set the project file property on the template, the generated files will be added to the project. The file that can not be modified will be hidden under the file that can be changed.

Properties on the Entities.cst template:

Property	Description
1.Mapping	
DbmlFile	The path to the dbml file used generate the entities from.
2.Class	
AssociationNamingSuffix	Suffix for Many To Many Associations.
AuditingEnabled	Flag indicating if the default state of the AuditingEnabled property.
BaseDirectory	The folder to save the generated DataContext and base class files.
DefaultConnectionString	Default Connection String Name for DatabaseContext.
Framework	The version of the .Net Framework that is being used. Version 35_1 is necessary for dynamic data and data services attributes.
IncludeDataContract	Include serialization attributes on the generated entities. The attributes generated are based on the framework attribute above.
IncludeDataRules	Include CodeSmith.Data rules.
IncludeDataServices	Include Data Services attributes on the generated entities.
OutputDirectory	The folder to save the generated files.

ViewDirectory	The folder to save the generated view files.
---------------	--

Managers Template

The manager template is for helping you get started with business logic for the LINQ entities. The managers will have common queries that are created from keys and indexes on the table. The manager will also have rules for the entity properties to make sure required fields are not null and that the length of a string does not exceed the max length the column allows.

The template works by creating a second partial class that has a Manager property. The manager will then have a property for each entity that has a manager. Here is a sample of the syntax for using the managers:

```
SampleDataContext db = new SampleDataContext();
// use the primary key
Task task = db.Manager.Task.GetByKey(taskId);
// use a foreign key
var myTasks = db.Manager.Task.GetByAssignedID(userId);
// the methods return IQueryable so you can add expressions
var openTasks = db.Manager.Task.GetByStatusID(statusId).OrderBy(t => t.CreateDate);
```

The manager will add business rules to your entities based on metadata attributes. These rules are based on the same rules found in the DataAnnotations.

```
private class Metadata
{
    // Only Attributes in the class will be preserved.
    public int OrderId { get; set; }
    [Required]
    [StringLength(20)]
    public string UserId { get; set; }
    [Range (typeof(DateTime), "10-31-2008", "12-25-2200")]
    public System.DateTime OrderDate { get; set; }
    .....
}
```

The manager also provides a business rules engine to your entities. In addition to the default validation rules that are generated, you can add custom rules by implementing the AddRules partial method in the custom entity class.

```
static partial void AddRules()
{
    // Rule allows the Name property to be a max of 150 characters.
    RuleManager.AddShared<Task>(new LengthRule("Name", 150));
    // Rule that validates the value of the property using regex.
    RuleManager.AddShared<Task>(new RegexRule("Name", ".*"));
    // Rule allows only users in certain security roles to update.
    RuleManager.AddShared<Task>(new UpdateRule(new string[]
{ "Administrator", "Updaters" }));
}
```

Properties on the Managers.cst template:

Property	Description
1.Database	
SourceDatabase	The source database to keys and indexes from for generating the manager classes.
2.Class	
Framework	The version of the .Net Framework that is being used.
3.Mapping	
DbmlFile	The path to the dbml file used generate the manager classes from.
4.Manager	
DataContextFile	The location of the DataContext File.
DataContextName	The class name of the DataContext that supports the managers.
ManagerDirectory	The folder to save the generated manager files.
ManagerNamespace	The namespace to use for the generated manager class files.
MethodPrefix	The prefix of query method names. For example, 'GetBy'.
MethodKeySuffix	The suffix of the primary key query method names. For example, 'Key'.

Query Extensions Template

The queries template is an optional alternative to the manager template. If you don't want to use the manager framework, you can use this template to generate some common queries for an entity. While its possible to use both the manger and query templates, they do duplicate some functionality. The template works by generating an extension class for `Table<Entity>`. This allows the queries to be off the `DataContext`.

```
SampleDataContext db = new SampleDataContext();
Task task = db.Task.GetByKey(1);
```

This is an example of what the extension class looks like.

```
/// <summary>
/// The query extension class for Task.
/// </summary>
public static partial class TaskQueryExtension
{
    /// <summary>
    /// Gets an instance by the primary key.
    /// </summary>
    public static Task GetByKey(this Table<Task> entity, int taskID)
    {
        if (entity.Context.LoadOptions == null)
            return Query.GetByKey.Invoke((SampleDataContext)entity.Context, taskID);
        else
            return entity.FirstOrDefault(t => t.TaskID == taskID);
    }
    /// <summary>
    /// Gets a query by an index.
    /// </summary>
    public static IQueryable<Task> GetByStatusID(this Table<Task> entity, int statusID)
    {
        if (entity.Context.LoadOptions == null)
            return Query.GetByStatusID.Invoke((SampleDataContext)entity.Context, statusID);
        else
            return entity.Where(t => t.StatusID == statusID);
    }
}
#region Query
```

```

    /// <summary>
    /// A private class for lazy loading static compiled queries.
    /// </summary>
    private static partial class Query
    {
        internal static readonly Func<SampleDataContext, int, Task> GetByKey =
            CompiledQuery.Compile(
                (SampleDataContext db, int taskID) =>
                    db.Task.FirstOrDefault(t => t.TaskID == taskID));
        internal static readonly Func<SampleDataContext, int, IQueryable<Task>> GetByStatusID
        =
            CompiledQuery.Compile(
                (SampleDataContext db, int statusID) =>
                    db.Task.Where(t => t.StatusID == statusID));
    }
    #endregion
}

```

Properties on the Queries.cst template:

Property	Description
1.Database	
SourceDatabase	The source database to keys and indexes from for generating the manager classes.
2.Mapping	
DbmlFile	The path to the dbml file used generate the query classes from.
3.Query	
QueryDirectory	The folder to save the generated query extension files.
MethodPrefix	The prefix of query method names. For example 'GetBy'.
MethodKeySuffix	The suffix of the primary key query method names. For example 'Key'.

Quick Start Template

The QuickStart.cst template is used to create a basic project using Linq to Sql. The template will create a visual studio solution with up to three included projects: A Data Project, an Interface Project, and a Test Project.

Properties on the QuickStart.cst template:

Property	Description
1.Database	
SourceDatabase	The Database that will be accessed by the website.
2.Solution	
Language	The language that the project will be generated in.
LaunchVisualStudio	Should visual studio launch when the project finishes generating.
Location	The root directory of the generated solution.
SolutionName	The name of the Visual Studio Solution.
VisualStudioVersion	The version of Visual Studio you will be using.
3.Data Project	
CopyTemplatesToFolder	Create a copy of the Linq to Sql Directory and its contents to the data project directory.
DataProjectName	The name of the data project directory and project.
QueryPattern	The type of pattern to use for automatically generated queries.
4.Interface Project	
IncludeDataServices	Include Data Service Files and References in the interface project.
InterfaceProjectName	The name of the interface project directory and project.
ProjectType	The type of interface project to be created. Can be none.
5.TestProject	
IncludeTestProject	Include a test project in the solution.
TestProjectName	The name of the test project directory and project.

Downloads

Latest Release

The latest version of PLINQO is always available on GitHub releases: [Download latest version of PLINQO](#)

PLINQO requires the latest version of [CodeSmith Generator Professional](#)

Source Code

If you like to get into the code and see how things work, the PLINQO source is available to view online or checkout using git from [here](#).

Sample Projects

You can find sample projects using PLINQO For Linq-to-SQL on our Google code site. Click on a link below to view the CSharp or Visual Basic samples.

- **CSharp**
 - Tracker Sample
 - Petshop Sample
- **Visual Basic**
 - Tracker Sample
 - Petshop Sample

Contributions

Many have contributed to the PLINQO Framework either by getting the word out or by providing patches, ideas and concepts that have become part of the PLINQO framework. The contributions are listed below. Thank you to each and everyone of you that have contributed in one way or another.

Blake Niemyjski

[Extending the CodeSmith Generator PLINQO Templates](#)

[How-to: Preserve column names in generated PLINQO entities.](#)

Terry Aney

[Batch Updates and Deletes](#)

[LINQ to SQL Batch UpdatesDeletes Fix for Could not translate expression](#)

Tom DuPont

[Advanced Future Queries, Part 2](#)

[Advanced Future Queries, Part 1](#)

[Query Extension Support OR Operators](#)

[PLINQO Method Prefixes](#)

[How PLINQO Improves Performance](#)

Russ Thomas

[PLINQO Review](#)

[PLINQO - Converting a project from LINQ to SQL](#)

PLINQO Blog Posts

[Visual Studio Magazine - LINQ to SQL Lives On](#)

[Tony Sneed - Who is improving LINQ to SQL](#)

[Kevin Lawry - Does LINQ StinQ? Not with PLINQO](#)

[InfoQ Interview with Eric J. Smith](#)

[Pete Montgomery - Query Result Cache](#)

[LINQ: BUILDING AN IQUERYABLE PROVIDER](#)

Testimonials

"PLINQO is a great example of the kind of additions we can expect from CodeSmith and others to add value to Microsoft's LINQ technology."

~ Robert Bazinet

"If you've been hesitant to try LINQ because of 'black box' or code re-generation concerns, now is the time to give it a shot. Download a trial of CodeSmith and try PLINQO."

~ Kevin Lawry

"I'm using PLINQO to create LINQ to SQL classes! Makes one file per class and has built-in caching! Now I can't live without it."

~ John Bledsoe