Table of Contents

LINQ to SQL

5/10/2017 • 1 min to read • Edit Online

LINQ to SQL is a component of .NET Framework version 3.5 that provides a run-time infrastructure for managing relational data as objects.

> **NOTE**
>
> Relational data appears as a collection of two-dimensional tables (*relations* or *flat files*), where common columns relate tables to each other. To use LINQ to SQL effectively, you must have some familiarity with the underlying principles of relational databases.

In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer. When the application runs, LINQ to SQL translates into SQL the language-integrated queries in the object model and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back to objects that you can work with in your own programming language.

Developers using Visual Studio typically use the Object Relational Designer, which provides a user interface for implementing many of the features of LINQ to SQL.

The documentation that is included with this release of LINQ to SQL describes the basic building blocks, processes, and techniques you need for building LINQ to SQL applications. You can also search the MSDN Library for specific issues, and you can participate in the LINQ Forum, where you can discuss more complex topics in detail with experts. Finally, the LINQ to SQL: .NET Language-Integrated Query for Relational Data white paper details LINQ to SQL technology, complete with Visual Basic and C# code examples.

In This Section

Getting Started
Provides a condensed overview of LINQ to SQL along with information about how to get started using LINQ to SQL.

Programming Guide
Provides steps for mapping, querying, updating, debugging, and similar tasks.

Reference
Provides reference information about several aspects of LINQ to SQL. Topics include SQL-CLR Type Mapping, Standard Query Operator Translation, and more.

Samples
Provides links to Visual Basic and C# samples.

Related Sections

LINQ (Language-Integrated Query)
Provides an overview of LINQ technologies.

LINQ
Describes LINQ technologies for Visual Basic users.

LINQ to ADO.NET

Links to the ADO.NET portal.

LINQ to SQL Walkthroughs
Lists walkthroughs available for LINQ to SQL.

Downloading Sample Databases
Describes how to download sample databases used in the documentation.

LinqDataSource Technology Overview
Describes how the LinqDataSource control exposes Language-Integrated Query (LINQ) to Web developers through the ASP.NET data-source control architecture.

Getting Started

5/1/2017 • 1 min to read • Edit Online

By using LINQ to SQL, you can use the LINQ technology to access SQL databases just as you would access an in-memory collection.

For example, the `nw` object in the following code is created to represent the `Northwind` database, the `Customers` table is targeted, the rows are filtered for `Customers` from `London`, and a string for `CompanyName` is selected for retrieval.

When the loop is executed, the collection of `CompanyName` values is retrieved.

```
// Northwnd inherits from System.Data.Linq.DataContext.
Northwnd nw = new Northwnd(@"northwnd.mdf");
// or, if you are not using SQL Server Express
// Northwnd nw = new Northwnd("Database=Northwind;Server=server_name;Integrated Security=SSPI");

var companyNameQuery =
    from cust in nw.Customers
    where cust.City == "London"
    select cust.CompanyName;

foreach (var customer in companyNameQuery)
{
    Console.WriteLine(customer);
}
```

```
' Northwnd inherits from System.Data.Linq.DataContext.
Dim nw As New Northwnd("c:\northwnd.mdf")
' or, if you are not using SQL Server Express
' Dim nw As New Northwnd("Database=Northwind;Server=dschwart7;Integrated Security=SSPI")

Dim companyNameQuery = _
    From cust In nw.Customers _
    Where cust.City = "London" _
    Select cust.CompanyName

For Each customer In companyNameQuery
    Console.WriteLine(customer)
Next
```

Next Steps

For some additional examples, including inserting and updating, see What You Can Do With LINQ to SQL.

Next, try some walkthroughs and tutorials to have a hands-on experience of using LINQ to SQL. See Learning by Walkthroughs.

Finally, learn how to get started on your own LINQ to SQL project by reading Typical Steps for Using LINQ to SQL.

See Also

LINQ to SQL
Introduction to LINQ
The LINQ to SQL Object Model

What You Can Do With LINQ to SQL

5/1/2017 • 3 min to read • Edit Online

LINQ to SQL supports all the key capabilities you would expect as a SQL developer. You can query for information, and insert, update, and delete information from tables.

Selecting

Selecting (*projection*) is achieved by just writing a LINQ query in your own programming language, and then executing that query to retrieve the results. LINQ to SQL itself translates all the necessary operations into the necessary SQL operations that you are familiar with. For more information, see LINQ to SQL.

In the following example, the company names of customers from London are retrieved and displayed in the console window.

```
// Northwnd inherits from System.Data.Linq.DataContext.
Northwnd nw = new Northwnd(@"northwnd.mdf");
// or, if you are not using SQL Server Express
// Northwnd nw = new Northwnd("Database=Northwind;Server=server_name;Integrated Security=SSPI");

var companyNameQuery =
    from cust in nw.Customers
    where cust.City == "London"
    select cust.CompanyName;

foreach (var customer in companyNameQuery)
{
    Console.WriteLine(customer);
}
```

```
' Northwnd inherits from System.Data.Linq.DataContext.
Dim nw As New Northwnd("c:\northwnd.mdf")
' or, if you are not using SQL Server Express
' Dim nw As New Northwnd("Database=Northwind;Server=dschwart7;Integrated Security=SSPI")

Dim companyNameQuery = _
    From cust In nw.Customers _
    Where cust.City = "London" _
    Select cust.CompanyName

For Each customer In companyNameQuery
    Console.WriteLine(customer)
Next
```

Inserting

To execute a SQL `Insert`, just add objects to the object model you have created, and call SubmitChanges on the DataContext.

In the following example, a new customer and information about the customer is added to the `Customers` table by using InsertOnSubmit.

```
// Northwnd inherits from System.Data.Linq.DataContext.
Northwnd nw = new Northwnd(@"northwnd.mdf");

Customer cust = new Customer();
cust.CompanyName = "SomeCompany";
cust.City = "London";
cust.CustomerID = "98128";
cust.PostalCode = "55555";
cust.Phone = "555-555-5555";
nw.Customers.InsertOnSubmit(cust);

// At this point, the new Customer object is added in the object model.
// In LINQ to SQL, the change is not sent to the database until
// SubmitChanges is called.
nw.SubmitChanges();
```

```
' Northwnd inherits from System.Data.Linq.DataContext.
Dim nw As New Northwnd("c:\northwnd.mdf")

Dim cust As New Customer With {.CompanyName = "SomeCompany", _
    .City = "London", _
    .CustomerID = 98128, _
    .PostalCode = 55555, .Phone = "555-555-5555"}
nw.Customers.InsertOnSubmit(cust)
' At this point, the new Customer object is added in the object model.
' In LINQ to SQL, the change is not sent to the database until
' SubmitChanges is called.
nw.SubmitChanges()
```

Updating

To `Update` a database entry, first retrieve the item and edit it directly in the object model. After you have modified the object, call SubmitChanges on the DataContext to update the database.

In the following example, all customers who are from London are retrieved. Then the name of the city is changed from "London" to "London - Metro". Finally, SubmitChanges is called to send the changes to the database.

```
Northwnd nw = new Northwnd(@"northwnd.mdf");

var cityNameQuery =
    from cust in nw.Customers
    where cust.City.Contains("London")
    select cust;

foreach (var customer in cityNameQuery)
{
    if (customer.City == "London")
    {
        customer.City = "London - Metro";
    }
}
nw.SubmitChanges();
```

```
Dim nw As New Northwnd("c:\northwnd.mdf")
Dim cityNameQuery = _
    From cust In nw.Customers _
    Where cust.City.Contains("London") _
    Select cust

For Each customer In cityNameQuery
    If customer.City = "London" Then
        customer.City = "London - Metro"
    End If
Next
nw.SubmitChanges()
```

Deleting

To `Delete` an item, remove the item from the collection to which it belongs, and then call SubmitChanges on the DataContext to commit the change.

NOTE

LINQ to SQL does not recognize cascade-delete operations. If you want to delete a row in a table that has constraints against it, see How to: Delete Rows From the Database.

In the following example, the customer who has `CustomerID` of `98128` is retrieved from the database. Then, after confirming that the customer row was retrieved, DeleteOnSubmit is called to remove that object from the collection. Finally, SubmitChanges is called to forward the deletion to the database.

```
Northwnd nw = new Northwnd(@"northwnd.mdf");
var deleteIndivCust =
    from cust in nw.Customers
    where cust.CustomerID == "98128"
    select cust;

if (deleteIndivCust.Count() > 0)
{
    nw.Customers.DeleteOnSubmit(deleteIndivCust.First());
    nw.SubmitChanges();
}
```

```
Dim nw As New Northwnd("c:\northwnd.mdf")
Dim deleteIndivCust = _
    From cust In nw.Customers _
    Where cust.CustomerID = 98128 _
    Select cust

If deleteIndivCust.Count > 0 Then
    nw.Customers.DeleteOnSubmit(deleteIndivCust.First)
    nw.SubmitChanges()
End If
```

See Also

Programming Guide
The LINQ to SQL Object Model
Getting Started

Typical Steps for Using LINQ to SQL

5/1/2017 • 3 min to read • Edit Online

To implement a LINQ to SQL application, you follow the steps described later in this topic. Note that many steps are optional. It is very possible that you can use your object model in its default state.

For a really fast start, use the Object Relational Designer to create your object model and start coding your queries.

Creating the Object Model

The first step is to create an object model from the metadata of an existing relational database. The object model represents the database according to the programming language of the developer. For more information, see The LINQ to SQL Object Model.

1. Select a tool to create the model.

Three tools are available for creating the model.

- The Object Relational Designer

  This designer provides a rich user interface for creating an object model from an existing database. This tool is part of the Visual Studio IDE, and is best suited to small or medium databases.

- The SQLMetal code-generation tool

  This command-line utility provides a slightly different set of options from the O/R Designer. Modeling large databases is best done by using this tool. For more information, see SqlMetal.exe (Code Generation Tool).

- A code editor

  You can write your own code by using either the Visual Studio code editor or another editor. We do not recommend this approach, which can be prone to errors, when you have an existing database and can use either the O/R Designer or the SQLMetal tool. However, the code editor can be valuable for refining or modifying code you have already generated by using other tools. For more information, see How to: Customize Entity Classes by Using the Code Editor.

2. Select the kind of code you want to generate.

- A C# or Visual Basic source code file for attribute-based mapping.

  You then include this code file in your Visual Studio project. For more information, see Attribute-Based Mapping.

- An XML file for external mapping.

  By using this approach, you can keep the mapping metadata out of your application code. For more information, see External Mapping.

  > **NOTE**
  >
  > The O/R Designer does not support the generation of external mapping files. You must use the SQLMetal tool to implement this feature.

- A DBML file, which you can modify before generating a final code file.

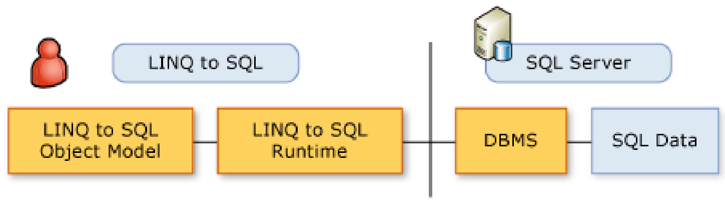
  This is an advanced feature.

3. Refine the code file to reflect the needs of your application.

For this purpose, you can use either the O/R Designer or the code editor.

Using the Object Model

The following illustration shows the relationship between the developer and the data in a two-tier scenario. For other scenarios, see N-Tier and Remote Applications with LINQ to SQL.



Now that you have the object model, you describe information requests and manipulate data within that model. You think in terms of the objects and properties in your object model and not in terms of the rows and columns of the database. You do not deal directly with the database.

When you instruct LINQ to SQL to either execute a query that you have described or call `SubmitChanges()` on data that you have manipulated, LINQ to SQL communicates with the database in the language of the database.

The following represents typical steps for using the object model that you have created.

1. Create queries to retrieve information from the database.

For more information, see Query Concepts and Query Examples.

2. Override default behaviors for Insert, Update, and Delete.

This step is optional. For more information, see Customizing Insert, Update, and Delete Operations.

3. Set appropriate options to detect and report concurrency conflicts.

You can leave your model with its default values for handling concurrency conflicts, or you can change it to suit your purposes. For more information, see How to: Specify Which Members are Tested for Concurrency Conflicts and How to: Specify When Concurrency Exceptions are Thrown.

4. Establish an inheritance hierarchy.

This step is optional. For more information, see Inheritance Support.

5. Provide an appropriate user interface.

This step is optional, and depends on how your application will be used.

6. Debug and test your application.

For more information, see Debugging Support.

See Also

Getting Started
Creating the Object Model
Stored Procedures

Downloading Sample Databases

5/1/2017 • 1 min to read • Edit Online

A number of samples and walkthroughs in the LINQ to SQL documentation use the Northwind sample database and SQL Server Express Edition. You can download these products free of charge from the Microsoft download site.

Downloading the Northwind Database

To download and install the Northwind sample database for SQL Server

1. Start Internet Explorer.
2. Go to the Northwind and Pubs Sample Databases Web site.
3. Click **Download**.
4. In the **File Download** dialog box, select **Save**.
5. After the file has downloaded, double-click the **Nwind.exe** file to install the database.

   By default, the database is installed at *drive*:\SQL Server 2000 Sample Databases.

Downloading SQL Server Express Edition

SQL Server Express Edition is available without charge, and you can redistribute it with applications. If you are using Visual Studio, SQL Server Express Edition is included in the Pro and greater editions.

To download and install SQL Server Express Edition

1. Start Internet Explorer.
2. Go to the Microsoft Download Center Web site.
3. In the keywords box, type **SQL Server Express**.
4. Click **Go**.
5. On the results page, click the link to the **Microsoft SQL Server 2005 Express Edition** download page.
6. Follow the installation instructions on the Web site.

Downloading Management Studio Express

If you want to modify a database that you have downloaded, you can access the database from **Server Explorer** in the Visual Studio integrated development environment (IDE), or use Microsoft SQL Server Management Studio Express (SSMSE).

To download Management Studio Express

- Follow the instructions at the SSMSE site.

See Also

Getting Started

Learning by Walkthroughs

5/1/2017 • 3 min to read • Edit Online

The LINQ to SQL documentation provides several walkthroughs. This topic addresses some general walkthrough issues (including troubleshooting), and provides links to several entry-level walkthroughs for learning about LINQ to SQL.

NOTE
The walkthroughs in this Getting Started section expose you to the basic code that supports LINQ to SQL technology. In actual practice, you will typically use the Object Relational Designer and Windows Forms projects to implement your LINQ to SQL applications. The O/R Designer documentation provides examples and walkthroughs for this purpose.

Getting Started Walkthroughs

Several walkthroughs are available in this section. These walkthroughs are based on the sample Northwind database, and present LINQ to SQL features at a gentle pace with minimal complexities.

A typical progression to follow would be as follows:

OBJECTIVE	VISUAL BASIC	C#
Create an entity class and execute a simple query.	Walkthrough: Simple Object Model and Query (Visual Basic)	Walkthrough: Simple Object Model and Query (C#)
Add a second class and execute a more complex query. (Requires completion of previous walkthrough).	Walkthrough: Querying Across Relationships (Visual Basic)	Walkthrough: Querying Across Relationships (C#)
Add, change, and delete items in the database.	Walkthrough: Manipulating Data (Visual Basic)	Walkthrough: Manipulating Data (C#)
Use stored procedures.	Walkthrough: Using Only Stored Procedures (Visual Basic)	Walkthrough: Using Only Stored Procedures (C#)

General

The following information pertains to these walkthroughs in general:

- Environment: Each LINQ to SQL walkthrough uses Visual Studio as its integrated development environment (IDE).
- SQL engines: These walkthroughs are written to be implemented by using SQL Server Express. If you do not have SQL Server Express, you can download it free of charge. For more information, see Downloading Sample Databases.

NOTE

LINQ to SQL walkthroughs use a file name as a connection string. Simply specifying a file name is a convenience that LINQ to SQL provides for SQL Server Express users. Always pay attention to security issues. For more information, see Security in LINQ to SQL.

- LINQ to SQL walkthroughs typically require the Northwind sample database. For more information, see Downloading Sample Databases.
- The dialog boxes and menu commands you see in walkthroughs might differ from those described in Help, depending on your active settings or Visual Studio edition. To change your settings, click **Import and Export Settings** on the **Tools** menu. For more information, see Customizing Development Settings in Visual Studio.
- For walkthroughs that address multi-tier scenarios, a server must be located on a computer that is distinct from the development computer, and you must have appropriate permissions to access the server.
- The name of the class that typically represents the Orders table in the Northwind sample database is `[Order]`. The escaping is required because `Order` is a keyword in Visual Basic.

Troubleshooting

Run-time errors can occur because you do not have sufficient permissions to access the databases used in these walkthroughs. See the following steps to help resolve the most common of these issues.

Log-On Issues

Your application might be trying to access the database by way of a database logon it does not accept.

To verify or change the database log on

1. On the Windows **Start** menu, point to **All Programs**, **Microsoft SQL Server 2005**, point to **Configuration Tools**, and then click **SQL Server Configuration Manager**.
2. In the left pane of the **SQL Server Configuration Manager**, click **SQL Server 2005 Services**.
3. In the right pane, right-click **SQL Server (SQLEXPRESS)**, and then click **Properties**.
4. Click the **Log On** tab and verify how you are trying to log on to the server.

   In most cases, **Local System** works.

   If you make a change, click **Restart** to restart the service.

Protocols

At times, protocols might not be set correctly for your application to access the database. For example, the **Named Pipes** protocol, which is required for walkthroughs in LINQ to SQL, is not enabled by default.

To enable the Named Pipes protocol

1. In the left pane of the **SQL Server Configuration Manager**, expand **SQL Server 2005 Network Configuration**, and then click **Protocols for SQLEXPRESS**.
2. In the right pane, make sure that the **Named Pipes** protocol is enabled. If it is not, right-click **Name Pipes** and then click **Enable**.

   You will have to stop and restart the service. Follow the steps in the next block.

Stopping and Restarting the Service

You must stop and restart services before your changes can take effect.

To stop and restart the service

1. In the left pane of the **SQL Server Configuration Manager**, click **SQL Server 2005 Services**.
2. In the right pane, right-click **SQL Server (SQLEXPRESS)**, and then click **Stop**.
3. Right-click **SQL Server (SQLEXPRESS)**, and then click **Restart**.

See Also

Getting Started

Walkthrough: Simple Object Model and Query (Visual Basic)

5/1/2017 • 6 min to read • Edit Online

This walkthrough provides a fundamental end-to-end LINQ to SQL scenario with minimal complexities. You will create an entity class that models the Customers table in the sample Northwind database. You will then create a simple query to list customers who are located in London.

This walkthrough is code-oriented by design to help show LINQ to SQL concepts. Normally speaking, you would use the Object Relational Designer to create your object model.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual Basic Development Settings.

Prerequisites

- This walkthrough uses a dedicated folder ("c:\linqtest") to hold files. Create this folder before you begin the walkthrough.
- This walkthrough requires the Northwind sample database. If you do not have this database on your development computer, you can download it from the Microsoft download site. For instructions, see Downloading Sample Databases. After you have downloaded the database, copy the file to the c:\linqtest folder.

Overview

This walkthrough consists of six main tasks:

- Creating a LINQ to SQL solution in Visual Studio.
- Mapping a class to a database table.
- Designating properties on the class to represent database columns.
- Specifying the connection to the Northwind database.
- Creating a simple query to run against the database.
- Executing the query and observing the results.

Creating a LINQ to SQL Solution

In this first task, you create a Visual Studio solution that contains the necessary references to build and run a LINQ to SQL project.

To create a LINQ to SQL solution

1. On the **File** menu, click **New Project**.

2. In the **Project types** pane of the **New Project** dialog box, click **Visual Basic**.

3. In the **Templates** pane, click **Console Application**.

4. In the **Name** box, type **LinqConsoleApp**.

5. Click **OK**.

Adding LINQ References and Directives

This walkthrough uses assemblies that might not be installed by default in your project. If `System.Data.Linq` is not listed as a reference in your project (click **Show All Files** in **Solution Explorer** and expand the **References** node), add it, as explained in the following steps.

To add System.Data.Linq

1. In **Solution Explorer**, right-click **References**, and then click **Add Reference**.

2. In the **Add Reference** dialog box, click **.NET**, click the System.Data.Linq assembly, and then click **OK**.

   The assembly is added to the project.

3. Also in the **Add Reference** dialog box, click **.NET**, scroll to and click System.Windows.Forms, and then click **OK**.

   This assembly, which supports the message box in the walkthrough, is added to the project.

4. Add the following directives above `Module1`:

```
Imports System.Data.Linq
Imports System.Data.Linq.Mapping
Imports System.Windows.Forms
```

Mapping a Class to a Database Table

In this step, you create a class and map it to a database table. Such a class is termed an *entity class*. Note that the mapping is accomplished by just adding the TableAttribute attribute. The Name property specifies the name of the table in the database.

To create an entity class and map it to a database table

- Type or paste the following code into Module1.vb immediately above `Sub Main`:

```
<Table(Name:="Customers")> _
Public Class Customer
End Class
```

Designating Properties on the Class to Represent Database Columns

In this step, you accomplish several tasks.

- You use the ColumnAttribute attribute to designate `CustomerID` and `City` properties on the entity class as representing columns in the database table.

- You designate the `CustomerID` property as representing a primary key column in the database.

- You designate `_CustomerID` and `_City` fields for private storage. LINQ to SQL can then store and retrieve values directly, instead of using public accessors that might include business logic.

To represent characteristics of two database columns

- Type or paste the following code into Module1.vb just before `End Class`:

  ```
  Private _CustomerID As String
  <Column(IsPrimaryKey:=True, Storage:="_CustomerID")> _
  Public Property CustomerID() As String
      Get
          Return Me._CustomerID
      End Get
      Set(ByVal value As String)
          Me._CustomerID = value
      End Set
  End Property

  Private _City As String
  <Column(Storage:="_City")> _
  Public Property City() As String
      Get
          Return Me._City
      End Get
      Set(ByVal value As String)
          Me._City = value
      End Set
  End Property
  ```

Specifying the Connection to the Northwind Database

In this step you use a DataContext object to establish a connection between your code-based data structures and the database itself. The DataContext is the main channel through which you retrieve objects from the database and submit changes.

You also declare a `Table(Of Customer)` to act as the logical, typed table for your queries against the Customers table in the database. You will create and execute these queries in later steps.

To specify the database connection

- Type or paste the following code into the `Sub Main` method.

  Note that the `northwnd.mdf` file is assumed to be in the linqtest folder. For more information, see the Prerequisites section earlier in this walkthrough.

  ```
  ' Use a connection string.
  Dim db As New DataContext _
      ("c:\linqtest\northwnd.mdf")

  ' Get a typed table to run queries.
  Dim Customers As Table(Of Customer) = _
      db.GetTable(Of Customer)()
  ```

Creating a Simple Query

In this step, you create a query to find which customers in the database Customers table are located in London. The query code in this step just describes the query. It does not execute it. This approach is known as *deferred execution*. For more information, see Introduction to LINQ Queries (C#).

You will also produce a log output to show the SQL commands that LINQ to SQL generates. This logging feature (which uses Log) is helpful in debugging, and in determining that the commands being sent to the database accurately represent your query.

To create a simple query

- Type or paste the following code into the `Sub Main` method after the `Table(Of Customer)` declaration:

```
' Attach the log to show generated SQL in a console window.
db.Log = Console.Out

' Query for customers in London.
Dim custQuery = _
    From cust In Customers _
    Where cust.City = "London" _
    Select cust
```

Executing the Query

In this step, you actually execute the query. The query expressions you created in the previous steps are not evaluated until the results are needed. When you begin the `For Each` iteration, a SQL command is executed against the database and objects are materialized.

To execute the query

1. Type or paste the following code at the end of the `Sub Main` method (after the query description):

```
' Format the message box.
Dim msg As String = "", title As String = "London customers:", _
    response As MsgBoxResult, style As MsgBoxStyle = _
    MsgBoxStyle.Information

' Execute the query.
For Each custObj In custQuery
    msg &= String.Format(custObj.CustomerID & vbCrLf)
Next

' Display the results.
response = MsgBox(msg, style, title)
```

2. Press F5 to debug the application.

> **NOTE**
> If your application generates a run-time error, see the Troubleshooting section of Learning by Walkthroughs.

   The message box displays a list of six customers. The Console window displays the generated SQL code.

3. Click **OK** to dismiss the message box.

   The application closes.

4. On the **File** menu, click **Save All**.

   You will need this application if you continue with the next walkthrough.

Next Steps

The Walkthrough: Querying Across Relationships (Visual Basic) topic continues where this walkthrough ends. The Querying Across Relationships walkthrough demonstrates how LINQ to SQL can query across tables, similar to *joins* in a relational database.

If you want to do the Querying Across Relationships walkthrough, make sure to save the solution for the walkthrough you have just completed, which is a prerequisite.

See Also

Walkthrough: Querying Across Relationships (Visual Basic)

5/1/2017 • 4 min to read • Edit Online

This walkthrough demonstrates the use of LINQ to SQL *associations* to represent foreign-key relationships in the database.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual Basic Development Settings.

Prerequisites

You must have completed Walkthrough: Simple Object Model and Query (Visual Basic). This walkthrough builds on that one, including the presence of the northwnd.mdf file in c:\linqtest.

Overview

This walkthrough consists of three main tasks:

- Adding an entity class to represent the Orders table in the sample Northwind database.
- Supplementing annotations to the `Customer` class to enhance the relationship between the `Customer` and `Order` classes.
- Creating and running a query to test the process of obtaining `Order` information by using the `Customer` class.

Mapping Relationships across Tables

After the `Customer` class definition, create the `Order` entity class definition that includes the following code, which indicates that `Orders.Customer` relates as a foreign key to `Customers.CustomerID`.

To add the Order entity class

- Type or paste the following code after the `Customer` class:

```
<Table(Name:="Orders")> _
Public Class Order
    Private _OrderID As Integer
    Private _CustomerID As String
    Private _Customers As EntityRef(Of Customer)

    Public Sub New()
        Me._Customers = New EntityRef(Of Customer)()
    End Sub

    <Column(Storage:="_OrderID", DbType:="Int NOT NULL IDENTITY", _
        IsPrimaryKey:=True, IsDBGenerated:=True)> _
    Public ReadOnly Property OrderID() As Integer
        Get
            Return Me._OrderID
        End Get
    End Property
    ' No need to specify a setter because IsDBGenerated is true.

    <Column(Storage:="_CustomerID", DbType:="NChar(5)")> _
    Public Property CustomerID() As String
        Get
            Return Me._CustomerID
        End Get
        Set(ByVal value As String)
            Me._CustomerID = value
        End Set
    End Property

    <Association(Storage:="_Customers", ThisKey:="CustomerID")> _
    Public Property Customers() As Customer
        Get
            Return Me._Customers.Entity
        End Get
        Set(ByVal value As Customer)
            Me._Customers.Entity = value
        End Set
    End Property
End Class
```

Annotating the Customer Class

In this step, you annotate the `Customer` class to indicate its relationship to the `Order` class. (This addition is not strictly necessary, because defining the relationship in either direction is sufficient to create the link. But adding this annotation does enable you to easily navigate objects in either direction.)

To annotate the Customer class

- Type or paste the following code into the `Customer` class:

```
Private _Orders As EntitySet(Of Order)

Public Sub New()
    Me._Orders = New EntitySet(Of Order)()
End Sub

<Association(Storage:="_Orders", OtherKey:="CustomerID")> _
Public Property Orders() As EntitySet(Of Order)
    Get
        Return Me._Orders
    End Get
    Set(ByVal value As EntitySet(Of Order))
        Me._Orders.Assign(value)
    End Set
End Property
```

Creating and Running a Query across the Customer-Order Relationship

You can now access `Order` objects directly from the `Customer` objects, or in the opposite order. You do not need an explicit *join* between customers and orders.

To access Order objects by using Customer objects

1. Modify the `Sub Main` method by typing or pasting the following code into the method:

```
' Query for customers who have no orders.
Dim custQuery = _
    From cust In Customers _
    Where Not cust.Orders.Any() _
    Select cust

Dim msg As String = "", title As String = _
    "Customers With No Orders", response As MsgBoxResult, _
    style As MsgBoxStyle = MsgBoxStyle.Information

For Each custObj In custQuery
    msg &= String.Format(custObj.CustomerID & vbCrLf)
Next
response = MsgBox(msg, style, title)
```

2. Press F5 to debug your application.

   Two names appear in the message box, and the Console window shows the generated SQL code.

3. Close the message box to stop debugging.

Creating a Strongly Typed View of Your Database

It is much easier to start with a strongly typed view of your database. By strongly typing the DataContext object, you do not need calls to GetTable. You can use strongly typed tables in all your queries when you use the strongly typed DataContext object.

In the following steps, you will create `Customers` as a strongly typed table that maps to the Customers table in the database.

To strongly type the DataContext object

1. Add the following code above the `Customer` class declaration.

```
Public Class Northwind
    Inherits DataContext
    ' Table(Of T) abstracts database details  per
    ' table/data type.
    Public Customers As Table(Of Customer)
    Public Orders As Table(Of Order)

    Public Sub New(ByVal connection As String)
        MyBase.New(connection)
    End Sub
End Class
```

2. Modify `Sub Main` to use the strongly typed DataContext as follows:

```
' Use a connection string.
Dim db As New Northwind _
    ("C:\linqtest\northwnd.mdf")

' Query for customers from Seattle.
Dim custs = _
    From cust In db.Customers _
    Where cust.City = "Seattle" _
    Select cust

For Each custObj In custs
    Console.WriteLine("ID=" & custObj.CustomerID)
Next

' Freeze the console window.
Console.ReadLine()
```

3. Press F5 to debug your application.

   The Console window output is:

   ```
   ID=WHITC
   ```

4. Press Enter in the Console window to close the application.

5. On the **File** menu, click **Save All** if you want to save this application.

Next Steps

The next walkthrough (Walkthrough: Manipulating Data (Visual Basic)) demonstrates how to manipulate data. That walkthrough does not require that you save the two walkthroughs in this series that you have already completed.

See Also

Learning by Walkthroughs

Walkthrough: Manipulating Data (Visual Basic)

5/1/2017 • 6 min to read • Edit Online

This walkthrough provides a fundamental end-to-end LINQ to SQL scenario for adding, modifying, and deleting data in a database. You will use a copy of the sample Northwind database to add a customer, change the name of a customer, and delete an order.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual Basic Development Settings.

Prerequisites

This walkthrough requires the following:

- This walkthrough uses a dedicated folder ("c:\linqtest2") to hold files. Create this folder before you begin the walkthrough.
- The Northwind sample database.

  If you do not have this database on your development computer, you can download it from the Microsoft download site. For instructions, see Downloading Sample Databases. After you have downloaded the database, copy the northwnd.mdf file to the c:\linqtest2 folder.
- A Visual Basic code file generated from the Northwind database.

  You can generate this file by using either the Object Relational Designer or the SQLMetal tool. This walkthrough was written by using the SQLMetal tool with the following command line:

  sqlmetal /code:"c:\linqtest2\northwind.vb" /language:vb "C:\linqtest2\northwnd.mdf" /pluralize

  For more information, see SqlMetal.exe (Code Generation Tool).

Overview

This walkthrough consists of six main tasks:

- Creating the LINQ to SQL solution in Visual Studio.
- Adding the database code file to the project.
- Creating a new customer object.
- Modifying the contact name of a customer.
- Deleting an order.
- Submitting these changes to the Northwind database.

Creating a LINQ to SQL Solution

In this first task, you create a Visual Studio solution that contains the necessary references to build and run a LINQ to SQL project.

To create a LINQ to SQL solution

1. On the Visual Studio **File** menu, click **New Project**.
2. In the **Project types** pane in the **New Project** dialog box, click **Visual Basic**.
3. In the **Templates** pane, click **Console Application**.
4. In the **Name** box, type **LinqDataManipulationApp**.
5. Click **OK**.

Adding LINQ References and Directives

This walkthrough uses assemblies that might not be installed by default in your project. If `System.Data.Linq` is not listed as a reference in your project (click **Show All Files** in **Solution Explorer** and expand the **References** node), add it, as explained in the following steps.

To add System.Data.Linq

1. In **Solution Explorer**, right-click **References**, and then click **Add Reference**.
2. In the **Add Reference** dialog box, click **.NET**, click the System.Data.Linq assembly, and then click **OK**.

   The assembly is added to the project.

3. In the code editor, add the following directives above **Module1**:

```
Imports System.Data.Linq
Imports System.Data.Linq.Mapping
```

Adding the Northwind Code File to the Project

These steps assume that you have used the SQLMetal tool to generate a code file from the Northwind sample database. For more information, see the Prerequisites section earlier in this walkthrough.

To add the northwind code file to the project

1. On the **Project** menu, click **Add Existing Item**.
2. In the **Add Existing Item** dialog box, navigate to c:\linqtest2\northwind.vb, and then click **Add**.

   The northwind.vb file is added to the project.

Setting Up the Database Connection

First, test your connection to the database. Note especially that the name of the database, Northwnd, has no i character. If you generate errors in the next steps, review the northwind.vb file to determine how the Northwind partial class is spelled.

To set up and test the database connection

1. Type or paste the following code into `Sub Main`:

```
' Use a connection string, but connect to
'     the temporary copy of the database.
Dim db As New Northwnd _
    ("C:\linqtest2\northwnd.mdf")

' Keep the console window open after activity stops.
Console.ReadLine()
```

2. Press F5 to test the application at this point.

   A **Console** window opens.

   Close the application by pressing Enter in the **Console** window, or by clicking **Stop Debugging** on the Visual Studio **Debug** menu.

Creating a New Entity

Creating a new entity is straightforward. You can create objects (such as `Customer`) by using the `New` keyword.

In this and the following sections, you are making changes only to the local cache. No changes are sent to the database until you call SubmitChanges toward the end of this walkthrough.

To add a new Customer entity object

1. Create a new `Customer` by adding the following code before `Console.ReadLine` in `Sub Main`:

```
' Create the new Customer object.
Dim newCust As New Customer()
newCust.CompanyName = "AdventureWorks Cafe"
newCust.CustomerID = "A3VCA"

' Add the customer to the Customers table.
db.Customers.InsertOnSubmit(newCust)

Console.WriteLine("Customers matching CA before insert:")

Dim custQuery = _
    From cust In db.Customers _
    Where cust.CustomerID.Contains("CA") _
    Select cust

For Each cust In custQuery
    Console.WriteLine("Customer ID: " & cust.CustomerID)
Next
```

2. Press F5 to debug the solution.

   The results that are shown in the console window are as follows:

   `Customers matching CA before insert:`

   `Customer ID: CACTU`

   `Customer ID: RICAR`

   Note that the new row does not appear in the results. The new data has not yet been submitted to the database.

3. Press Enter in the **Console** window to stop debugging.

Updating an Entity

In the following steps, you will retrieve a `Customer` object and modify one of its properties.

To change the name of a Customer

- Add the following code above `Console.ReadLine()`:

```
Dim existingCust = _
    (From cust In db.Customers _
    Where cust.CustomerID = "ALFKI" _
    Select cust).First()

' Change the contact name of the customer.
existingCust.ContactName = "New Contact"
```

Deleting an Entity

Using the same customer object, you can delete the first order.

The following code demonstrates how to sever relationships between rows, and how to delete a row from the database.

To delete a row

- Add the following code just above `Console.ReadLine()`:

```
' Access the first element in the Orders collection.
Dim ord0 As Order = existingCust.Orders(0)

' Access the first element in the OrderDetails collection.
Dim detail0 As OrderDetail = ord0.OrderDetails(0)

' Display the order to be deleted.
Console.WriteLine _
    (vbCrLf & "The Order Detail to be deleted is: OrderID = " _
    & detail0.OrderID)

' Mark the Order Detail row for deletion from the database.
db.OrderDetails.DeleteOnSubmit(detail0)
```

Submitting Changes to the Database

The final step required for creating, updating, and deleting objects is to actually submit the changes to the database. Without this step, your changes are only local and will not appear in query results.

To submit changes to the database

1. Insert the following code just above `Console.ReadLine`:

```
db.SubmitChanges()
```

2. Insert the following code (after `SubmitChanges`) to show the before and after effects of submitting the changes:

```
Console.WriteLine(vbCrLf & "Customers matching CA after update:")
Dim finalQuery = _
    From cust In db.Customers _
    Where cust.CustomerID.Contains("CA") _
    Select cust

For Each cust In finalQuery
    Console.WriteLine("Customer ID: " & cust.CustomerID)
Next
```

3. Press F5 to debug the solution.

   The console window appears as follows:

```
Customers matching CA before update:
Customer ID: CACTU
Customer ID: RICAR

The Order Detail to be deleted is: OrderID = 10643

Customers matching CA after update:
Customer ID: A3VCA
Customer ID: CACTU
Customer ID: RICAR
```

4. Press Enter in the **Console** window to stop debugging.

> **NOTE**
>
> After you have added the new customer by submitting the changes, you cannot execute this solution again as is, because you cannot add the same customer again as is. To execute the solution again, change the value of the customer ID to be added.

See Also

Learning by Walkthroughs

Walkthrough: Using Only Stored Procedures (Visual Basic)

5/10/2017 • 6 min to read • Edit Online

This walkthrough provides a basic end-to-end LINQ to SQL scenario for accessing data by using stored procedures only. This approach is often used by database administrators to limit how the datastore is accessed.

NOTE

You can also use stored procedures in LINQ to SQL applications to override default behavior, especially for `Create`, `Update`, and `Delete` processes. For more information, see Customizing Insert, Update, and Delete Operations.

For purposes of this walkthrough, you will use two methods that have been mapped to stored procedures in the Northwind sample database: CustOrdersDetail and CustOrderHist. The mapping occurs when you run the SqlMetal command-line tool to generate a Visual Basic file. For more information, see the Prerequisites section later in this walkthrough.

This walkthrough does not rely on the Object Relational Designer. Developers using Visual Studio can also use the O/R Designer to implement stored procedure functionality. See LINQ to SQL Tools in Visual Studio.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual Basic Development Settings.

Prerequisites

This walkthrough requires the following:

- This walkthrough uses a dedicated folder ("c:\linqtest3") to hold files. Create this folder before you begin the walkthrough.
- The Northwind sample database.

  If you do not have this database on your development computer, you can download it from the Microsoft download site. For instructions, see Downloading Sample Databases. After you have downloaded the database, copy the northwnd.mdf file to the c:\linqtest3 folder.
- A Visual Basic code file generated from the Northwind database.

  This walkthrough was written by using the SqlMetal tool with the following command line:

  sqlmetal /code:"c:\linqtest3\northwind.vb" /language:vb "c:\linqtest3\northwnd.mdf" /sprocs /functions /pluralize

  For more information, see SqlMetal.exe (Code Generation Tool).

Overview

This walkthrough consists of six main tasks:

- Setting up the LINQ to SQL solution in Visual Studio.
- Adding the System.Data.Linq assembly to the project.
- Adding the database code file to the project.
- Creating a connection to the database.
- Setting up the user interface.
- Running and testing the application.

Creating a LINQ to SQL Solution

In this first task, you create a Visual Studio solution that contains the necessary references to build and run a LINQ to SQL project.

To create a LINQ to SQL solution

1. On the Visual Studio **File** menu, click **New Project**.
2. In the **Project types** pane in the **New Project** dialog box, expand **Visual Basic**, and then click **Windows**.
3. In the **Templates** pane, click **Windows Forms Application**.
4. In the **Name** box, type **SprocOnlyApp**.
5. Click **OK**.

   The Windows Forms Designer opens.

Adding the LINQ to SQL Assembly Reference

The LINQ to SQL assembly is not included in the standard Windows Forms Application template. You will have to add the assembly yourself, as explained in the following steps:

To add System.Data.Linq.dll

1. In **Solution Explorer**, click **Show All Files**.
2. In **Solution Explorer**, right-click **References**, and then click **Add Reference**.
3. In the **Add Reference** dialog box, click **.NET**, click the System.Data.Linq assembly, and then click **OK**.

   The assembly is added to the project.

Adding the Northwind Code File to the Project

This step assumes that you have used the SqlMetal tool to generate a code file from the Northwind sample database. For more information, see the Prerequisites section earlier in this walkthrough.

To add the northwind code file to the project

1. On the **Project** menu, click **Add Existing Item**.
2. In the **Add Existing Item** dialog box, move to c:\linqtest3\northwind.vb, and then click **Add**.

   The northwind.vb file is added to the project.

Creating a Database Connection

In this step, you define the connection to the Northwind sample database. This walkthrough uses "c:\linqtest3\northwnd.mdf" as the path.

To create the database connection

1. In **Solution Explorer**, right-click **Form1.vb**, and then click **View Code**.

   `Class Form1` appears in the code editor.

2. Type the following code into the `Form1` code block:

   ```
   Dim db As New Northwnd("c:\linqtest3\northwnd.mdf")
   ```

Setting up the User Interface

In this task you create an interface so that users can execute stored procedures to access data in the database. In the application that you are developing with this walkthrough, users can access data in the database only by using the stored procedures embedded in the application.

To set up the user interface

1. Return to the Windows Forms Designer (**Form1.vb[Design]**).

2. On the **View** menu, click **Toolbox**.

   The toolbox opens.

   > **NOTE**
   >
   > Click the **AutoHide** pushpin to keep the toolbox open while you perform the remaining steps in this section.

3. Drag two buttons, two text boxes, and two labels from the toolbox onto **Form1**.

   Arrange the controls as in the accompanying illustration. Expand **Form1** so that the controls fit easily.

4. Right-click **Label1**, and then click **Properties**.

5. Change the **Text** property from **Label1** to **Enter OrderID:**.

6. In the same way for **Label2**, change the **Text** property from **Label2** to **Enter CustomerID:**.

7. In the same way, change the **Text** property for **Button1** to **Order Details**.

8. Change the **Text** property for **Button2** to **Order History**.

   Widen the button controls so that all the text is visible.

To handle button clicks

1. Double-click **Order Details** on **Form1** to create the `Button1` event handler and open the code editor.

2. Type the following code into the `Button1` handler:

```
' Declare a variable to hold the contents of
' TextBox1 as an argument for the stored
' procedure.
Dim parm As String = TextBox1.Text

' Declare a variable to hold the results returned
' by the stored procedure.
Dim custQuery = db.CustOrdersDetail(parm)

' Clear the message box of previous results.
Dim msg As String = ""
Dim response As MsgBoxResult

' Execute the stored procedure and store the results.
For Each custOrdersDetail As CustOrdersDetailResult In custQuery
    msg &= custOrdersDetail.ProductName & vbCrLf
Next

' Display the results.
If msg = "" Then
    msg = "No results."
End If
response = MsgBox(msg)

' Clear the variables before continuing.
parm = ""
TextBox1.Text = ""
```

3. Now double-click **Button2** on Form1 to create the `Button2` event handler and open the code editor.

4. Type the following code into the `Button2` handler:

```
' Comments in the code for Button2 are the same
' as for Button1.
Dim parm As String = TextBox2.Text

Dim custQuery2 = db.CustOrderHist(parm)
Dim msg As String = ""
Dim response As MsgBoxResult

For Each custOrdHist As CustOrderHistResult In custQuery2
    msg &= custOrdHist.ProductName & vbCrLf
Next

If msg = "" Then
    msg = "No results."
End If

response = MsgBox(msg)
parm = ""
TextBox2.Text = ""
```

Testing the Application

Now it is time to test your application. Note that your contact with the datastore is limited to whatever actions the two stored procedures can take. Those actions are to return the products included for any orderID you enter, or to return a history of products ordered for any CustomerID you enter.

To test the application

1. Press F5 to start debugging.

   Form1 appears.

2. In the **Enter OrderID** box, type **10249** and then click **Order Details**.

   A message box lists the products included in order 10249.

   Click **OK** to close the message box.

3. In the **Enter CustomerID** box, type `ALFKI`, and then click **Order History**.

   A message box lists the order history for customer ALFKI.

   Click **OK** to close the message box.

4. In the **Enter OrderID** box, type `123`, and then click **Order Details**.

   A message box displays "No results."

   Click **OK** to close the message box.

5. On the **Debug** menu, click **Stop debugging**.

   The debug session closes.

6. If you have finished experimenting, you can click **Close Project** on the **File** menu, and save your project when you are prompted.

Next Steps

You can enhance this project by making some changes. For example, you could list available stored procedures in a list box and have the user select which procedures to execute. You could also stream the output of the reports to a text file.

See Also

Learning by Walkthroughs
Stored Procedures

Walkthrough: Simple Object Model and Query (C#)

5/1/2017 • 5 min to read • Edit Online

This walkthrough provides a fundamental end-to-end LINQ to SQL scenario with minimal complexities. You will create an entity class that models the Customers table in the sample Northwind database. You will then create a simple query to list customers who are located in London.

This walkthrough is code-oriented by design to help show LINQ to SQL concepts. Normally speaking, you would use the Object Relational Designer to create your object model.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual C# Development Settings.

Prerequisites

- This walkthrough uses a dedicated folder ("c:\linqtest5") to hold files. Create this folder before you begin the walkthrough.
- This walkthrough requires the Northwind sample database. If you do not have this database on your development computer, you can download it from the Microsoft download site. For instructions, see Downloading Sample Databases. After you have downloaded the database, copy the file to the c:\linqtest5 folder.

Overview

This walkthrough consists of six main tasks:

- Creating a LINQ to SQL solution in Visual Studio.
- Mapping a class to a database table.
- Designating properties on the class to represent database columns.
- Specifying the connection to the Northwind database.
- Creating a simple query to run against the database.
- Executing the query and observing the results.

Creating a LINQ to SQL Solution

In this first task, you create a Visual Studio solution that contains the necessary references to build and run a LINQ to SQL project.

To create a LINQ to SQL solution

1. On the Visual Studio **File** menu, point to **New**, and then click **Project**.
2. In the **Project types** pane of the **New Project** dialog box, click **Visual C#**.

3. In the **Templates** pane, click **Console Application**.

4. In the **Name** box, type **LinqConsoleApp**.

5. In the **Location** box, verify where you want to store your project files.

6. Click **OK**.

Adding LINQ References and Directives

This walkthrough uses assemblies that might not be installed by default in your project. If System.Data.Linq is not listed as a reference in your project (expand the **References** node in **Solution Explorer**), add it, as explained in the following steps.

To add System.Data.Linq

1. In **Solution Explorer**, right-click **References**, and then click **Add Reference**.

2. In the **Add Reference** dialog box, click **.NET**, click the System.Data.Linq assembly, and then click **OK**.

   The assembly is added to the project.

3. Add the following directives at the top of **Program.cs**:

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

Mapping a Class to a Database Table

In this step, you create a class and map it to a database table. Such a class is termed an *entity class*. Note that the mapping is accomplished by just adding the TableAttribute attribute. The Name property specifies the name of the table in the database.

To create an entity class and map it to a database table

- Type or paste the following code into Program.cs immediately above the `Program` class declaration:

```
[Table(Name = "Customers")]
public class Customer
{
}
```

Designating Properties on the Class to Represent Database Columns

In this step, you accomplish several tasks.

- You use the ColumnAttribute attribute to designate `CustomerID` and `City` properties on the entity class as representing columns in the database table.

- You designate the `CustomerID` property as representing a primary key column in the database.

- You designate `_CustomerID` and `_City` fields for private storage. LINQ to SQL can then store and retrieve values directly, instead of using public accessors that might include business logic.

To represent characteristics of two database columns

- Type or paste the following code into Program.cs inside the curly braces for the `Customer` class.

```
private string _CustomerID;
[Column(IsPrimaryKey=true, Storage="_CustomerID")]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        this._CustomerID = value;
    }

}

private string _City;
[Column(Storage="_City")]
public string City
{
    get
    {
        return this._City;
    }
    set
    {
        this._City=value;
    }
}
```

Specifying the Connection to the Northwind Database

In this step you use a DataContext object to establish a connection between your code-based data structures and the database itself. The DataContext is the main channel through which you retrieve objects from the database and submit changes.

You also declare a `Table<Customer>` to act as the logical, typed table for your queries against the Customers table in the database. You will create and execute these queries in later steps.

To specify the database connection

- Type or paste the following code into the `Main` method.

  Note that the `northwnd.mdf` file is assumed to be in the linqtest5 folder. For more information, see the Prerequisites section earlier in this walkthrough.

  ```
  // Use a connection string.
  DataContext db = new DataContext
      (@"c:\linqtest5\northwnd.mdf");

  // Get a typed table to run queries.
  Table<Customer> Customers = db.GetTable<Customer>();
  ```

Creating a Simple Query

In this step, you create a query to find which customers in the database Customers table are located in London. The query code in this step just describes the query. It does not execute it. This approach is known as *deferred execution*. For more information, see Introduction to LINQ Queries (C#).

You will also produce a log output to show the SQL commands that LINQ to SQL generates. This logging feature (which uses Log) is helpful in debugging, and in determining that the commands being sent to the database accurately represent your query.

To create a simple query

- Type or paste the following code into the `Main` method after the `Table<Customer>` declaration.

```
// Attach the log to show generated SQL.
db.Log = Console.Out;

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in Customers
    where cust.City == "London"
    select cust;
```

Executing the Query

In this step, you actually execute the query. The query expressions you created in the previous steps are not evaluated until the results are needed. When you begin the `foreach` iteration, a SQL command is executed against the database and objects are materialized.

To execute the query

1. Type or paste the following code at the end of the `Main` method (after the query description).

```
foreach (Customer cust in custQuery)
{
    Console.WriteLine("ID={0}, City={1}", cust.CustomerID,
        cust.City);
}

// Prevent console window from closing.
Console.ReadLine();
```

2. Press F5 to debug the application.

> **NOTE**
>
> If your application generates a run-time error, see the Troubleshooting section of Learning by Walkthroughs.

The query results in the console window should appear as follows:

```
ID=AROUT, City=London
```

```
ID=BSBEV, City=London
```

```
ID=CONSH, City=London
```

```
ID=EASTC, City=London
```

```
ID=NORTS, City=London
```

```
ID=SEVES, City=London
```

3. Press Enter in the console window to close the application.

Next Steps

The Walkthrough: Querying Across Relationships (C#) topic continues where this walkthrough ends. The Query Across Relationships walkthrough demonstrates how LINQ to SQL can query across tables, similar to *joins* in a relational database.

If you want to do the Query Across Relationships walkthrough, make sure to save the solution for the walkthrough you have just completed, which is a prerequisite.

See Also

Learning by Walkthroughs

Walkthrough: Querying Across Relationships (C#)

5/1/2017 • 3 min to read • Edit Online

This walkthrough demonstrates the use of LINQ to SQL *associations* to represent foreign-key relationships in the database.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual C# Development Settings.

Prerequisites

You must have completed Walkthrough: Simple Object Model and Query (C#). This walkthrough builds on that one, including the presence of the northwnd.mdf file in c:\linqtest5.

Overview

This walkthrough consists of three main tasks:

- Adding an entity class to represent the Orders table in the sample Northwind database.
- Supplementing annotations to the `Customer` class to enhance the relationship between the `Customer` and `Order` classes.
- Creating and running a query to test obtaining `Order` information by using the `Customer` class.

Mapping Relationships Across Tables

After the `Customer` class definition, create the `Order` entity class definition that includes the following code, which indicates that `Order.Customer` relates as a foreign key to `Customer.CustomerID`.

To add the Order entity class

- Type or paste the following code after the `Customer` class:

```
[Table(Name = "Orders")]
public class Order
{
    private int _OrderID = 0;
    private string _CustomerID;
    private EntityRef<Customer> _Customer;
    public Order() { this._Customer = new EntityRef<Customer>(); }

    [Column(Storage = "_OrderID", DbType = "Int NOT NULL IDENTITY",
    IsPrimaryKey = true, IsDbGenerated = true)]
    public int OrderID
    {
        get { return this._OrderID; }
        // No need to specify a setter because IsDBGenerated is
        // true.
    }

    [Column(Storage = "_CustomerID", DbType = "NChar(5)")]
    public string CustomerID
    {
        get { return this._CustomerID; }
        set { this._CustomerID = value; }
    }

    [Association(Storage = "_Customer", ThisKey = "CustomerID")]
    public Customer Customer
    {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }
}
```

Annotating the Customer Class

In this step, you annotate the `Customer` class to indicate its relationship to the `Order` class. (This addition is not strictly necessary, because defining the relationship in either direction is sufficient to create the link. But adding this annotation does enable you to easily navigate objects in either direction.)

To annotate the Customer class

- Type or paste the following code into the `Customer` class:

```
private EntitySet<Order> _Orders;
public Customer()
{
    this._Orders = new EntitySet<Order>();
}

[Association(Storage = "_Orders", OtherKey = "CustomerID")]
public EntitySet<Order> Orders
{
    get { return this._Orders; }
    set { this._Orders.Assign(value); }
}
```

Creating and Running a Query Across the Customer-Order Relationship

You can now access `Order` objects directly from the `Customer` objects, or in the opposite order. You do not need an explicit *join* between customers and orders.

To access Order objects by using Customer objects

1. Modify the `Main` method by typing or pasting the following code into the method:

```
// Query for customers who have placed orders.
var custQuery =
    from cust in Customers
    where cust.Orders.Any()
    select cust;

foreach (var custObj in custQuery)
{
    Console.WriteLine("ID={0}, Qty={1}", custObj.CustomerID,
        custObj.Orders.Count);
}
```

2. Press F5 to debug your application.

NOTE

You can eliminate the SQL code in the Console window by commenting out `db.Log = Console.Out;`.

3. Press Enter in the Console window to stop debugging.

Creating a Strongly Typed View of Your Database

It is much easier to start with a strongly typed view of your database. By strongly typing the DataContext object, you do not need calls to GetTable. You can use strongly typed tables in all your queries when you use the strongly typed DataContext object.

In the following steps, you will create `Customers` as a strongly typed table that maps to the Customers table in the database.

To strongly type the DataContext object

1. Add the following code above the `Customer` class declaration.

```
public class Northwind : DataContext
{
    // Table<T> abstracts database details per table/data type.
    public Table<Customer> Customers;
    public Table<Order> Orders;

    public Northwind(string connection) : base(connection) { }
}
```

2. Modify the `Main` method to use the strongly typed DataContext as follows:

```
// Use a connection string.
Northwind db = new Northwind(@"C:\linqtest5\northwnd.mdf");

// Query for customers from Seattle.
var custQuery =
    from cust in db.Customers
    where cust.City == "Seattle"
    select cust;

foreach (var custObj in custQuery)
{
    Console.WriteLine("ID={0}", custObj.CustomerID);
}
// Freeze the console window.
Console.ReadLine();
```

3. Press F5 to debug your application.

   The Console window output is:

   ```
   ID=WHITC
   ```

4. Press Enter in the console window to stop debugging.

Next Steps

The next walkthrough (Walkthrough: Manipulating Data (C#)) demonstrates how to manipulate data. That walkthrough does not require that you save the two walkthroughs in this series that you have already completed.

See Also

Learning by Walkthroughs

Walkthrough: Manipulating Data (C#)

5/1/2017 • 5 min to read • Edit Online

This walkthrough provides a fundamental end-to-end LINQ to SQL scenario for adding, modifying, and deleting data in a database. You will use a copy of the sample Northwind database to add a customer, change the name of a customer, and delete an order.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual C# Development Settings.

Prerequisites

This walkthrough requires the following:

- This walkthrough uses a dedicated folder ("c:\linqtest6") to hold files. Create this folder before you begin the walkthrough.

- The Northwind sample database.

  If you do not have this database on your development computer, you can download it from the Microsoft download site. For instructions, see Downloading Sample Databases. After you have downloaded the database, copy the northwnd.mdf file to the c:\linqtest6 folder.

- A C# code file generated from the Northwind database.

  You can generate this file by using either the Object Relational Designer or the SQLMetal tool. This walkthrough was written by using the SQLMetal tool with the following command line:

  sqlmetal /code:"c:\linqtest6\northwind.cs" /language:csharp "C:\linqtest6\northwnd.mdf" /pluralize

  For more information, see SqlMetal.exe (Code Generation Tool).

Overview

This walkthrough consists of six main tasks:

- Creating the LINQ to SQL solution in Visual Studio.

- Adding the database code file to the project.

- Creating a new customer object.

- Modifying the contact name of a customer.

- Deleting an order.

- Submitting these changes to the Northwind database.

Creating a LINQ to SQL Solution

In this first task, you create a Visual Studio solution that contains the necessary references to build and run a LINQ to SQL project.

To create a LINQ to SQL solution

1. On the Visual Studio **File** menu, point to **New**, and then click **Project**.
2. In the **Project types** pane in the **New Project** dialog box, click **Visual C#**.
3. In the **Templates** pane, click **Console Application**.
4. In the **Name** box, type **LinqDataManipulationApp**.
5. In the **Location** box, verify where you want to store your project files.
6. Click **OK**.

Adding LINQ References and Directives

This walkthrough uses assemblies that might not be installed by default in your project. If System.Data.Linq is not listed as a reference in your project, add it, as explained in the following steps:

To add System.Data.Linq

1. In **Solution Explorer**, right-click **References**, and then click **Add Reference**.
2. In the **Add Reference** dialog box, click **.NET**, click the System.Data.Linq assembly, and then click **OK**.

   The assembly is added to the project.
3. Add the following directives at the top of Program.cs:

   ```
   using System.Data.Linq;
   using System.Data.Linq.Mapping;
   ```

Adding the Northwind Code File to the Project

These steps assume that you have used the SQLMetal tool to generate a code file from the Northwind sample database. For more information, see the Prerequisites section earlier in this walkthrough.

To add the northwind code file to the project

1. On the **Project** menu, click **Add Existing Item**.
2. In the **Add Existing Item** dialog box, navigate to c:\linqtest6\northwind.cs, and then click **Add**.

   The northwind.cs file is added to the project.

Setting Up the Database Connection

First, test your connection to the database. Note especially that the database, Northwnd, has no i character. If you generate errors in the next steps, review the northwind.cs file to determine how the Northwind partial class is spelled.

To set up and test the database connection

1. Type or paste the following code into the `Main` method in the Program class:

```
// Use the following connection string.
Northwnd db = new Northwnd(@"c:\linqtest6\northwnd.mdf");

// Keep the console window open after activity stops.
Console.ReadLine();
```

2. Press F5 to test the application at this point.

   A **Console** window opens.

   You can close the application by pressing Enter in the **Console** window, or by clicking **Stop Debugging** on the Visual Studio **Debug** menu.

Creating a New Entity

Creating a new entity is straightforward. You can create objects (such as `Customer`) by using the `new` keyword.

In this and the following sections, you are making changes only to the local cache. No changes are sent to the database until you call SubmitChanges toward the end of this walkthrough.

To add a new Customer entity object

1. Create a new `Customer` by adding the following code before `Console.ReadLine();` in the `Main` method:

```
// Create the new Customer object.
Customer newCust = new Customer();
newCust.CompanyName = "AdventureWorks Cafe";
newCust.CustomerID = "ADVCA";

// Add the customer to the Customers table.
db.Customers.InsertOnSubmit(newCust);

Console.WriteLine("\nCustomers matching CA before insert");

foreach (var c in db.Customers.Where(cust => cust.CustomerID.Contains("CA")))
{
    Console.WriteLine("{0}, {1}, {2}",
        c.CustomerID, c.CompanyName, c.Orders.Count);
}
```

2. Press F5 to debug the solution.

3. Press Enter in the **Console** window to stop debugging and continue the walkthrough.

Updating an Entity

In the following steps, you will retrieve a `Customer` object and modify one of its properties.

To change the name of a Customer

- Add the following code above `Console.ReadLine();`:

```
// Query for specific customer.
// First() returns one object rather than a collection.
var existingCust =
    (from c in db.Customers
     where c.CustomerID == "ALFKI"
     select c)
    .First();

// Change the contact name of the customer.
existingCust.ContactName = "New Contact";
```

Deleting an Entity

Using the same customer object, you can delete the first order.

The following code demonstrates how to sever relationships between rows, and how to delete a row from the database. Add the following code before `Console.ReadLine` to see how objects can be deleted:

To delete a row

- Add the following code just above `Console.ReadLine();`:

```
// Access the first element in the Orders collection.
Order ord0 = existingCust.Orders[0];

// Access the first element in the OrderDetails collection.
OrderDetail detail0 = ord0.OrderDetails[0];

// Display the order to be deleted.
Console.WriteLine
    ("The Order Detail to be deleted is: OrderID = {0}, ProductID = {1}",
    detail0.OrderID, detail0.ProductID);

// Mark the Order Detail row for deletion from the database.
db.OrderDetails.DeleteOnSubmit(detail0);
```

Submitting Changes to the Database

The final step required for creating, updating, and deleting objects, is to actually submit the changes to the database. Without this step, your changes are only local and will not appear in query results.

To submit changes to the database

1. Insert the following code just above `Console.ReadLine`:

```
db.SubmitChanges();
```

2. Insert the following code (after `SubmitChanges`) to show the before and after effects of submitting the changes:

```
Console.WriteLine("\nCustomers matching CA after update");
foreach (var c in db.Customers.Where(cust =>
    cust.CustomerID.Contains("CA")))
{
    Console.WriteLine("{0}, {1}, {2}",
        c.CustomerID, c.CompanyName, c.Orders.Count);
}
```

3. Press F5 to debug the solution.

4. Press Enter in the **Console** window to close the application.

> **NOTE**
>
> After you have added the new customer by submitting the changes, you cannot execute this solution again as is. To execute the solution again, change the name of the customer and customer ID to be added.

See Also

Learning by Walkthroughs

Walkthrough: Using Only Stored Procedures (C#)

5/10/2017 • 6 min to read • Edit Online

This walkthrough provides a basic end-to-end LINQ to SQL scenario for accessing data by executing stored procedures only. This approach is often used by database administrators to limit how the datastore is accessed.

> **NOTE**
> You can also use stored procedures in LINQ to SQL applications to override default behavior, especially for `Create`, `Update`, and `Delete` processes. For more information, see Customizing Insert, Update, and Delete Operations.

For purposes of this walkthrough, you will use two methods that have been mapped to stored procedures in the Northwind sample database: CustOrdersDetail and CustOrderHist. The mapping occurs when you run the SqlMetal command-line tool to generate a C# file. For more information, see the Prerequisites section later in this walkthrough.

This walkthrough does not rely on the Object Relational Designer. Developers using Visual Studio can also use the O/R Designer to implement stored procedure functionality. See LINQ to SQL Tools in Visual Studio.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

This walkthrough was written by using Visual C# Development Settings.

Prerequisites

This walkthrough requires the following:

- This walkthrough uses a dedicated folder ("c:\linqtest7") to hold files. Create this folder before you begin the walkthrough.
- The Northwind sample database.

  If you do not have this database on your development computer, you can download it from the Microsoft download site. For instructions, see Downloading Sample Databases. After you have downloaded the database, copy the northwnd.mdf file to the c:\linqtest7 folder.
- A C# code file generated from the Northwind database.

  This walkthrough was written by using the SqlMetal tool with the following command line:

  sqlmetal /code:"c:\linqtest7\northwind.cs" /language:csharp "c:\linqtest7\northwnd.mdf" /sprocs /functions /pluralize

  For more information, see SqlMetal.exe (Code Generation Tool).

Overview

This walkthrough consists of six main tasks:

- Setting up the LINQ to SQL solution in Visual Studio.
- Adding the System.Data.Linq assembly to the project.
- Adding the database code file to the project.
- Creating a connection with the database.
- Setting up the user interface.
- Running and testing the application.

Creating a LINQ to SQL Solution

In this first task, you create a Visual Studio solution that contains the necessary references to build and run a LINQ to SQL project.

To create a LINQ to SQL solution

1. On the Visual Studio **File** menu, point to **New**, and then click **Project**.
2. In the **Project types** pane in the **New Project** dialog box, click **Visual C#**.
3. In the **Templates** pane, click **Windows Forms Application**.
4. In the **Name** box, type **SprocOnlyApp**.
5. In the **Location** box, verify where you want to store your project files.
6. Click **OK**.

   The Windows Forms Designer opens.

Adding the LINQ to SQL Assembly Reference

The LINQ to SQL assembly is not included in the standard Windows Forms Application template. You will have to add the assembly yourself, as explained in the following steps:

To add System.Data.Linq.dll

1. In **Solution Explorer**, right-click **References**, and then click **Add Reference**.
2. In the **Add Reference** dialog box, click **.NET**, click the System.Data.Linq assembly, and then click **OK**.

   The assembly is added to the project.

Adding the Northwind Code File to the Project

This step assumes that you have used the SqlMetal tool to generate a code file from the Northwind sample database. For more information, see the Prerequisites section earlier in this walkthrough.

To add the northwind code file to the project

1. On the **Project** menu, click **Add Existing Item**.
2. In the **Add Existing Item** dialog box, move to c:\linqtest7\northwind.cs, and then click **Add**.

   The northwind.cs file is added to the project.

Creating a Database Connection

In this step, you define the connection to the Northwind sample database. This walkthrough uses "c:\linqtest7\northwnd.mdf" as the path.

To create the database connection

1. In **Solution Explorer**, right-click **Form1.cs**, and then click **View Code**.

2. Type the following code into the `Form1` class:

```
Northwnd db = new Northwnd(@"c:\linqtest7\northwnd.mdf");
```

Setting up the User Interface

In this task you set up an interface so that users can execute stored procedures to access data in the database. In the applications that you are developing with this walkthrough, users can access data in the database only by using the stored procedures embedded in the application.

To set up the user interface

1. Return to the Windows Forms Designer (**Form1.cs[Design]**).

2. On the **View** menu, click **Toolbox**.

   The toolbox opens.

   > **NOTE**
   >
   > Click the **AutoHide** pushpin to keep the toolbox open while you perform the remaining steps in this section.

3. Drag two buttons, two text boxes, and two labels from the toolbox onto **Form1**.

   Arrange the controls as in the accompanying illustration. Expand **Form1** so that the controls fit easily.

4. Right-click **label1**, and then click **Properties**.

5. Change the **Text** property from **label1** to **Enter OrderID:**.

6. In the same way for **label2**, change the **Text** property from **label2** to **Enter CustomerID:**.

7. In the same way, change the **Text** property for **button1** to **Order Details**.

8. Change the **Text** property for **button2** to **Order History**.

   Widen the button controls so that all the text is visible.

To handle button clicks

1. Double-click **Order Details** on **Form1** to open the button1 event handler in the code editor.

2. Type the following code into the `button1` handler:

```
// Declare a variable to hold the contents of
// textBox1 as an argument for the stored
// procedure.
string param = textBox1.Text;

// Declare a variable to hold the results
// returned by the stored procedure.
var custquery = db.CustOrdersDetail(Convert.ToInt32(param));

// Execute the stored procedure and display the results.
string msg = "";
foreach (CustOrdersDetailResult custOrdersDetail in custquery)
{
    msg = msg + custOrdersDetail.ProductName + "\n";
}
if (msg == "")
    msg = "No results.";
MessageBox.Show(msg);

// Clear the variables before continuing.
param = "";
textBox1.Text = "";
```

3. Now double-click **button2** on **Form1** to open the `button2` handler

4. Type the following code into the `button2` handler:

```
// Comments in the code for button2 are the same
// as for button1.
string param = textBox2.Text;

var custquery = db.CustOrderHist(param);

string msg = "";
foreach (CustOrderHistResult custOrdHist in custquery)
{
    msg = msg + custOrdHist.ProductName + "\n";
}
MessageBox.Show(msg);

param = "";
textBox2.Text = "";
```

Testing the Application

Now it is time to test your application. Note that your contact with the datastore is limited to whatever actions the two stored procedures can take. Those actions are to return the products included for any orderID you enter, or to return a history of products ordered for any CustomerID you enter.

To test the application

1. Press F5 to start debugging.

   Form1 appears.

2. In the **Enter OrderID** box, type `10249`, and then click **Order Details**.

   A message box lists the products included in order 10249.

   Click **OK** to close the message box.

3. In the **Enter CustomerID** box, type `ALFKI`, and then click **Order History**.

A message box appears that lists the order history for customer ALFKI.

Click **OK** to close the message box.

4. In the **Enter OrderID** box, type `123`, and then click **Order Details**.

   A message box appears that displays "No results."

   Click **OK** to close the message box.

5. On the **Debug** menu, click **Stop debugging**.

   The debug session closes.

6. If you have finished experimenting, you can click **Close Project** on the **File** menu, and save your project when you are prompted.

Next Steps

You can enhance this project by making some changes. For example, you could list available stored procedures in a list box and have the user select which procedures to execute. You could also stream the output of the reports to a text file.

See Also

Learning by Walkthroughs
Stored Procedures

Programming Guide

5/1/2017 • 1 min to read • Edit Online

This section contains information about how to create and use your LINQ to SQL object model. If you are using Visual Studio, you can also use the Object Relational Designer to perform many of these same tasks.

You can also search the MSDN Library for specific issues, and you can participate in the LINQ Forum, where you can discuss more complex topics in detail with experts. Finally, the LINQ to SQL: .NET Language-Integrated Query for Relational Data white paper details LINQ to SQL technology, complete with Visual Basic and C# code examples.

In This Section

Creating the Object Model
Describes how to generate an object model.

Communicating with the Database
Describes how to use a DataContext object as a conduit to the database.

Querying the Database
Describes how to execute queries in LINQ to SQL, and provides many examples.

Making and Submitting Data Changes
Describes how change data in the database.

Debugging Support
Describes the support available for debugging LINQ to SQL projects.

Background Information
Includes additional items, such as concurrency conflict resolution, creating new databases, and more, for more advanced users.

Related Sections

LINQ to SQL
Provides links to topics that explain the LINQ to SQL technology and demonstrate features.

Stored Procedures
Includes links to topics that illustrate how to use stored procedures.

Introduction to LINQ
Provides resources to help you begin to learn about LINQ to SQL.

Creating the Object Model

5/1/2017 • 1 min to read • Edit Online

You can create your object model from an existing database and use the model in its default state. You can also customize many aspects of the model and its behavior.

If you are using Visual Studio, you can use the Object Relational Designer to create your object model.

In This Section

How to: Generate the Object Model in Visual Basic or C#
Describes how to use the SQLMetal command-line tool. Also provides a link to the Object Relational Designer for Visual Studio users

How to: Generate the Object Model as an External File
Describes how to generate an external mapping file instead of using attribute-based mapping.

How to: Generate Customized Code by Modifying a DBML File
Describes how to generate Visual Basic or C# code from a DBML metadata file.

How to: Validate DBML and External Mapping Files
Describes how to validate mapping files that you have modified (advanced).

How to: Make Entities Serializable
Describes how to add appropriate attributes to make entities serializable.

How to: Customize Entity Classes by Using the Code Editor
Describes how to use the code editor to write your own mapping code, or customize code that has been autogenerated.

Related Sections

The LINQ to SQL Object Model
Provides details about the LINQ to SQL object model.

Typical Steps for Using LINQ to SQL
Explains the typical steps that you follow to implement a LINQ to SQL application.

How to: Generate the Object Model in Visual Basic or C#

5/1/2017 • 1 min to read • Edit Online

In LINQ to SQL, an object model in your own programming language is mapped to a relational database. Two tools are available for automatically generating a Visual Basic or C# model from the metadata of an existing database.

- If you are using Visual Studio, you can use the Object Relational Designer to generate an object model. The O/R Designer provides a rich user interface to help you generate a LINQ to SQL object model.
- The SQLMetal command-line tool. For more information, see SqlMetal.exe (Code Generation Tool).

> **NOTE**
> If you do not have an existing database and want to create one from an object model, you can create your object model by using your code editor and CreateDatabase. For more information, see How to: Dynamically Create a Database.

Documentation for the O/R Designer provides examples of how to generate a Visual Basic or C# object model by using the O/R Designer. The following information provide examples of how to use the SQLMetal command-line tool. For more information, see SqlMetal.exe (Code Generation Tool).

Example

The SQLMetal command line shown in the following example produces Visual Basic code as the attribute-based object model of the Northwind sample database. Stored procedures and functions are also rendered.

```
sqlmetal /code:northwind.vb /language:vb "c:\northwnd.mdf" /sprocs /functions
```

Example

The SQLMetal command line shown in the following example produces C# code as the attribute-based object model of the Northwind sample database. Stored procedures and functions are also rendered, and table names are automatically pluralized.

```
sqlmetal /code:northwind.cs /language:csharp "c:\northwnd.mdf" /sprocs /functions /pluralize
```

See Also

Programming Guide
The LINQ to SQL Object Model
Learning by Walkthroughs
How to: Customize Entity Classes by Using the Code Editor
Attribute-Based Mapping
SqlMetal.exe (Code Generation Tool)
External Mapping
Downloading Sample Databases

How to: Generate the Object Model as an External File

6/2/2017 • 1 min to read • Edit Online

As an alternative to attribute-based mapping, you can generate your object model as an external XML file by using the SQLMetal command-line tool. For more information, see SqlMetal.exe (Code Generation Tool). By using an external XML mapping file, you reduce clutter in your code. You can also change behavior by modifying the external file without recompiling the binaries of your application. For more information, see External Mapping.

NOTE

The Object Relational Designer does not support generation of an external mapping file.

Example

The following command generates an external mapping file from the Northwind sample database.

```
sqlmetal /server:myserver /database:northwind /map:externalfile.xml
```

Example

The following excerpt from an external mapping file shows the mapping for the Customers table in the Northwind sample database. This excerpt was generated by executing SQLMetal with the **/map** option.

```
<?xml version="1.0" encoding="utf-8"?>
<Database xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
Name="northwnd">
  <Table Name="Customers">
    <Type Name=".Customer">
      <Column Name="CustomerID" Member="CustomerID" Storage="_CustomerID" DbType="NChar(5) NOT NULL"
CanBeNull="False" IsPrimaryKey="True" />
      <Column Name="CompanyName" Member="CompanyName" Storage="_CompanyName" DbType="NVarChar(40) NOT NULL"
CanBeNull="False" />
      <Column Name="ContactName" Member="ContactName" Storage="_ContactName" DbType="NVarChar(30)" />
      <Column Name="ContactTitle" Member="ContactTitle" Storage="_ContactTitle" DbType="NVarChar(30)" />
      <Column Name="Address" Member="Address" Storage="_Address" DbType="NVarChar(60)" />
      <Column Name="City" Member="City" Storage="_City" DbType="NVarChar(15)" />
      <Column Name="Region" Member="Region" Storage="_Region" DbType="NVarChar(15)" />
      <Column Name="PostalCode" Member="PostalCode" Storage="_PostalCode" DbType="NVarChar(10)" />
      <Column Name="Country" Member="Country" Storage="_Country" DbType="NVarChar(15)" />
      <Column Name="Phone" Member="Phone" Storage="_Phone" DbType="NVarChar(24)" />
      <Column Name="Fax" Member="Fax" Storage="_Fax" DbType="NVarChar(24)" />
      <Association Name="FK_CustomerCustomerDemo_Customers" Member="CustomerCustomerDemos"
Storage="_CustomerCustomerDemos" ThisKey="CustomerID" OtherTable="CustomerCustomerDemo" OtherKey="CustomerID"
DeleteRule="NO ACTION" />
      <Association Name="FK_Orders_Customers" Member="Orders" Storage="_Orders" ThisKey="CustomerID"
OtherTable="Orders" OtherKey="CustomerID" DeleteRule="NO ACTION" />
    </Type>
  </Table>
</Database>
```

See Also

Creating the Object Model
External Mapping
How to: Generate the Object Model in Visual Basic or C#

How to: Generate Customized Code by Modifying a DBML File

5/1/2017 • 1 min to read • Edit Online

You can generate Visual Basic or C# source code from a database markup language (.dbml) metadata file. This approach provides an opportunity to customize the default .dbml file before you generate the application mapping code. This is an advanced feature.

The steps in this process are as follows:

1. Generate a .dbml file.
2. Use an editor to modify the .dbml file. Note that the .dbml file must validate against the schema definition (.xsd) file for LINQ to SQL .dbml files. For more information, see Code Generation in LINQ to SQL.
3. Generate the Visual Basic or C# source code.

The following examples use the SQLMetal command-line tool. For more information, see SqlMetal.exe (Code Generation Tool).

Example

The following code generates a .dbml file from the Northwind sample database. As source for the database metadata, you can use either the name of the database or the name of the .mdf file.

```
sqlmetal /server:myserver /database:northwind /dbml:mymeta.dbml
sqlmetal /dbml:mymeta.dbml mydbfile.mdf
```

Example

The following code generates Visual Basic or C# source code file from a .dbml file.

```
sqlmetal /namespace:nwind /code:nwind.vb /language:vb DBMLFile.dbml
sqlmetal /namespace:nwind /code:nwind.cs /language:csharp DBMLFile.dbml
```

See Also

Code Generation in LINQ to SQL
SqlMetal.exe (Code Generation Tool)
Creating the Object Model

How to: Validate DBML and External Mapping Files

5/1/2017 • 2 min to read • Edit Online

External mapping files and .dbml files that you modify must be validated against their respective schema definitions. This topic provides Visual Studio users with the steps to implement the validation process.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To validate a .dbml or XML file

1. On the Visual Studio **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, click the .dbml or XML mapping file that you want to validate.

   The file opens in the **XML Editor**.
3. Right-click the window, and then click **Properties**.
4. In the **Properties** window, click the ellipsis for the **Schemas** property.

   The **XML Schemas** dialog box opens.
5. Note the appropriate schema definition for your purpose.
   - DbmlSchema.xsd is the schema definition for validating a .dbml file. For more information, see Code Generation in LINQ to SQL.
   - LinqToSqlMapping.xsd is the schema definition for validating an external XML mapping file. For more information, see External Mapping.
6. In the **Use** column of the desired schema definition row, click to open the drop-down box, and then click **Use this schema**.

   The schema definition file is now associated with your DBML or XML mapping file.

   Make sure no other schema definitions are selected.
7. On the **View** menu, click **Error List**.

   Determine whether errors, warnings, or messages have been generated. If not, the XML file is valid against the schema definition.

Alternate Method for Supplying Schema Definition

If for some reason the appropriate .xsd file does not appear in the **XML Schemas** dialog box, you can download the .xsd file from a Help topic. The following steps help you save the downloaded file in the Unicode format required by the Visual Studio XML Editor.

To copy a schema definition file from a Help topic

1. Locate the Help topic that contains the schema definition as described earlier in this topic.
   - For .dbml files, see Code Generation in LINQ to SQL.

- For external mapping files, see External Mapping.

2. Click **Copy Code** to copy the code file to the Clipboard.

3. Start Notepad to create a new file.

4. Paste the code from the clipboard into Notepad file.

5. On the Notepad **File** menu, click **Save As**.

6. In the **Encoding** box, select **Unicode**.

   > **IMPORTANT**
   >
   > This selection guarantees that the Unicode-16 byte-order marker (`FFFE`) is prepended to the text file.

7. In the **File name** box, create a file name with an .xsd extension.

See Also

Reference

How to: Make Entities Serializable

5/1/2017 • 1 min to read • Edit Online

You can make entities serializable when you generate your code. Entity classes are decorated with the DataContractAttribute attribute, and columns with the DataMemberAttribute attribute.

Developers using Visual Studio can use the Object Relational Designer for this purpose.

If you are using the SQLMetal command-line tool, use the **/serialization** option with the `unidirectional` argument. For more information, see SqlMetal.exe (Code Generation Tool).

Example

The following SQLMetal command lines produce files that have serializable entities.

```
sqlmetal /code:nwserializable.vb /language:vb "c:\northwnd.mdf" /sprocs /functions /pluralize
/serialization:unidirectional
```

```
sqlmetal /code:nwserializable.cs /language:csharp "c:\northwnd.mdf" /sprocs /functions /pluralize
/serialization:unidirectional
```

See Also

Serialization
Creating the Object Model

How to: Customize Entity Classes by Using the Code Editor

5/1/2017 • 1 min to read • Edit Online

Developers using Visual Studio can use the Object Relational Designer to create or customize their entity classes.

You can also use the Visual Studio code editor to write your own mapping code or to customize code that has already been generated. For more information, see Attribute-Based Mapping.

The topics in this section describe how to customize your object model.

How to: Specify Database Names
Describes how to use Name.

How to: Represent Tables as Classes
Describes how to use TableAttribute.

How to: Represent Columns as Class Members
Describes how to use ColumnAttribute.

How to: Represent Primary Keys
Describes how to use IsPrimaryKey.

How to: Map Database Relationships
Provides examples of using the AssociationAttribute attribute.

How to: Represent Columns as Database-Generated
Describes how to use IsDbGenerated.

How to: Represent Columns as Timestamp or Version Columns
Describes how to use IsVersion.

How to: Specify Database Data Types
Describes how to use DbType.

How to: Represent Computed Columns
Describes how to use Expression.

How to: Specify Private Storage Fields
Describes how to use Storage.

How to: Represent Columns as Allowing Null Values
Describes how to use CanBeNull.

How to: Map Inheritance Hierarchies
Describes the mappings required to specify an inheritance hierarchy.

How to: Specify Concurrency-Conflict Checking
Describes how to use UpdateCheck.

See Also

SqlMetal.exe (Code Generation Tool)

How to: Specify Database Names

5/1/2017 • 1 min to read • Edit Online

Use the Name property on a DatabaseAttribute attribute to specify the name of a database when a name is not supplied by the connection.

For code samples, see Name.

To specify the name of the database

1. Add the DatabaseAttribute attribute to the class declaration for the database.

2. Add the Name property to the DatabaseAttribute attribute.

3. Set the Name property value to the name that you want to specify.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Represent Tables as Classes

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL TableAttribute attribute to designate a class as an entity class associated with a database table.

To map a class to a database table

- Add the TableAttribute attribute to the class declaration.

Example

The following code establishes the `Customer` class as an entity class that is associated with the `Customers` database table.

```
[Table(Name = "Customers")]
public class Customer
{
    // ...
}
```

```
<Table(Name:="Customers")> _
Public Class Customer
    ' ...
End Class
```

You do not have to specify the Name property if the name can be inferred. If you do not specify a name, the name is presumed to be the same name as that of the property or field.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Represent Columns as Class Members

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL ColumnAttribute attribute to associate a field or property with a database column.

To map a field or property to a database column

- Add the ColumnAttribute attribute to the property or field declaration.

Example

The following code maps the `CustomerID` field in the `Customer` class to the `CustomerID` column in the `Customers` database table.

```
[Table(Name="Customers")]
public class customer
{
    [Column(Name="CustomerID")]
    public string CustomerID;
    // ...
}
```

```
<Table(Name:="Customers")> _
Public Class Customer
    <Column(Name:="CustomerID")> _
    Public CustomerID As String
    ' ...
End Class
```

You do not have to specify the Name property if the name can be inferred. If you do not specify a name, the name is presumed to be the same name as that of the property or field.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Represent Primary Keys

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL IsPrimaryKey property on the ColumnAttribute attribute to designate a property or field to represent the primary key for a database column.

For code examples, see IsPrimaryKey.

NOTE

LINQ to SQL does not support computed columns as primary keys.

To designate a property or field as a primary key

1. Add the IsPrimaryKey property to the ColumnAttribute attribute.

2. Specify the value as `true`.

See Also

The LINQ to SQL Object Model

How to: Customize Entity Classes by Using the Code Editor

How to: Map Database Relationships

5/1/2017 • 3 min to read • Edit Online

You can encode as property references in your entity class any data relationships that will always be the same. In the Northwind sample database, for example, because customers typically place orders, there is always a relationship in the model between customers and their orders.

LINQ to SQL defines an AssociationAttribute attribute to help represent such relationships. This attribute is used together with the EntitySet<TEntity> and EntityRef<TEntity> types to represent what would be a foreign key relationship in a database. For more information, see the Association Attribute section of Attribute-Based Mapping.

> **NOTE**
> AssociationAttribute and ColumnAttribute Storage property values are case sensitive. For example, ensure that values used in the attribute for the AssociationAttribute.Storage property match the case for the corresponding property names used elsewhere in the code. This applies to all .NET programming languages, even those which are not typically case sensitive, including Visual Basic. For more information about the Storage property, see System.Data.Linq.Mapping.DataAttribute.Storage.

Most relationships are one-to-many, as in the example later in this topic. You can also represent one-to-one and many-to-many relationships as follows:

- One-to-one: Represent this kind of relationship by including EntitySet<TEntity> on both sides.

  For example, consider a `Customer` - `SecurityCode` relationship, created so that the customer's security code will not be found in the `Customer` table and can be accessed only by authorized persons.

- Many-to-many: In many-to-many relationships, the primary key of the link table (also named the *junction* table) is often formed by a composite of the foreign keys from the other two tables.

  For example, consider an `Employee` - `Project` many-to-many relationship formed by using link table `EmployeeProject`. LINQ to SQL requires that such a relationship be modeled by using three classes: `Employee`, `Project`, and `EmployeeProject`. In this case, changing the relationship between an `Employee` and a `Project` can appear to require an update of the primary key `EmployeeProject`. However, this situation is best modeled as deleting an existing `EmployeeProject` and the creating a new `EmployeeProject`.

  > **NOTE**
  > Relationships in relational databases are typically modeled as foreign key values that refer to primary keys in other tables. To navigate between them you explicitly associate the two tables by using a relational *join* operation.
  >
  > Objects in LINQ to SQL, on the other hand, refer to each other by using property references or collections of references that you navigate by using *dot* notation.

Example

In the following one-to-many example, the `Customer` class has a property that declares the relationship between customers and their orders. The `Orders` property is of type EntitySet<TEntity>. This type signifies that this relationship is one-to-many (one customer to many orders). The OtherKey property is used to describe how this association is accomplished, namely, by specifying the name of the property in the related class to be compared with this one. In this example, the `CustomerID` property is compared, just as a database *join* would compare that column value.

> **NOTE**
>
> If you are using Visual Studio, you can use the Object Relational Designer to create an association between classes.

```
[Table(Name = "Customers")]
public partial class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID;
    // ...
    private EntitySet<Order> _Orders;
    [Association(Storage = "_Orders", OtherKey = "CustomerID")]
    public EntitySet<Order> Orders
    {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
    }
}
```

```
<Table(Name:="Customers")> _
Public Class Customer
    <Column(IsPrimaryKey:=True)> _
Public CustomerID As String
    ' ...
    Private _Orders As EntitySet(Of Order)
    <Association(Storage:="_Orders", OtherKey:="CustomerID")> _
    Public Property Orders() As EntitySet(Of Order)
        Get
            Return Me._Orders
        End Get
        Set(ByVal value As EntitySet(Of Order))
            Me._Orders.Assign(value)
        End Set
    End Property
End Class
```

Example

You can also reverse the situation. Instead of using the `Customer` class to describe the association between customers and orders, you can use the `Order` class. The `Order` class uses the EntityRef<TEntity> type to describe the relationship back to the customer, as in the following code example.

> **NOTE**
>
> The EntityRef<TEntity> class supports *deferred loading*. For more information, *see* Deferred versus Immediate Loading.

```
[Table(Name = "Orders")]
public class Order
{
    [Column(IsPrimaryKey = true)]
    public int OrderID;
    [Column]
    public string CustomerID;
    private EntityRef<Customer> _Customer;
    [Association(Storage = "_Customer", ThisKey = "CustomerID")]
    public Customer Customer
    {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }
}
```

```
<Table(Name:="Orders")> _
Public Class Order
    <Column(IsPrimaryKey:=True)> _
    Public OrderID As Integer
    <Column()> _
    Public CustomerID As String
    Private _Customer As EntityRef(Of Customer)
    <Association(Storage:="Customer", ThisKey:="CustomerID")> _
    Public Property Customer() As Customer
        Get
            Return Me._Customer.Entity
        End Get
        Set(ByVal value As Customer)
            Me._Customer.Entity = value
        End Set
    End Property
End Class
```

See Also

How to: Customize Entity Classes by Using the Code Editor
The LINQ to SQL Object Model

How to: Represent Columns as Database-Generated

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL IsDbGenerated property on the ColumnAttribute attribute to designate a field or property as representing a database-generated column.

For code examples, see IsDbGenerated.

To designate a field or property as representing a database-generated column

1. Add the IsDbGenerated property to the ColumnAttribute attribute.

2. Set the property value to `true`.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Represent Columns as Timestamp or Version Columns

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL IsVersion property of the ColumnAttribute attribute to designate a field or property as representing a database column that holds database timestamps or version numbers.

For code examples, see IsVersion.

To designate a field or property as representing a timestamp or version column

1. Add the IsVersion property to the ColumnAttribute attribute.

2. Set the IsVersion property value to `true`.

See Also

The LINQ to SQL Object Model
How to: Specify Which Members are Tested for Concurrency Conflicts
How to: Customize Entity Classes by Using the Code Editor

How to: Specify Database Data Types

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL DbType property on a ColumnAttribute attribute to specify the exact text that defines the column in a T-SQL table declaration.

You must specify the DbType property only if you plan to use CreateDatabase to create an instance of the database.

For code examples, see DbType.

To specify text to define a data type in a T-SQL table

1. Add the DbType property to the ColumnAttribute attribute.

2. Set the value of the DbType property to the exact text that is used by T-SQL.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Represent Computed Columns

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL Expression property on a ColumnAttribute attribute to represent a column whose contents are the result of calculation.

For code examples, see Expression.

NOTE
LINQ to SQL does not support computed columns as primary keys.

To represent a computed column

1. Add the Expression property to the ColumnAttribute attribute.

2. Assign a string representation of the formula to the Expression property.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Specify Private Storage Fields

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL Storage property on the DataAttribute attribute to designate the name of an underlying storage field.

For code examples, see Storage.

To specify the name of an underlying storage field

1. Add the Storage property to the ColumnAttribute attribute.

2. Assign the name of the field as the value of the Storage property.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Represent Columns as Allowing Null Values

5/1/2017 • 1 min to read • Edit Online

Use the LINQ to SQL CanBeNull property on the ColumnAttribute attribute to specify that the associated database column can hold null values.

For code examples, see CanBeNull.

To designate a column as allowing null values

1. Add the CanBeNull property to the ColumnAttribute attribute.
2. Set the CanBeNull property value to `true`.

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

How to: Map Inheritance Hierarchies

5/10/2017 • 2 min to read • Edit Online

To implement inheritance mapping in LINQ, you must specify the attributes and attribute properties on the root class of the inheritance hierarchy as described in the following steps. Developers using Visual Studio can use the Object Relational Designer to map inheritance hierarchies. See How to: Configure inheritance by using the O/R Designer.

NOTE

No special attributes or properties are required on the subclasses. Note especially that subclasses do not have the TableAttribute attribute.

To map an inheritance hierarchy

1. Add the TableAttribute attribute to the root class.

2. Also to the root class, add an InheritanceMappingAttribute attribute for each class in the hierarchy structure.

3. For each InheritanceMappingAttribute attribute, define a Code property.

   This property holds a value that appears in the database table in the IsDiscriminator column to indicate which class or subclass this row of data belongs to.

4. For each InheritanceMappingAttribute attribute, also add a Type property.

   This property holds a value that specifies which class or subclass the key value signifies.

5. On only one of the InheritanceMappingAttribute attributes, add an IsDefault property.

   This property serves to designate a *fallback* mapping when the discriminator value from the database table does not match any Code value in the inheritance mappings.

6. Add an IsDiscriminator property for a ColumnAttribute attribute.

   This property signifies that this is the column that holds the Code value.

Example

NOTE

If you are using Visual Studio, you can use the Object Relational Designer to configure inheritance. See How to: Configure inheritance by using the O/R Designer

In the following code example, `Vehicle` is defined as the root class, and the previous steps have been implemented to describe the hierarchy for LINQ.

```
[Table]
[InheritanceMapping(Code = "C", Type = typeof(Car))]
[InheritanceMapping(Code = "T", Type = typeof(Truck))]
[InheritanceMapping(Code = "V", Type = typeof(Vehicle),
    IsDefault = true)]
public class Vehicle
{
    [Column(IsDiscriminator = true)]
    public string DiscKey;
    [Column(IsPrimaryKey = true)]
    public string VIN;
    [Column]
    public string MfgPlant;
}
public class Car : Vehicle
{
    [Column]
    public int TrimCode;
    [Column]
    public string ModelName;
}

public class Truck : Vehicle
{
    [Column]
    public int Tonnage;
    [Column]
    public int Axles;
}
```

```
<Table()> _
<InheritanceMapping(Code:="C", Type:=GetType(Car))> _
<InheritanceMapping(Code:="T", Type:=GetType(Truck))> _
<InheritanceMapping(Code:="V", Type:=GetType(Vehicle), _
    IsDefault:=True)> _
Public Class Vehicle
    <Column(IsDiscriminator:=True)> _
        Private DiscKey As String
    <Column(IsPrimaryKey:=True)> _
        Private VIN As String
    <Column()> _
        Private MfgPlant As String
End Class

Public Class Car
    Inherits Vehicle
    <Column()> _
        Private TrimCode As Integer
    <Column()> _
        Private ModelName As String
End Class

Public Class Truck
    Inherits Vehicle
    <Column()> _
        Private Tonnage As Integer
    <Column()> _
        Private Axles As Integer
End Class
```

See Also

Inheritance Support

How to: Customize Entity Classes by Using the Code Editor

How to: Specify Concurrency-Conflict Checking

5/1/2017 • 1 min to read • Edit Online

You can specify which columns of the database are to be checked for concurrency conflicts when you call SubmitChanges. For more information, see How to: Specify Which Members are Tested for Concurrency Conflicts.

Example

The following code specifies that the `HomePage` member should never be tested during update checks. For more information, see UpdateCheck.

```
[Column(Storage="_HomePage", DbType="NText", UpdateCheck=UpdateCheck.Never)]
public string HomePage
{
    get
    {
        return this._HomePage;
    }
    set
    {
        if ((this._HomePage != value))
 {
      this.OnHomePageChanging(value);
     this.SendPropertyChanging();
            this._HomePage = value;
     this.SendPropertyChanged("HomePage");
            this.OnHomePageChanged();
 }
    }
}
```

```
<Column(Storage:="_HomePage", DbType:="NText", UpdateCheck:=UpdateCheck.Never)>  _
Public Property HomePage() As String
    Get
        Return Me._HomePage
    End Get
    Set(ByVal value As String)
        If ((Me._HomePage <> value)  _
            = false) Then
     Me.OnHomePageChanging(value)
            Me.SendPropertyChanging
            Me._HomePage = value
            Me.SendPropertyChanged("HomePage")
            Me.OnHomePageChanged
        End If
    End Set
End Property
```

See Also

The LINQ to SQL Object Model
How to: Customize Entity Classes by Using the Code Editor

Communicating with the Database

5/1/2017 • 1 min to read • Edit Online

The topics in this section describe some basic aspects of how you establish and maintain communication with the database.

In This Section

How to: Connect to a Database
Describes how to use the DataContext class to connect to a database.

How to: Directly Execute SQL Commands
Describes how you can use ExecuteCommand to send SQL-language commands.

How to: Reuse a Connection Between an ADO.NET Command and a DataContext
Provides examples of how to use an existing ADO.NET connection in a LINQ to SQL application.

See Also

Programming Guide

How to: Connect to a Database

5/10/2017 • 2 min to read • Edit Online

The DataContext is the main conduit by which you connect to a database, retrieve objects from it, and submit changes back to it. You use the DataContext just as you would use an ADO.NET SqlConnection. In fact, the DataContext is initialized with a connection or connection string that you supply. For more information, see DataContext Methods (O/R Designer).

The purpose of the DataContext is to translate your requests for objects into SQL queries to be made against the database, and then to assemble objects out of the results. The DataContext enables Language-Integrated Query (LINQ) by implementing the same operator pattern as the Standard Query Operators, such as `Where` and `Select`.

IMPORTANT

Maintaining a secure connection is of the highest importance. For more information, see Security in LINQ to SQL.

Example

In the following example, the DataContext is used to connect to the Northwind sample database and to retrieve rows of customers whose city is London.

```
// DataContext takes a connection string.
DataContext db = new DataContext(@"c:\Northwnd.mdf");

// Get a typed table to run queries.
Table<Customer> Customers = db.GetTable<Customer>();

// Query for customers from London.
var query =
    from cust in Customers
    where cust.City == "London"
    select cust;

foreach (var cust in query)
    Console.WriteLine("id = {0}, City = {1}", cust.CustomerID, cust.City);
```

```
' DataContext takes a connection string.
Dim db As New DataContext("…\Northwnd.mdf")

' Get a typed table to run queries.
Dim Customers As Table(Of Customer) = db.GetTable(Of Customer)()

' Query for customer from London.
Dim Query = _
    From cust In Customers _
    Where cust.City = "London" _
    Select cust

For Each cust In Query
    Console.WriteLine("id=" & cust.CustomerID & _
        ", City=" & cust.City)
Next
```

Each database table is represented as a `Table` collection available by way of the GetTable method, by using the entity class to identify it.

Example

Best practice is to declare a strongly typed DataContext instead of relying on the basic DataContext class and the GetTable method. A strongly typed DataContext declares all `Table` collections as members of the context, as in the following example.

```
public partial class Northwind : DataContext
{
    public Table<Customer> Customers;
    public Table<Order> Orders;
    public Northwind(string connection) : base(connection) { }
}
```

```
Partial Public Class Northwind
    Inherits DataContext
    Public Customers As Table(Of Customer)
    Public Orders As Table(Of Order)
    Public Sub New(ByVal connection As String)
        MyBase.New(connection)
    End Sub
End Class
```

You can then express the query for customers from London more simply as:

```
Northwnd db = new Northwnd(@"c:\Northwnd.mdf");
var query =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
foreach (var cust in query)
    Console.WriteLine("id = {0}, City = {1}", cust.CustomerID,
        cust.City);
```

```
Dim db As New Northwind("...\Northwnd.mdf")

Dim query = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust

For Each cust In query
    Console.WriteLine("id=" & cust.CustomerID & _
        ", City=" & cust.City)
Next
```

See Also

Communicating with the Database

How to: Directly Execute SQL Commands

5/1/2017 • 1 min to read • Edit Online

Assuming a DataContext connection, you can use ExecuteCommand to execute SQL commands that do not return objects.

Example

The following example causes SQL Server to increase UnitPrice by 1.00.

```
db.ExecuteCommand("UPDATE Products SET UnitPrice = UnitPrice + 1.00");
```

```
    db.ExecuteCommand _
("UPDATE Products SET UnitPrice = UnitPrice + 1.00")
```

See Also

How to: Directly Execute SQL Queries
Communicating with the Database

How to: Reuse a Connection Between an ADO.NET Command and a DataContext

5/1/2017 • 1 min to read • Edit Online

Because LINQ to SQL is a part of the ADO.NET family of technologies and is based on services provided by ADO.NET, you can reuse a connection between an ADO.NET command and a DataContext.

Example

The following example shows how to reuse the same connection between an ADO.NET command and the DataContext.

```
string connString = @"Data Source=.\SQLEXPRESS;AttachDbFilename=c:\northwind.mdf;
    Integrated Security=True; Connect Timeout=30; User Instance=True";
SqlConnection nwindConn = new SqlConnection(connString);
nwindConn.Open();

Northwnd interop_db = new Northwnd(nwindConn);

SqlTransaction nwindTxn = nwindConn.BeginTransaction();

try
{
    SqlCommand cmd = new SqlCommand(
        "UPDATE Products SET QuantityPerUnit = 'single item' WHERE ProductID = 3");
    cmd.Connection = nwindConn;
    cmd.Transaction = nwindTxn;
    cmd.ExecuteNonQuery();

    interop_db.Transaction = nwindTxn;

    Product prod1 = interop_db.Products
        .First(p => p.ProductID == 4);
    Product prod2 = interop_db.Products
        .First(p => p.ProductID == 5);
    prod1.UnitsInStock -= 3;
    prod2.UnitsInStock -= 5;

    interop_db.SubmitChanges();

    nwindTxn.Commit();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine("Error submitting changes... all changes rolled back.");
}

nwindConn.Close();
```

```
Dim conString = "Data Source=.\SQLEXPRESS;AttachDbFilename=c:\northwind.mdf; Integrated Security=True;Connect
Timeout=30;User Instance=True"
Dim northwindCon = New SqlConnection(conString)
northwindCon.Open()

Dim db = New Northwnd("...")
Dim northwindTransaction = northwindCon.BeginTransaction()

Try
    Dim cmd = New SqlCommand( _
            "UPDATE Products SET QuantityPerUnit = 'single item' " & _
            "WHERE ProductID = 3")
    cmd.Connection = northwindCon
    cmd.Transaction = northwindTransaction
    cmd.ExecuteNonQuery()

    db.Transaction = northwindTransaction

    Dim prod1 = (From prod In db.Products _
  Where prod.ProductID = 4).First
    Dim prod2 = (From prod In db.Products _
  Where prod.ProductID = 5).First
    prod1.UnitsInStock -= 3
    prod2.UnitsInStock -= 5

    db.SubmitChanges()

    northwindTransaction.Commit()

Catch e As Exception

    Console.WriteLine(e.Message)
    Console.WriteLine("Error submitting changes... " & _
"all changes rolled back.")
End Try

northwindCon.Close()
```

See Also

Background Information
ADO.NET and LINQ to SQL
Communicating with the Database

Querying the Database

5/1/2017 • 1 min to read • Edit Online

This group of topics describes how to develop and execute queries in LINQ to SQL projects.

In This Section

How to: Query for Information

5/1/2017 • 1 min to read • Edit Online

Queries in LINQ to SQL use the same syntax as queries in LINQ. The only difference is that the objects referenced in LINQ to SQL queries are mapped to elements in a database. For more information, see Introduction to LINQ Queries (C#).

LINQ to SQL translates the queries you write into equivalent SQL queries and sends them to the server for processing.

Some features of LINQ queries might need special attention in LINQ to SQL applications. For more information, see Query Concepts.

Example

The following query asks for a list of customers from London. In this example, `Customers` is a table in the Northwind sample database.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

' Query for customers in London.
Dim custQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust
```

See Also

Creating the Object Model
Downloading Sample Databases
Querying the Database

How to: Retrieve Information As Read-Only

5/1/2017 • 1 min to read • Edit Online

When you do not intend to change the data, you can increase the performance of queries by seeking read-only results.

You implement read-only processing by setting ObjectTrackingEnabled to `false`.

NOTE

When ObjectTrackingEnabled is set to `false`, DeferredLoadingEnabled is implicitly set to `false`.

Example

The following code retrieves a read-only collection of employee hire dates.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

db.ObjectTrackingEnabled = false;
IOrderedQueryable<Employee> hireQuery =
    from emp in db.Employees
    orderby emp.HireDate
    select emp;

foreach (Employee empObj in hireQuery)
{
    Console.WriteLine("EmpID = {0}, Date Hired = {1}",
        empObj.EmployeeID, empObj.HireDate);
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

db.ObjectTrackingEnabled = False
Dim hireQuery = _
    From emp In db.Employees _
    Select emp _
    Order By emp.HireDate

For Each empObj As Employee In hireQuery
    Console.WriteLine("EmpID = {0}, Date Hired = {1}", _
            empObj.EmployeeID, empObj.HireDate)
Next
```

See Also

Query Concepts
Querying the Database
Deferred versus Immediate Loading

How to: Control How Much Related Data Is Retrieved

5/1/2017 • 1 min to read • Edit Online

Use the LoadWith method to specify which data related to your main target should be retrieved at the same time. For example, if you know you will need information about customers' orders, you can use LoadWith to make sure that the order information is retrieved at the same time as the customer information. This approach results in only one trip to the database for both sets of information.

NOTE

You can retrieve data related to the main target of your query by retrieving a cross-product as one large projection, such as retrieving orders when you target customers. But this approach often has disadvantages. For example, the results are just projections and not entities that can be changed and persisted by LINQ to SQL. And you can be retrieving lots of data that you do not need.

Example

In the following example, all the `Orders` for all the `Customers` who are located in London are retrieved when the query is executed. As a result, successive access to the `Orders` property on a `Customer` object does not trigger a new database query.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
db.LoadOptions = dlo;

var londonCustomers =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (var custObj in londonCustomers)
{
    Console.WriteLine(custObj.CustomerID);
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

Dim dlo As DataLoadOptions = New DataLoadOptions()
dlo.LoadWith(Of Customer)(Function(c As Customer) c.Orders)
db.LoadOptions = dlo

Dim londonCustomers = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust

For Each custObj In londonCustomers
    Console.WriteLine(custObj.CustomerID)
Next
```

See Also

How to: Filter Related Data

5/1/2017 • 1 min to read • Edit Online

Use the AssociateWith method to specify sub-queries to limit the amount of retrieved data.

Example

In the following example, the AssociateWith method limits the `Orders` retrieved to those that have not been shipped today. Without this approach, all `Orders` would have been retrieved even though only a subset is desired.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
DataLoadOptions dlo = new DataLoadOptions();
dlo.AssociateWith<Customer>(c => c.Orders.Where(p => p.ShippedDate != DateTime.Today));
db.LoadOptions = dlo;
var custOrderQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custOrderQuery)
{
    Console.WriteLine(custObj.CustomerID);
    foreach (Order ord in custObj.Orders)
    {
        Console.WriteLine("\t {0}",ord.OrderDate);
    }
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

Dim dlo As DataLoadOptions = New DataLoadOptions()
dlo.AssociateWith(Of Customer)(Function(c As Customer) _
        c.Orders.Where(Function(p) p.ShippedDate <> DateTime.Today))
db.LoadOptions = dlo

Dim custOrderQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust

For Each custObj In custOrderQuery
    Console.WriteLine(custObj.CustomerID)
    For Each ord In custObj.Orders
        Console.WriteLine("{0}{1}", vbTab, ord.OrderDate)
    Next

Next
```

See Also

Querying the Database

How to: Turn Off Deferred Loading

5/1/2017 • 1 min to read • Edit Online

You can turn off deferred loading by setting DeferredLoadingEnabled to `false`. For more information, see Deferred versus Immediate Loading.

NOTE

Deferred loading is turned off by implication when object tracking is turned off. For more information, see How to: Retrieve Information As Read-Only.

Example

The following example shows how to turn off deferred loading by setting DeferredLoadingEnabled to `false`.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
db.DeferredLoadingEnabled = false;

DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Customer>(c => c.Orders);
ds.LoadWith<Order>(o => o.OrderDetails);
db.LoadOptions = ds;

var custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custQuery)
{
    Console.WriteLine("Customer ID: {0}", custObj.CustomerID);
    foreach (Order ord in custObj.Orders)
    {
        Console.WriteLine("\tOrder ID: {0}", ord.OrderID);
        foreach (OrderDetail detail in ord.OrderDetails)
        {
            Console.WriteLine("\t\tProduct ID: {0}", detail.ProductID);
        }
    }
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

db.DeferredLoadingEnabled = False

Dim ds As New DataLoadOptions()
ds.LoadWith(Function(c As Customer) c.Orders)
ds.LoadWith(Of Order)(Function(o) o.OrderDetails)
db.LoadOptions = ds

Dim custQuery = From cust In db.Customers _
    Where cust.City = "London" _
    Select cust

For Each custObj In custQuery
    Console.WriteLine("Customer ID: {0}", custObj.CustomerID)
    For Each ord In custObj.Orders
        Console.WriteLine(vbTab & "Order ID: {0}", ord.OrderID)
        For Each detail In ord.OrderDetails
            Console.WriteLine(vbTab & vbTab & "Product ID: {0}", _
                detail.ProductID)
        Next
    Next
Next
```

See Also

Query Concepts
Querying the Database

How to: Directly Execute SQL Queries

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL translates the queries you write into parameterized SQL queries (in text form) and sends them to the SQL server for processing.

SQL cannot execute the variety of methods that might be locally available to your application. LINQ to SQL tries to convert these local methods to equivalent operations and functions that are available inside the SQL environment. Most methods and operators on .NET Framework built-in types have direct translations to SQL commands. Some can be produced from the functions that are available. Those that cannot be produced generate run-time exceptions. For more information, see SQL-CLR Type Mapping.

In cases where a LINQ to SQL query is insufficient for a specialized task, you can use the ExecuteQuery method to execute a SQL query, and then convert the result of your query directly into objects.

Example

In the following example, assume that the data for the `Customer` class is spread over two tables (customer1 and customer2). The query returns a sequence of `Customer` objects.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
IEnumerable<Customer> results = db.ExecuteQuery<Customer>
(@"SELECT c1.custid as CustomerID, c2.custName as ContactName
    FROM customer1 as c1, customer2 as c2
    WHERE c1.custid = c2.custid"
);
```

```
Dim db As New Northwnd("c:\northwnd.mdf")
Dim results As IEnumerable(Of Customer) = _
    db.ExecuteQuery(Of Customer) _
    ("SELECT c1.custID as CustomerID," & _
    "c2.custName as ContactName" & _
    "FROM customer1 AS c1, customer2 as c2" & _
    "WHERE c1.custid = c2.custid")
```

As long as the column names in the tabular results match column properties of your entity class, LINQ to SQL creates your objects out of any SQL query.

Example

The ExecuteQuery method also allows for parameters. Use code such as the following to execute a parameterized query.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
IEnumerable<Customer> results = db.ExecuteQuery<Customer>
    ("SELECT contactname FROM customers WHERE city = {0}",
    "London");
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim results As IEnumerable(Of Customer) = _
db.ExecuteQuery(Of Customer) _
("SELECT contactname FROM customers WHERE city = {0}, 'London'")
```

The parameters are expressed in the query text by using the same curly notation used by `Console.WriteLine()` and `String.Format()`. In fact, `String.Format()` is actually called on the query string you provide, substituting the curly braced parameters with generated parameter names such as @p0, @p1 …, @p(n).

See Also

Background Information
Querying the Database

How to: Store and Reuse Queries

5/1/2017 • 2 min to read • Edit Online

When you have an application that executes structurally similar queries many times, you can often increase performance by compiling the query one time and executing it several times with different parameters. For example, an application might have to retrieve all the customers who are in a particular city, where the city is specified at runtime by the user in a form. LINQ to SQL supports the use of *compiled queries* for this purpose.

> **NOTE**
>
> This pattern of usage represents the most common use for compiled queries. Other approaches are possible. For example, compiled queries can be stored as static members on a partial class that extends the code generated by the designer.

Example

In many scenarios you might want to reuse the queries across thread boundaries. In such cases, storing the compiled queries in static variables is especially effective. The following code example assumes a `Queries` class designed to store compiled queries, and assumes a Northwind class that represents a strongly typed DataContext.

```
public static Func<Northwnd, string, IQueryable<Customer>>
    CustomersByCity =
        CompiledQuery.Compile((Northwnd db, string city) =>
            from c in db.Customers where c.City == city select c);

public static Func<Northwnd, string, IQueryable<Customer>>
    CustomersById = CompiledQuery.Compile((Northwnd db,
    string id) => db.Customers.Where(c => c.CustomerID == id));
```

```
Class Queries

    Public Shared CustomersByCity As  _
        Func(Of Northwnd, String, IQueryable(Of Customer)) = _
            CompiledQuery.Compile(Function(db As Northwnd, _
    city As String) _
        From c In db.Customers Where c.City = city Select c)

    Public Shared CustomersById As  _
        Func(Of Northwnd, String, IQueryable(Of Customer)) = _
            CompiledQuery.Compile(Function(db As Northwnd, _
    id As String) _
        db.Customers.Where(Function(c) c.CustomerID = id))

End Class
```

```
// The following example invokes such a compiled query in the main
// program.

public IEnumerable<Customer> GetCustomersByCity(string city)
{
    var myDb = GetNorthwind();
    return Queries.CustomersByCity(myDb, city);
}
```

```
' The following example invokes such a compiled query in the main
' program
Public Function GetCustomersByCity(ByVal city As String) As  _
    IEnumerable(Of Customer)

    Dim myDb = GetNorthwind()
    Return Queries.CustomersByCity(myDb, city)
End Function
```

Example

You cannot currently store (in static variables) queries that return an *anonymous type*, because type has no name to provide as a generic argument. The following example shows how you can work around the issue by creating a type that can represent the result, and then use it as a generic argument.

```
class SimpleCustomer
{
    public string ContactName { get; set; }
}

class Queries2
{
    public static Func<Northwnd, string, IEnumerable<SimpleCustomer>> CustomersByCity =
        CompiledQuery.Compile<Northwnd, string, IEnumerable<SimpleCustomer>>(
        (Northwnd db, string city) =>
        from c in db.Customers
        where c.City == city
        select new SimpleCustomer { ContactName = c.ContactName });
}
```

```
Class SimpleCustomer
    Private _ContactName As String
    Public Property ContactName() As String
        Get
            Return _ContactName
        End Get
        Set(ByVal value As String)
            _ContactName = value
        End Set
    End Property
End Class

Class Queries2
    Public Shared CustomersByCity As Func(Of Northwnd, String, IEnumerable(Of SimpleCustomer)) = _
        CompiledQuery.Compile(Of Northwnd, String, IEnumerable(Of SimpleCustomer))( _
        Function(db As Northwnd, city As String) _
        From c In db.Customers _
        Where c.City = city _
        Select New SimpleCustomer With {.ContactName = c.ContactName})
End Class
```

See Also

CompiledQuery
Query Concepts
Querying the Database

How to: Handle Composite Keys in Queries

5/1/2017 • 1 min to read • Edit Online

Some operators can take only one argument. If your argument must include more than one column from the database, you must create an anonymous type to represent the combination.

Example

The following example shows a query that invokes the `GroupBy` operator, which can take only one `key` argument.

```
        var query =
from cust in db.Customers
group cust.ContactName by new { City = cust.City, Region = cust.Region };

        foreach (var grp in query)
        {
            Console.WriteLine("\nLocation Key: {0}", grp.Key);
            foreach (var listing in grp)
            {
                Console.WriteLine("\t{0}", listing);
            }
        }
```

```
Dim query = _
From cust In db.Customers _
Group cust.ContactName By Key = New With {cust.City, cust.Region} _
Into Group

For Each grp In query
    Console.WriteLine("Location Key: {0}", grp.Key)
    For Each listing In grp.Group
        Console.WriteLine(vbTab & "0}", listing)
    Next
Next
```

Example

The same situation pertains to joins, as in the following example:

```
        var query =
from ord in db.Orders
from prod in db.Products
join det in db.OrderDetails
    on new { ord.OrderID, prod.ProductID } equals new { det.OrderID, det.ProductID }
    into details
from det in details
select new { ord.OrderID, prod.ProductID, det.UnitPrice };
```

```
Dim query = From ord In db.Orders, prod In db.Products _
    Join det In db.OrderDetails _
    On New With {ord.OrderID, prod.ProductID} Equals _
    New With {det.OrderID, det.ProductID} _
    Select ord.OrderID, prod.ProductID, det.UnitPrice
```

See Also

Query Concepts

How to: Retrieve Many Objects At Once

5/1/2017 • 1 min to read • Edit Online

You can retrieve many objects in one query by using LoadWith.

Example

The following code uses the LoadWith method to retrieve both `Customer` and `Order` objects.

```
Northwnd db = new Northwnd(@"northwnd.mdf");
DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Customer>(c => c.Orders);
ds.LoadWith<Order>(o => o.OrderDetails);
db.LoadOptions = ds;

var custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custQuery)
{
    Console.WriteLine("Customer ID: {0}", custObj.CustomerID);
    foreach (Order ord in custObj.Orders)
    {
        Console.WriteLine("\tOrder ID: {0}", ord.OrderID);
        foreach (OrderDetail detail in ord.OrderDetails)
        {
            Console.WriteLine("\t\tProduct ID: {0}", detail.ProductID);
        }
    }
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")
Dim ds As DataLoadOptions = New DataLoadOptions()
ds.LoadWith(Of Customer)(Function(c) c.Orders)
ds.LoadWith(Of Order)(Function(o) o.OrderDetails)
db.LoadOptions = ds

Dim custQuery = From cust In db.Customers() _
                Where cust.City = "London" _
                Select cust

For Each custObj In custQuery
    Console.WriteLine("Customer ID: " & custObj.CustomerID)

    For Each ord In custObj.Orders
        Console.WriteLine(vbTab & "Order ID: " & ord.OrderID)

        For Each detail In ord.OrderDetails
            Console.WriteLine(vbTab & vbTab & _
                        "Product ID: {0}", detail.ProductID)
        Next
    Next
Next
```

See Also

How to: Filter at the DataContext Level

5/1/2017 • 1 min to read • Edit Online

You can filter `EntitySets` at the `DataContext` level. Such filters apply to all queries done with that DataContext instance.

Example

In the following example, System.Data.Linq.DataLoadOptions.AssociateWith(LambdaExpression) is used to filter the pre-loaded orders for customers by `ShippedDate`.

```
Northwnd db = new Northwnd(@"northwnd.mdf");
// Preload Orders for Customer.
// One directive per relationship to be preloaded.
DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Customer>(c => c.Orders);
ds.AssociateWith<Customer>
    (c => c.Orders.Where(p => p.ShippedDate != DateTime.Today));
db.LoadOptions = ds;

var custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custQuery)
{
    Console.WriteLine("Customer ID: {0}", custObj.CustomerID);
    foreach (Order ord in custObj.Orders)
    {
        Console.WriteLine("\tOrder ID: {0}", ord.OrderID);
        foreach (OrderDetail detail in ord.OrderDetails)
        {
            Console.WriteLine("\t\tProduct ID: {0}", detail.ProductID);
        }
    }
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")
' Preload Orders for Customer.
' One directive per relationship to be preloaded.
Dim ds As DataLoadOptions = New DataLoadOptions()
ds.LoadWith(Of Customer)(Function(cust) cust.Orders)
ds.AssociateWith(Of Customer)( _
    Function(cust) _
        From ord In cust.Orders _
        Where ord.ShippedDate <> DateTime.Today)

db.LoadOptions = ds

Dim custQuery = From cust In db.Customers _
                Where cust.City = "London" _
                Select cust

For Each custObj In custQuery
    Console.WriteLine("Customer ID: " & custObj.CustomerID)

    For Each ord In custObj.Orders
        Console.WriteLine(vbTab & "Order ID: " & ord.OrderID)

        For Each detail In ord.OrderDetails
            Console.WriteLine(vbTab & vbTab & _
                "Product ID: " & detail.ProductID)
        Next
    Next
Next
```

See Also

Query Concepts

Query Examples

5/1/2017 • 1 min to read • Edit Online

This section provides Visual Basic and C# examples of typical LINQ to SQL queries. Developers using Visual Studio can find many more examples in a sample solution available in the Samples section. For more information, see Samples.

IMPORTANT

db is often used in code examples in LINQ to SQL documentation. *db* is assumed to be an instance of a *Northwind* class, which inherits from DataContext.

In This Section

Aggregate Queries
Describes how to use Average, Count, and so forth.

Return the First Element in a Sequence
Provides examples of using First.

Return Or Skip Elements in a Sequence
Provides examples of using Take and Skip.

Sort Elements in a Sequence
Provides examples of using OrderBy.

Group Elements in a Sequence
Provides examples of using GroupBy.

Eliminate Duplicate Elements from a Sequence
Provides examples of using Distinct.

Determine if Any or All Elements in a Sequence Satisfy a Condition
Provides examples of using All and Any.

Concatenate Two Sequences
Provides examples of using Concat.

Return the Set Difference Between Two Sequences
Provides examples of using Except.

Return the Set Intersection of Two Sequences
Provides examples of using Intersect.

Return the Set Union of Two Sequences
Provides examples of using Union.

Convert a Sequence to an Array
Provides examples of using ToArray.

Convert a Sequence to a Generic List
Provides examples of using ToList.

Convert a Type to a Generic IEnumerable

Provides examples of using AsEnumerable.

Formulate Joins and Cross-Product Queries
Provides examples of using foreign-key navigation in the `from`, `where`, and `select` clauses.

Formulate Projections
Provides examples of combining `select` with other features (for example, *anonymous types*) to form query projections.

Related Sections

Standard Query Operators Overview
Explains the concept of standard query operators.

Query Concepts
Explains how LINQ to SQL uses concepts that apply to queries.

Programming Guide
Provides a portal to topics that explain programming concepts related to LINQ to SQL.

Aggregate Queries

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL supports the `Average`, `Count`, `Max`, `Min`, and `Sum` aggregate operators. Note the following characteristics of aggregate operators in LINQ to SQL:

- Aggregate queries are executed immediately.

  For more information, see Introduction to LINQ Queries (C#).

- Aggregate queries typically return a number instead of a collection.

  For more information, see Aggregation Operations.

- You cannot call aggregates against anonymous types.

The examples in the following topics derive from the Northwind sample database. For more information, see Downloading Sample Databases.

In This Section

Return the Average Value From a Numeric Sequence
Demonstrates how to use the Average operator.

Count the Number of Elements in a Sequence
Demonstrates how to use the Count operator.

Find the Maximum Value in a Numeric Sequence
Demonstrates how to use the Max operator.

Find the Minimum Value in a Numeric Sequence
Demonstrates how to use the Min operator.

Compute the Sum of Values in a Numeric Sequence
Demonstrates how to use the Sum operator.

Related Sections

Query Examples
Provides links to LINQ to SQL queries in Visual Basic and C#.

Query Concepts
Provides links to topics that explain concepts for designing LINQ queries in LINQ to SQL.

Introduction to LINQ Queries (C#)
Explains how queries work in LINQ.

Return the Average Value From a Numeric Sequence

5/1/2017 • 1 min to read • Edit Online

The Average operator computes the average of a sequence of numeric values.

NOTE

The LINQ to SQL translation of `Average` of integer values is computed as an integer, not as a double.

Example

The following example returns the average of `Freight` values in the `Orders` table.

Results from the sample Northwind database would be `78.2442`.

```
System.Nullable<Decimal> averageFreight =
    (from ord in db.Orders
    select ord.Freight)
    .Average();

Console.WriteLine(averageFreight);
```

```
Dim averageFreight = Aggregate ord In db.Orders _
                     Into Average(ord.Freight)

Console.WriteLine(averageFreight)
```

Example

The following example returns the average of the unit price of all `Products` in the `Products` table.

Results from the sample Northwind database would be `28.8663`.

```
System.Nullable<Decimal> averageUnitPrice =
    (from prod in db.Products
    select prod.UnitPrice)
    .Average();

Console.WriteLine(averageUnitPrice);
```

```
Dim averageUnitPrice = Aggregate prod In db.Products _
                       Into Average(prod.UnitPrice)

Console.WriteLine(averageUnitPrice)
```

Example

The following example uses the `Average` operator to find those `Products` whose unit price is higher than the average unit price of the category it belongs to. The example then displays the results in groups.

Note that this example requires the use of the `var` keyword in C#, because the return type is anonymous.

```
var priceQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    select new
    {
        grouping.Key,
        ExpensiveProducts =
            from prod2 in grouping
            where prod2.UnitPrice > grouping.Average(prod3 =>
                prod3.UnitPrice)
        select prod2
    };

foreach (var grp in priceQuery)
{
    Console.WriteLine(grp.Key);
    foreach (var listing in grp.ExpensiveProducts)
    {
        Console.WriteLine(listing.ProductName);
    }
}
```

```
Dim priceQuery = From prod In db.Products() _
    Group prod By prod.CategoryID Into grouping = Group _
    Select CategoryID, _
    ExpensiveProducts = _
        (From prod2 In grouping _
        Where prod2.UnitPrice > _
        grouping.Average(Function(prod3) _
        prod3.UnitPrice) _
        Select prod2)

For Each grp In priceQuery
    Console.WriteLine(grp.CategoryID)
    For Each listing In grp.ExpensiveProducts
        Console.WriteLine(listing.ProductName)
    Next
Next
```

If you run this query against the Northwind sample database, the results should resemble of the following:

`1`

`Côte de Blaye`

`Ipoh Coffee`

`2`

`Grandma's Boysenberry Spread`

`Northwoods Cranberry Sauce`

`Sirop d'érable`

`Vegie-spread`

`3`

`Sir Rodney's Marmalade`

`Gumbär Gummibärchen`

`Schoggi Schokolade`

`Tarte au sucre`

`4`

`Queso Manchego La Pastora`

`Mascarpone Fabioli`

`Raclette Courdavault`

`Camembert Pierrot`

`Gudbrandsdalsost`

`Mozzarella di Giovanni`

`5`

`Gustaf's Knäckebröd`

`Gnocchi di nonna Alice`

`Wimmers gute Semmelknödel`

`6`

`Mishi Kobe Niku`

`Thüringer Rostbratwurst`

`7`

`Rössle Sauerkraut`

`Manjimup Dried Apples`

`8`

`Ikura`

`Carnarvon Tigers`

`Nord-Ost Matjeshering`

`Gravad lax`

See Also

Aggregate Queries

Count the Number of Elements in a Sequence

5/1/2017 • 1 min to read • Edit Online

Use the Count operator to count the number of elements in a sequence.

Running this query against the Northwind sample database produces an output of `91`.

Example

The following example counts the number of `Customers` in the database.

```
System.Int32 customerCount = db.Customers.Count();
Console.WriteLine(customerCount);
```

```
Dim customerCount = db.Customers.Count()
Console.WriteLine(customerCount)
```

Example

The following example counts the number of products in the database that have not been discontinued.

Running this example against the Northwind sample database produces an output of `69`.

```
System.Int32 notDiscontinuedCount =
    (from prod in db.Products
    where !prod.Discontinued
    select prod)
    .Count();

Console.WriteLine(notDiscontinuedCount);
```

```
Dim notDiscontinuedCount = Aggregate prod In db.Products _
                           Into Count(Not prod.Discontinued)

Console.WriteLine(notDiscontinuedCount)
```

See Also

Aggregate Queries
Downloading Sample Databases

Find the Maximum Value in a Numeric Sequence

5/1/2017 • 1 min to read • Edit Online

Use the Max operator to find the highest value in a sequence of numeric values.

Example

The following example finds the latest date of hire for any employee.

If you run this query against the sample Northwind database, the output is: `11/15/1994 12:00:00 AM`.

```
System.Nullable<DateTime> latestHireDate =
    (from emp in db.Employees
    select emp.HireDate)
    .Max();

Console.WriteLine(latestHireDate);
```

```
Dim latestHireDate = Aggregate emp In db.Employees _
                     Into Max(emp.HireDate)

Console.WriteLine(latestHireDate)
```

Example

The following example finds the most units in stock for any product.

If you run this example against the sample Northwind database, the output is: `125`.

```
System.Nullable<Int16> maxUnitsInStock =
    (from prod in db.Products
    select prod.UnitsInStock)
    .Max();

Console.WriteLine(maxUnitsInStock);
```

```
Dim maxUnitsInStock = Aggregate prod In db.Products _
                      Into Max(prod.UnitsInStock)

Console.WriteLine(maxUnitsInStock)
```

Example

The following example uses Max to find the `Products` that have the highest unit price in each category. The output then lists the results by category.

```
var maxQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    select new
    {
        grouping.Key,
        MostExpensiveProducts =
            from prod2 in grouping
            where prod2.UnitPrice == grouping.Max(prod3 =>
                prod3.UnitPrice)
            select prod2
    };

foreach (var grp in maxQuery)
{
    Console.WriteLine(grp.Key);
    foreach (var listing in grp.MostExpensiveProducts)
    {
        Console.WriteLine(listing.ProductName);
    }
}
```

```
Dim maxQuery = From prod In db.Products() _
    Group prod By prod.CategoryID Into grouping = Group _
    Select CategoryID, _
    MostExpensiveProducts = _
        (From prod2 In grouping _
        Where prod2.UnitPrice = _
        grouping.Max(Function(prod3) prod3.UnitPrice))

For Each grp In maxQuery
    Console.WriteLine(grp.CategoryID)
    For Each listing In grp.MostExpensiveProducts
        Console.WriteLine(listing.ProductName)
    Next
Next
```

If you run the previous query against the Northwind sample database, your results will resemble the following:

`1`

`Côte de Blaye`

`2`

`Vegie-spread`

`3`

`Sir Rodney's Marmalade`

`4`

`Raclette Courdavault`

`5`

`Gnocchi di nonna Alice`

`6`

`Thüringer Rostbratwurst`

`7`

`Manjimup Dried Apples`

`8`

`Carnarvon Tigers`

See Also

Aggregate Queries
Downloading Sample Databases

Find the Minimum Value in a Numeric Sequence

5/1/2017 • 1 min to read • Edit Online

Use the Min operator to return the minimum value from a sequence of numeric values.

Example

The following example finds the lowest unit price of any product.

If you run this query against the Northwind sample database, the output is: `2.5000`.

```
System.Nullable<Decimal> lowestUnitPrice =
    (from prod in db.Products
    select prod.UnitPrice)
    .Min();

Console.WriteLine(lowestUnitPrice);
```

```
Dim lowestUnitPrice = Aggregate prod In db.Products _
                      Into Min(prod.UnitPrice)

Console.WriteLine(lowestUnitPrice)
```

Example

The following example finds the lowest freight amount for any order.

If you run this query against the Northwind sample database, the output is: `0.0200`.

```
System.Nullable<Decimal> lowestFreight =
    (from ord in db.Orders
    select ord.Freight)
    .Min();

Console.WriteLine(lowestFreight);
```

```
Dim lowestFreight = Aggregate ord In db.Orders _
                    Into Min(ord.Freight)

Console.WriteLine(lowestFreight)
```

Example

The following example uses Min to find the `Products` that have the lowest unit price in each category. The output is arranged by category.

```
var minQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    select new
    {
        grouping.Key,
        LeastExpensiveProducts =
            from prod2 in grouping
            where prod2.UnitPrice == grouping.Min(prod3 =>
            prod3.UnitPrice)
            select prod2
    };

foreach (var grp in minQuery)
{
    Console.WriteLine(grp.Key);
    foreach (var listing in grp.LeastExpensiveProducts)
    {
        Console.WriteLine(listing.ProductName);
    }
}
```

```
Dim minQuery = From prod In db.Products() _
    Group prod By prod.CategoryID Into grouping = Group _
    Select CategoryID, LeastExpensiveProducts = _
        From prod2 In grouping _
        Where prod2.UnitPrice = grouping.Min(Function(prod3) _
        prod3.UnitPrice)

For Each grp In minQuery
    Console.WriteLine(grp.CategoryID)
    For Each listing In grp.LeastExpensiveProducts
        Console.WriteLine(listing.ProductName)
    Next
Next
```

If you run the previous query against the Northwind sample database, your results will resemble the following:

`1`

`Guaraná Fantástica`

`2`

`Aniseed Syrup`

`3`

`Teatime Chocolate Biscuits`

`4`

`Geitost`

`5`

`Filo Mix`

`6`

`Tourtière`

`7`

`Longlife Tofu`

`8`

`Konbu`

See Also

Aggregate Queries
Downloading Sample Databases

Compute the Sum of Values in a Numeric Sequence

5/1/2017 • 1 min to read • Edit Online

Use the Sum operator to compute the sum of numeric values in a sequence.

Note the following characteristics of the `Sum` operator in LINQ to SQL:

- The Standard Query Operator aggregate operator `Sum` evaluates to zero for an empty sequence or a sequence that contains only nulls. In LINQ to SQL, the semantics of SQL are left unchanged. For this reason, `Sum` evaluates to null instead of to zero for an empty sequence or for a sequence that contains only nulls.
- SQL limitations on intermediate results apply to aggregates in LINQ to SQL. Sum of 32-bit integer quantities is not computed by using 64-bit results, and overflow can occur for the LINQ to SQL translation of `Sum`. This possibility exists even if the Standard Query Operator implementation does not cause an overflow for the corresponding in-memory sequence.

Example

The following example finds the total freight of all orders in the `Order` table.

If you run this query against the Northwind sample database, the output is: `64942.6900`.

```
System.Nullable<Decimal> totalFreight =
    (from ord in db.Orders
    select ord.Freight)
    .Sum();

Console.WriteLine(totalFreight);
```

```
Dim totalFreight = Aggregate ord In db.Orders _
                   Into Sum(ord.Freight)

Console.WriteLine(totalFreight)
```

Example

The following example finds the total number of units on order for all products.

If you run this query against the Northwind sample database, the output is: `780`.

Note that you must cast `short` types (for example, `UnitsOnOrder`) because `Sum` has no overload for short types.

```
System.Nullable<long> totalUnitsOnOrder =
    (from prod in db.Products
    select (long)prod.UnitsOnOrder)
    .Sum();

Console.WriteLine(totalUnitsOnOrder);
```

```
Dim totalUnitsOnOrder = Aggregate prod In db.Products _
                        Into Sum(prod.UnitsOnOrder)

Console.WriteLine(totalUnitsOnOrder)
```

See Also

Aggregate Queries
Downloading Sample Databases

Return the First Element in a Sequence

5/1/2017 • 1 min to read • Edit Online

Use the First operator to return the first element in a sequence. Queries that use First are executed immediately.

NOTE

LINQ to SQL does not support the Last operator.

Example

The following code finds the first `Shipper` in a table:

If you run this query against the Northwind sample database, the results are

`ID = 1, Company = Speedy Express`.

```
Shipper shipper = db.Shippers.First();
Console.WriteLine("ID = {0}, Company = {1}", shipper.ShipperID,
    shipper.CompanyName);
```

```
Dim shipper As Shipper = db.Shippers.First()
Console.WriteLine("ID = {0}, Company = {1}", shipper.ShipperID, _
        shipper.CompanyName)
```

Example

The following code finds the single `Customer` that has the `CustomerID` BONAP.

If you run this query against the Northwind sample database, the results are
`ID = BONAP, Contact = Laurence Lebihan`.

```
Customer custQuery =
    (from custs in db.Customers
    where custs.CustomerID == "BONAP"
    select custs)
    .First();

Console.WriteLine("ID = {0}, Contact = {1}", custQuery.CustomerID,
    custQuery.ContactName);
```

```
Dim custquery As Customer = _
    (From c In db.Customers _
    Where c.CustomerID = "BONAP" _
    Select c) _
    .First()

Console.WriteLine("ID = {0}, Contact = {1}", custquery.CustomerID, _
    custquery.ContactName)
```

See Also

Query Examples

Downloading Sample Databases

Return Or Skip Elements in a Sequence

5/1/2017 • 2 min to read • Edit Online

Use the Take operator to return a given number of elements in a sequence and then skip over the remainder.

Use the Skip operator to skip over a given number of elements in a sequence and then return the remainder.

NOTE
Take and Skip have certain limitations when they are used in queries against SQL Server 2000. For more information, see the "Skip and Take Exceptions in SQL Server 2000" entry in Troubleshooting.

LINQ to SQL translates Skip by using a subquery with the SQL `NOT EXISTS` clause. This translation has the following limitations:

- The argument must be a set. Multisets are not supported, even if ordered.
- The generated query can be much more complex than the query generated for the base query on which Skip is applied. This complexity can cause decrease in performance or even a time-out.

Example

The following example uses `Take` to select the first five `Employees` hired. Note that the collection is first sorted by `HireDate`.

```
IQueryable<Employee> firstHiredQuery =
    (from emp in db.Employees
    orderby emp.HireDate
    select emp)
    .Take(5);

foreach (Employee empObj in firstHiredQuery)
{
    Console.WriteLine("{0}, {1}", empObj.EmployeeID,
        empObj.HireDate);
}
```

```
Dim firstHiredQuery = _
    From emp In db.Employees _
    Select emp _
    Order By emp.HireDate _
    Take 5

For Each empObj As Employee In firstHiredQuery
    Console.WriteLine("{0}, {1}", empObj.EmployeeID, _
        empObj.HireDate)
Next
```

Example

The following example uses Skip to select all except the 10 most expensive `Products`.

```
IQueryable<Product> lessExpensiveQuery =
    (from prod in db.Products
    orderby prod.UnitPrice descending
    select prod)
    .Skip(10);

foreach (Product prodObj in lessExpensiveQuery)
{
    Console.WriteLine(prodObj.ProductName);
}
```

```
Dim lessExpensiveQuery = _
    From prod In db.Products _
    Select prod _
    Order By prod.UnitPrice Descending _
    Skip 10

For Each prodObj As Product In lessExpensiveQuery
    Console.WriteLine(prodObj.ProductName)
Next
```

Example

The following example combines the Skip and Take methods to skip the first 50 records and then return the next 10.

```
var custQuery2 =
    (from cust in db.Customers
    orderby cust.ContactName
    select cust)
    .Skip(50).Take(10);

foreach (var custRecord in custQuery2)
{
    Console.WriteLine(custRecord.ContactName);
}
```

```
Dim custQuery2 = _
    From cust In db.Customers _
    Order By (cust.ContactName) _
    Select cust _
    Skip 50 _
    Take 10

For Each custRecord As Customer In custQuery2
    Console.WriteLine(custRecord.ContactName)
Next
```

Take and Skip operations are well defined only against ordered sets. The semantics for unordered sets or multisets is undefined.

Because of the limitations on ordering in SQL, LINQ to SQL tries to move the ordering of the argument of the Take or Skip operator to the result of the operator.

NOTE

Translation is different for SQL Server 2000 and SQL Server 2005. If you plan to use Skip with a query of any complexity, use SQL Server 2005.

Consider the following LINQ to SQL query for SQL Server 2000:

```
IQueryable<Customer> custQuery3 =
    (from custs in db.Customers
     where custs.City == "London"
     orderby custs.CustomerID
     select custs)
    .Skip(1).Take(1);

foreach (var custObj in custQuery3)
{
    Console.WriteLine(custObj.CustomerID);
}
```

```
Dim custQuery3 = _
    From custs In db.Customers _
    Where custs.City = "London" _
    Select custs _
    Order By custs.CustomerID _
    Skip 1 _
    Take 1

For Each custObj In custQuery3
    Console.WriteLine(custObj.CustomerID)
Next
```

LINQ to SQL moves the ordering to the end in the SQL code, as follows:

```
SELECT TOP 1 [t0].[CustomerID], [t0].[CompanyName],
FROM [Customers] AS [t0]
WHERE (NOT (EXISTS(
    SELECT NULL AS [EMPTY]
    FROM (
        SELECT TOP 1 [t1].[CustomerID]
        FROM [Customers] AS [t1]
        WHERE [t1].[City] = @p0
        ORDER BY [t1].[CustomerID]
        ) AS [t2]
    WHERE [t0].[CustomerID] = [t2].[CustomerID]
    ))) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
```

When Take and Skip are chained together, all the specified ordering must be consistent. Otherwise, the results are undefined.

For non-negative, constant integral arguments based on the SQL specification, both Take and Skip are well-defined.

See Also

Query Examples
Standard Query Operator Translation

Sort Elements in a Sequence

5/1/2017 • 3 min to read • Edit Online

Use the OrderBy operator to sort a sequence according to one or more keys.

NOTE

LINQ to SQL is designed to support ordering by simple primitive types, such as `string`, `int`, and so on. It does not support ordering for complex multi-valued classes, such as anonymous types. It also does not support `byte` datatypes.

Example

The following example sorts `Employees` by date of hire.

```
IOrderedQueryable<Employee> hireQuery =
    from emp in db.Employees
    orderby emp.HireDate
    select emp;

foreach (Employee empObj in hireQuery)
{
    Console.WriteLine("EmpID = {0}, Date Hired = {1}",
        empObj.EmployeeID, empObj.HireDate);
}
```

```
Dim hireQuery = _
    From emp In db.Employees _
    Select emp _
    Order By emp.HireDate

For Each empObj As Employee In hireQuery
    Console.WriteLine("EmpID = {0}, Date Hired = {1}", _
        empObj.EmployeeID, empObj.HireDate)
Next
```

Example

The following example uses `where` to sort `Orders` shipped to `London` by freight.

```
IOrderedQueryable<Order> freightQuery =
    from ord in db.Orders
    where ord.ShipCity == "London"
    orderby ord.Freight
    select ord;

foreach (Order ordObj in freightQuery)
{
    Console.WriteLine("Order ID = {0}, Freight = {1}",
        ordObj.OrderID, ordObj.Freight);
}
```

```
Dim freightQuery = _
    From ord In db.Orders _
    Where ord.ShipCity = "London" _
    Select ord _
    Order By ord.Freight

For Each ordObj In freightQuery
    Console.WriteLine("Order ID = {0}, Freight = {1}", _
        ordObj.OrderID, ordObj.Freight)
Next
```

Example

The following example sorts `Products` by unit price from highest to lowest.

```
IOrderedQueryable<Product> priceQuery =
    from prod in db.Products
    orderby prod.UnitPrice descending
    select prod;

foreach (Product prodObj in priceQuery)
{
    Console.WriteLine("Product ID = {0}, Unit Price = {1}",
        prodObj.ProductID, prodObj.UnitPrice);
}
```

```
Dim priceQuery = _
    From prod In db.Products _
    Select prod _
    Order By prod.UnitPrice Descending

For Each prodObj In priceQuery
    Console.WriteLine("Product ID = {0}, Unit Price = {1}", _
       prodObj.ProductID, prodObj.UnitPrice)
Next
```

Example

The following example uses a compound `OrderBy` to sort `Customers` by city and then by contact name.

```
IOrderedQueryable<Customer> custQuery =
    from cust in db.Customers
    orderby cust.City, cust.ContactName
    select cust;

foreach (Customer custObj in custQuery)
{
    Console.WriteLine("City = {0}, Name = {1}", custObj.City,
        custObj.ContactName);
}
```

```
Dim custQuery = _
    From cust In db.Customers _
    Select cust _
    Order By cust.City, cust.ContactName

For Each custObj In custQuery
    Console.WriteLine("City = {0}, Name = {1}", custObj.City, _
        custObj.ContactName)
Next
```

Example

The following example sorts Orders from `EmployeeID 1` by ship-to country, and then by highest to lowest freight.

```
IOrderedQueryable<Order> ordQuery =
    from ord in db.Orders
    where ord.EmployeeID == 1
    orderby ord.ShipCountry, ord.Freight descending
    select ord;

foreach (Order ordObj in ordQuery)
{
    Console.WriteLine("Country = {0}, Freight = {1}",
        ordObj.ShipCountry, ordObj.Freight);
}
```

```
Dim ordQuery = _
    From ord In db.Orders _
    Where CInt(ord.EmployeeID.Value) = 1 _
    Select ord _
    Order By ord.ShipCountry, ord.Freight Descending

For Each ordObj In ordQuery
    Console.WriteLine("Country = {0}, Freight = {1}", _
        ordObj.ShipCountry, ordObj.Freight)
Next
```

Example

The following example combines OrderBy, Max, and GroupBy operators to find the `Products` that have the highest unit price in each category, and then sorts the group by category id.

```
var highPriceQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    orderby grouping.Key
    select new
    {
        grouping.Key,
        MostExpensiveProducts =
            from prod2 in grouping
            where prod2.UnitPrice == grouping.Max(p3 => p3.UnitPrice)
            select prod2
    };

foreach (var prodObj in highPriceQuery)
{
    Console.WriteLine(prodObj.Key);
    foreach (var listing in prodObj.MostExpensiveProducts)
    {
        Console.WriteLine(listing.ProductName);
    }
}
```

```
Dim highPriceQuery = From prod In db.Products _
    Group prod By prod.CategoryID Into grouping = Group _
    Order By CategoryID _
    Select CategoryID, _
    MostExpensiveProducts = _
        From prod2 In grouping _
        Where prod2.UnitPrice = _
        grouping.Max(Function(p3) p3.UnitPrice)

For Each prodObj In highPriceQuery
    Console.WriteLine(prodObj.CategoryID)
    For Each listing In prodObj.MostExpensiveProducts
        Console.WriteLine(listing.ProductName)
    Next
Next
```

If you run the previous query against the Northwind sample database, the results will resemble the following:

`1`

`Côte de Blaye`

`2`

`Vegie-spread`

`3`

`Sir Rodney's Marmalade`

`4`

`Raclette Courdavault`

`5`

`Gnocchi di nonna Alice`

`6`

`Thüringer Rostbratwurst`

`7`

`Manjimup Dried Apples`

`8`

`Carnarvon Tigers`

See Also

Query Examples
Downloading Sample Databases

Group Elements in a Sequence

5/1/2017 • 4 min to read • Edit Online

The GroupBy operator groups the elements of a sequence. The following examples use the Northwind database.

NOTE

Null column values in GroupBy queries can sometimes throw an InvalidOperationException. For more information, see the "GroupBy InvalidOperationException" section of Troubleshooting.

Example

The following example partitions `Products` by `CategoryID`.

```
IQueryable<IGrouping<Int32?, Product>> prodQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    select grouping;

foreach (IGrouping<Int32?, Product> grp in prodQuery)
{
    Console.WriteLine("\nCategoryID Key = {0}:", grp.Key);
    foreach (Product listing in grp)
    {
        Console.WriteLine("\t{0}", listing.ProductName);
    }
}
```

```
Dim prodQuery = From prod In db.Products _
    Group prod By prod.CategoryID Into grouping = Group

For Each grp In prodQuery
    Console.WriteLine(vbNewLine & "CategoryID Key = {0}:", _
        grp.CategoryID)
    For Each listing In grp.grouping
        Console.WriteLine(vbTab & listing.ProductName)
    Next
Next
```

Example

The following example uses Max to find the maximum unit price for each `CategoryID`.

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new
    {
        g.Key,
        MaxPrice = g.Max(p => p.UnitPrice)
    };
```

```
Dim query = From p In db.Products _
    Group p By p.CategoryID Into g = Group _
    Select CategoryID, MaxPrice = g.Max(Function(p) p.UnitPrice)
```

Example

The following example uses Average to find the average `UnitPrice` for each `CategoryID`.

```
var q2 =
    from p in db.Products
    group p by p.CategoryID into g
    select new
    {
        g.Key,
        AveragePrice = g.Average(p => p.UnitPrice)
    };
```

```
Dim q2 = From p In db.Products _
    Group p By p.CategoryID Into g = Group _
    Select CategoryID, AveragePrice = g.Average(Function(p) _
        p.UnitPrice)
```

Example

The following example uses Sum to find the total `UnitPrice` for each `CategoryID`.

```
var priceQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    select new
    {
        grouping.Key,
        TotalPrice = grouping.Sum(p => p.UnitPrice)
    };

foreach (var grp in priceQuery)
{
    Console.WriteLine("Category = {0}, Total price = {1}",
        grp.Key, grp.TotalPrice);
}
```

```
Dim priceQuery = From prod In db.Products _
    Group prod By prod.CategoryID Into grouping = Group _
    Select CategoryID, TotalPrice = grouping.Sum(Function(p) _
        p.UnitPrice)

For Each grp In priceQuery
    Console.WriteLine("Category = {0}, Total price = {1}", _
        grp.CategoryID, grp.TotalPrice)
Next
```

Example

The following example uses Count to find the number of discontinued `Products` in each `CategoryID`.

```
var disconQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    select new
    {
        grouping.Key,
        NumProducts = grouping.Count(p => p.Discontinued)
    };

foreach (var prodObj in disconQuery)
{
    Console.WriteLine("CategoryID = {0}, Discontinued# = {1}",
        prodObj.Key, prodObj.NumProducts);
}
```

```
Dim disconQuery = From prod In db.Products _
    Group prod By prod.CategoryID Into grouping = Group _
    Select CategoryID, NumProducts = grouping.Count(Function(p) _
        p.Discontinued)

For Each prodObj In disconQuery
    Console.WriteLine("CategoryID = {0}, Discontinued# = {1}", _
        prodObj.CategoryID, prodObj.NumProducts)
Next
```

Example

The following example uses a following `where` clause to find all categories that have at least 10 products.

```
var prodCountQuery =
    from prod in db.Products
    group prod by prod.CategoryID into grouping
    where grouping.Count() >= 10
    select new
    {
        grouping.Key,
        ProductCount = grouping.Count()
    };

foreach (var prodCount in prodCountQuery)
{
    Console.WriteLine("CategoryID = {0}, Product count = {1}",
        prodCount.Key, prodCount.ProductCount);
}
```

```
Dim prodCountQuery = From prod In db.Products _
    Group prod By prod.CategoryID Into grouping = Group _
    Where grouping.Count >= 10 _
    Select CategoryID, ProductCount = grouping.Count

For Each prodCount In prodCountQuery
    Console.WriteLine("CategoryID = {0}, Product count = {1}", _
        prodCount.CategoryID, prodCount.ProductCount)
Next
```

Example

The following example groups products by `CategoryID` and `SupplierID`.

```
var prodQuery =
    from prod in db.Products
    group prod by new
    {
        prod.CategoryID,
        prod.SupplierID
    }
    into grouping
    select new { grouping.Key, grouping };

foreach (var grp in prodQuery)
{
    Console.WriteLine("\nCategoryID {0}, SupplierID {1}",
        grp.Key.CategoryID, grp.Key.SupplierID);
    foreach (var listing in grp.grouping)
    {
        Console.WriteLine("\t{0}", listing.ProductName);
    }
}
```

```
Dim prodQuery = From prod In db.Products _
    Group prod By Key = New With {prod.CategoryID, prod.SupplierID} _
        Into grouping = Group

For Each grp In prodQuery
    Console.WriteLine(vbNewLine & "CategoryID {0}, SupplierID {1}", _
        grp.Key.CategoryID, grp.Key.SupplierID)
    For Each listing In grp.grouping
        Console.WriteLine(vbTab & listing.ProductName)
    Next
Next
```

Example

The following example returns two sequences of products. The first sequence contains products with unit price less than or equal to 10. The second sequence contains products with unit price greater than 10.

```
var priceQuery =
    from prod in db.Products
    group prod by new
    {
        Criterion = prod.UnitPrice > 10
    }
    into grouping
    select grouping;

foreach (var prodObj in priceQuery)
{
    if (prodObj.Key.Criterion == false)
        Console.WriteLine("Prices 10 or less:");
    else
        Console.WriteLine("\nPrices greater than 10");
    foreach (var listing in prodObj)
    {
        Console.WriteLine("{0}, {1}", listing.ProductName,
            listing.UnitPrice);
    }
}
```

```
Dim priceQuery = From prod In db.Products _
    Group prod By Key = New With {.Criterion = prod.UnitPrice > 10} _
        Into grouping = Group Select Key, grouping

For Each prodObj In priceQuery
    If prodObj.Key.Criterion = False Then
        Console.WriteLine("Prices 10 or less:")
    Else
        Console.WriteLine("\nPrices greater than 10")
        For Each listing In prodObj.grouping
            Console.WriteLine("{0}, {1}", listing.ProductName, _
                listing.UnitPrice)
        Next
    End If
Next
```

Example

The GroupBy operator can take only a single key argument. If you need to group by more than one key, you must create an anonymous type, as in the following example:

```
var custRegionQuery =
    from cust in db.Customers
    group cust.ContactName by new { City = cust.City, Region = cust.Region };

foreach (var grp in custRegionQuery)
{
    Console.WriteLine("\nLocation Key: {0}", grp.Key);
    foreach (var listing in grp)
    {
        Console.WriteLine("\t{0}", listing);
    }
}
```

```
Dim custRegionQuery = From cust In db.Customers _
    Group cust.ContactName By Key = New With _
        {cust.City, cust.Region} Into grouping = Group

For Each grp In custRegionQuery
    Console.WriteLine(vbNewLine & "Location Key: {0}", grp.Key)
    For Each listing In grp.grouping
        Console.WriteLine(vbTab & "{0}", listing)
    Next
Next
```

See Also

Query Examples
Downloading Sample Databases

Eliminate Duplicate Elements from a Sequence

5/1/2017 • 1 min to read • Edit Online

Use the Distinct operator to eliminate duplicate elements from a sequence.

Example

The following example uses Distinct to select a sequence of the unique cities that have customers.

```
IQueryable<String> cityQuery =
    (from cust in db.Customers
    select cust.City).Distinct();

foreach (String cityString in cityQuery)
{
    Console.WriteLine(cityString);
}
```

```
Dim cityQuery = _
    (From cust In db.Customers _
    Select cust.City).Distinct()

For Each cityString In cityQuery
    Console.WriteLine(cityString)
Next
```

See Also

Query Examples

Determine if Any or All Elements in a Sequence Satisfy a Condition

5/1/2017 • 1 min to read • Edit Online

The All operator returns `true` if all elements in a sequence satisfy a condition.

The Any operator returns `true` if any element in a sequence satisfies a condition.

Example

The following example returns a sequence of customers that have at least one order. The `Where` / `where` clause evaluates to `true` if the given `Customer` has any `Order`.

```
var OrdersQuery =
    from cust in db.Customers
    where cust.Orders.Any()
    select cust;
```

```
Dim OrdersQuery = _
    From cust In db.Customers _
    Where cust.Orders.Any() _
    Select cust
```

Example

The following Visual Basic code determines the list of customers who have not placed orders, and ensures that for every customer in that list, a contact name is provided.

```
Public Sub ContactsAvailable()
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim result = _
        (From cust In db.Customers _
        Where Not cust.Orders.Any() _
        Select cust).All(AddressOf ContactAvailable)

    If result Then
        Console.WriteLine _
    ("All of the customers who have made no orders have a contact name")
    Else
        Console.WriteLine _
    ("Some customers who have made no orders have no contact name")
    End If
End Sub

Function ContactAvailable(ByVal contact As Object) As Boolean
    Dim cust As Customer = CType(contact, Customer)
    Return (cust.ContactTitle Is Nothing OrElse _
        cust.ContactTitle.Trim().Length = 0)
End Function
```

Example

The following C# example returns a sequence of customers whose orders have a `ShipCity` beginning with "C". Also included in the return are customers who have no orders. (By design, the All operator returns `true` for an empty sequence.) Customers with no orders are eliminated in the console output by using the `Count` operator.

```
var custEmpQuery =
    from cust in db.Customers
    where cust.Orders.All(o => o.ShipCity.StartsWith("C"))
    orderby cust.CustomerID
    select cust;

foreach (Customer custObj in custEmpQuery)
{
    if (custObj.Orders.Count > 0)
        Console.WriteLine("CustomerID: {0}", custObj.CustomerID);
    foreach (Order ordObj in custObj.Orders)
    {
        Console.WriteLine("\t OrderID: {0}; ShipCity: {1}",
            ordObj.OrderID, ordObj.ShipCity);
    }
}
```

See Also

Query Examples

Concatenate Two Sequences

5/1/2017 • 1 min to read • Edit Online

Use the Concat operator to concatenate two sequences.

The Concat operator is defined for ordered multisets where the orders of the receiver and the argument are the same.

Ordering in SQL is the final step before results are produced. For this reason, the Concat operator is implemented by using `UNION ALL` and does not preserve the order of its arguments. To make sure ordering is correct in the results, make sure to explicitly order the results.

Example

This example uses Concat to return a sequence of all `Customer` and `Employee` telephone and fax numbers.

```
IQueryable<String> custQuery =
    (from cust in db.Customers
    select cust.Phone)
    .Concat
    (from cust in db.Customers
    select cust.Fax)
    .Concat
    (from emp in db.Employees
    select emp.HomePhone)
;

foreach (var custData in custQuery)
{
    Console.WriteLine(custData);
}
```

```
Dim custQuery = _
    (From c In db.Customers _
    Select c.Phone) _
    .Concat _
    (From c In db.Customers _
    Select c.Fax) _
    .Concat _
    (From e In db.Employees _
    Select e.HomePhone)

For Each custData In custQuery
    Console.WriteLine(custData)
Next
```

Example

This example uses Concat to return a sequence of all `Customer` and `Employee` name and telephone number mappings.

```
var infoQuery =
    (from cust in db.Customers
    select new { Name = cust.CompanyName, cust.Phone }
    )
   .Concat
       (from emp in db.Employees
       select new
       {
           Name = emp.FirstName + " " + emp.LastName,
           Phone = emp.HomePhone
       }
       );

foreach (var infoData in infoQuery)
{
    Console.WriteLine("Name = {0}, Phone = {1}",
        infoData.Name, infoData.Phone);
}
```

```
Dim infoQuery = _
    (From cust In db.Customers _
    Select Name = cust.CompanyName, Phone = cust.Phone) _
    .Concat _
        (From emp In db.Employees _
        Select Name = emp.FirstName & " " & emp.LastName, _
            Phone = emp.HomePhone)

For Each infoData In infoQuery
    Console.WriteLine("Name = " & infoData.Name & _
        ", Phone = " & infoData.Phone)
Next
```

See Also

Query Examples
Standard Query Operator Translation

Return the Set Difference Between Two Sequences

5/1/2017 • 1 min to read • Edit Online

Use the Except operator to return the set difference between two sequences.

Example

This example uses Except to return a sequence of all countries in which `Customers` live but in which no `Employees` live.

```
var infoQuery =
    (from cust in db.Customers
    select cust.Country)
    .Except
        (from emp in db.Employees
        select emp.Country)
;
```

```
Dim infoQuery = _
    (From cust In db.Customers _
    Select cust.Country) _
    .Except _
        (From emp In db.Employees _
        Select emp.Country)
```

In LINQ to SQL, the Except operation is well defined only on sets. The semantics for multisets is undefined.

See Also

Query Examples
Standard Query Operator Translation

Return the Set Intersection of Two Sequences

5/1/2017 • 1 min to read • Edit Online

Use the Intersect operator to return the set intersection of two sequences.

Example

This example uses Intersect to return a sequence of all countries in which both `Customers` and `Employees` live.

```
var infoQuery =
    (from cust in db.Customers
    select cust.Country)
    .Intersect
        (from emp in db.Employees
        select emp.Country)
;
```

```
Dim infoQuery = _
    (From cust In db.Customers _
    Select cust.Country) _
    .Intersect _
        (From emp In db.Employees _
        Select emp.Country)
```

In LINQ to SQL, the Intersect operation is well defined only on sets. The semantics for multisets is undefined.

See Also

Query Examples
Standard Query Operator Translation

Return the Set Union of Two Sequences

5/1/2017 • 1 min to read • Edit Online

Use the Union operator to return the set union of two sequences.

Example

This example uses Union to return a sequence of all countries in which there are either `Customers` or `Employees`.

```
var infoQuery =
    (from cust in db.Customers
    select cust.Country)
    .Union
        (from emp in db.Employees
        select emp.Country)
;
```

```
Dim infoQuery = _
    (From cust In db.Customers _
    Select cust.Country) _
    .Union _
        (From emp In db.Employees _
        Select emp.Country)
```

In LINQ to SQL, the Union operator is defined for multisets as the unordered concatenation of the multisets (effectively the result of the `UNION ALL` clause in SQL).

See Also

Query Examples
Standard Query Operator Translation

Convert a Sequence to an Array

5/1/2017 • 1 min to read • Edit Online

Use ToArray to create an array from a sequence.

Example

The following example uses ToArray to immediately evaluate a query into an array and to get the third element.

```
var custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
    Customer[] qArray = custQuery.ToArray();
```

```
Dim custQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust
    Dim qArray() As Customer = custQuery.ToArray()
```

See Also

Query Examples

Convert a Sequence to a Generic List

5/1/2017 • 1 min to read • Edit Online

Use ToList to create a generic List from a sequence.

Example

The following sample uses ToList to immediately evaluate a query into a generic List<T>.

```
var empQuery =
    from emp in db.Employees
    where emp.HireDate >= new DateTime(1994, 1, 1)
    select emp;
    List<Employee> qList = empQuery.ToList();
```

```
Dim empQuery = _
    From emp In db.Employees _
    Where emp.HireDate.Value >= #1/1/1994# _
    Select emp
Dim qList As List(Of Employee) = empQuery.ToList()
```

See Also

Query Examples

Convert a Type to a Generic IEnumerable

5/1/2017 • 1 min to read • Edit Online

Use AsEnumerable to return the argument typed as a generic `IEnumerable`.

Example

In this example, LINQ to SQL (using the default generic `Query`) would try to convert the query to SQL and execute it on the server. But the `where` clause references a user-defined client-side method (`isValidProduct`), which cannot be converted to SQL.

The solution is to specify the client-side generic IEnumerable<T> implementation of `where` to replace the generic IQueryable<T>. You do this by invoking the AsEnumerable operator.

```
private bool isValidProduct(Product prod)
{
    return prod.ProductName.LastIndexOf('C') == 0;
}

void ConvertToIEnumerable()
{
    Northwnd db = new Northwnd(@"c:\test\northwnd.mdf");
    Program pg = new Program();
    var prodQuery =
        from prod in db.Products.AsEnumerable()
        where isValidProduct(prod)
        select prod;
}
```

```
Private Function isValidProduct(ByVal prod As Product) As Boolean
    Return prod.ProductName.LastIndexOf("C") = 0
End Function

Sub ConvertToIEnumerable()
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim validProdQuery = _
        From prod In db.Products.AsEnumerable _
        Where isValidProduct(prod) _
        Select prod
End Sub
```

See Also

Query Examples

Formulate Joins and Cross-Product Queries

5/1/2017 • 4 min to read • Edit Online

The following examples show how to combine results from multiple tables.

Example

The following example uses foreign key navigation in the `From` clause in Visual Basic (`from` clause in C#) to select all orders for customers in London.

```
var infoQuery =
    from cust in db.Customers
    from ord in cust.Orders
    where cust.City == "London"
    select ord;
```

```
Dim infoQuery = _
From cust In db.Customers, ord In cust.Orders _
Where cust.City = "London" _
Select ord
```

Example

The following example uses foreign key navigation in the `Where` clause in Visual Basic (`where` clause in C#) to filter for out-of-stock `Products` whose `Supplier` is in the United States.

```
var infoQuery =
    from prod in db.Products
    where prod.Supplier.Country == "USA" && prod.UnitsInStock == 0
    select prod;
```

```
Dim infoQuery = _
    From prod In db.Products _
    Where prod.Supplier.Country = "USA" AndAlso _
        CShort(prod.UnitsInStock) = 0 _
    Select prod
```

Example

The following example uses foreign key navigation in the `From` clause in Visual Basic (`from` clause in C#) to filter for employees in Seattle and to list their territories.

```
var infoQuery =
    from emp in db.Employees
    from empterr in emp.EmployeeTerritories
    where emp.City == "Seattle"
    select new
    {
        emp.FirstName,
        emp.LastName,
        empterr.Territory.TerritoryDescription
    };
```

Example

The following example uses foreign key navigation in the `Select` clause in Visual Basic (`select` clause in C#) to filter for pairs of employees where one employee reports to the other and where both employees are from the same `City`.

```
var infoQuery =
    from emp1 in db.Employees
    from emp2 in emp1.Employees
    where emp1.City == emp2.City
    select new
    {
        FirstName1 = emp1.FirstName,
        LastName1 = emp1.LastName,
        FirstName2 = emp2.FirstName,
        LastName2 = emp2.LastName,
        emp1.City
    };
```

```
Dim infoQuery = _
    From e1 In db.Employees, e2 In e1.Employees _
    Where e1.City = e2.City _
    Select FirstName1 = e1.FirstName, _
        LastName1 = e1.LastName, FirstName2 = e2.FirstName, _
        LastName2 = e2.LastName, e1.City
```

Example

The following Visual Basic example looks for all customers and orders, makes sure that the orders are matched to customers, and guarantees that for every customer in that list, a contact name is provided.

```
Dim q1 = From c In db.Customers, o In db.Orders _
    Where c.CustomerID = o.CustomerID _
    Select c.CompanyName, o.ShipRegion

' Note that because the O/R designer generates class
' hierarchies for database relationships for you,
' the following code has the same effect as the above
' and is shorter:

Dim q2 = From c In db.Customers, o In c.Orders _
    Select c.CompanyName, o.ShipRegion

For Each nextItem In q2
    Console.WriteLine("{0}   {1}", nextItem.CompanyName, _
        nextItem.ShipRegion)
Next
```

Example

The following example explicitly joins two tables and projects results from both tables.

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID equals o.CustomerID
        into orders
    select new { c.ContactName, OrderCount = orders.Count() };
```

```
Dim q = From c In db.Customers _
    Group Join o In db.Orders On c.CustomerID Equals o.CustomerID _
        Into orders = Group _
    Select c.ContactName, OrderCount = orders.Count()
```

Example

The following example explicitly joins three tables and projects results from each of them.

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID equals o.CustomerID
        into ords
    join e in db.Employees on c.City equals e.City into emps
    select new
    {
        c.ContactName,
        ords = ords.Count(),
        emps = emps.Count()
    };
```

```
Dim q = From c In db.Customers _
    Group Join o In db.Orders On c.CustomerID Equals o.CustomerID _
        Into ords = Group _
        Group Join e In db.Employees On c.City Equals e.City _
            Into emps = Group _
    Select c.ContactName, ords = ords.Count(), emps = emps.Count()
```

Example

The following example shows how to achieve a `LEFT OUTER JOIN` by using `DefaultIfEmpty()`. The `DefaultIfEmpty()` method returns null when there is no `Order` for the `Employee`.

```
var q =
    from e in db.Employees
    join o in db.Orders on e equals o.Employee into ords
        from o in ords.DefaultIfEmpty()
        select new { e.FirstName, e.LastName, Order = o };
```

```
Dim q = From e In db.Employees() _
    Group Join o In db.Orders On e Equals o.Employee Into ords _
        = Group _
    From o In ords.DefaultIfEmpty() _
    Select e.FirstName, e.LastName, Order = o
```

Example

The following example projects a `let` expression resulting from a join.

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID equals o.CustomerID
        into ords
    let z = c.City + c.Country
        from o in ords
        select new { c.ContactName, o.OrderID, z };
```

```
Dim q = From c In db.Customers _
    Group Join o In db.Orders On c.CustomerID Equals o.CustomerID _
        Into ords = Group _
    Let z = c.City + c.Country _
        From o In ords _
        Select c.ContactName, o.OrderID, z
```

Example

The following example shows a `join` with a composite key.

```
var q =
    from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new { o.OrderID, p.ProductID } equals new
    {
        d.OrderID,
        d.ProductID
    } into details
        from d in details
        select new { o.OrderID, p.ProductID, d.UnitPrice };
```

```
Dim q = From o In db.Orders _
    From p In db.Products _
    Group Join d In db.OrderDetails On New With {o.OrderID, _
        p.ProductID} _
        Equals New With {d.OrderID, d.ProductID} Into details _
            = Group _
        From d In details _
    Select o.OrderID, p.ProductID, d.UnitPrice
```

Example

The following example shows how to construct a `join` where one side is nullable and the other is not.

```
var q =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID into emps
        from e in emps
        select new { o.OrderID, e.FirstName };
```

```
Dim q = From o In db.Orders _
    Group Join e In db.Employees On o.EmployeeID _
        Equals e.EmployeeID Into emps = Group _
        From e In emps _
    Select o.OrderID, e.FirstName
```

See Also

Query Examples

Formulate Projections

5/1/2017 • 3 min to read • Edit Online

The following examples show how the `select` statement in C# and `Select` statement in Visual Basic can be combined with other features to form query projections.

Example

The following example uses the `Select` clause in Visual Basic (`select` clause in C#) to return a sequence of contact names for `Customers`.

```
var nameQuery =
    from cust in db.Customers
    select cust.ContactName;
```

```
Dim nameQuery = From cust In db.Customers _
    Select cust.ContactName
```

Example

The following example uses the `Select` clause in Visual Basic (`select` clause in C#) and *anonymous types* to return a sequence of contact names and telephone numbers for `Customers`.

```
var infoQuery =
    from cust in db.Customers
    select new { cust.ContactName, cust.Phone };
```

```
Dim infoQuery = From cust In db.Customers _
    Select cust.ContactName, cust.Phone
```

Example

The following example uses the `Select` clause in Visual Basic (`select` clause in C#) and *anonymous types* to return a sequence of names and telephone numbers for employees. The `FirstName` and `LastName` fields are combined into a single field (`Name`), and the `HomePhone` field is renamed to `Phone` in the resulting sequence.

```
var info2Query =
    from emp in db.Employees
    select new
    {
        Name = emp.FirstName + " " + emp.LastName,
        Phone = emp.HomePhone
    };
```

```
Dim info2Query = From emp In db.Employees _
    Select Name = emp.FirstName & " " & emp.LastName, _
Phone = emp.HomePhone
```

Example

The following example uses the `Select` clause in Visual Basic (`select` clause in C#) and *anonymous types* to return a sequence of all `ProductID`s and a calculated value named `HalfPrice`. This value is set to the `UnitPrice` divided by 2.

```
var specialQuery =
    from prod in db.Products
    select new { prod.ProductID, HalfPrice = prod.UnitPrice / 2 };
```

```
Dim specialQuery = From prod In db.Products _
    Select prod.ProductID, HalfPrice = CDec(prod.UnitPrice) / 2
```

Example

The following example uses the `Select` clause in Visual Basic (`select` clause in C#) and a *conditional statement* to return a sequence of product name and product availability.

```
var prodQuery =
    from prod in db.Products
    select new
    {
        prod.ProductName,
        Availability =
            prod.UnitsInStock - prod.UnitsOnOrder < 0
        ? "Out Of Stock" : "In Stock"
    };
```

```
Dim prodQuery = From prod In db.Products _
Select prod.ProductName, Availability = _
    If(prod.UnitsInStock - prod.UnitsOnOrder < 0, _
    "Out Of Stock", "In Stock")
```

Example

The following example uses a Visual Basic `Select` clause (`select` clause in C#) and a *known type* (Name) to return a sequence of the names of employees.

```
public class Name
{
    public string FirstName = "";
    public string LastName = "";
}

 void empMethod()
 {
 Northwnd db = new Northwnd(@"c:\northwnd.mdf");
 var empQuery =
     from emp in db.Employees
     select new Name
     {
         FirstName = emp.FirstName,
         LastName = emp.LastName
     };
}
```

```
Public Class Name
    Public FirstName As String
    Public LastName As String
End Class

Dim db As New Northwnd("c:\northwnd.mdf")
Dim empQuery = From emp In db.Employees _
    Select New Name With {.FirstName = emp.FirstName, .LastName = _
        emp.LastName}
```

Example

The following example uses `Select` and `Where` in Visual Basic (`select` and `where` in C#) to return a *filtered sequence* of contact names for customers in London.

```
var contactQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust.ContactName;
```

```
Dim contactQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust.ContactName
```

Example

The following example uses a `Select` clause in Visual Basic (`select` clause in C#) and *anonymous types* to return a *shaped subset* of the data about customers.

```
var custQuery =
    from cust in db.Customers
    select new
    {
        cust.CustomerID,
        CompanyInfo = new { cust.CompanyName, cust.City, cust.Country },
        ContactInfo = new { cust.ContactName, cust.ContactTitle }
    };
```

```
Dim custQuery = From cust In db.Customers _
    Select cust.CustomerID, CompanyInfo = New With {cust.CompanyName, _
        cust.City, cust.Country}, ContactInfo = _
        New With {cust.ContactName, cust.ContactTitle}
```

Example

The following example uses nested queries to return the following results:

- A sequence of all orders and their corresponding `OrderID`s.
- A subsequence of the items in the order for which there is a discount.
- The amount of money saved if the cost of shipping is not included.

```
var ordQuery =
    from ord in db.Orders
    select new
    {
        ord.OrderID,
        DiscountedProducts =
            from od in ord.OrderDetails
            where od.Discount > 0.0
            select od,
        FreeShippingDiscount = ord.Freight
    };
```

```
Dim ordQuery = From ord In db.Orders _
    Select ord.OrderID, DiscountedProducts = _
    (From od In ord.OrderDetails _
        Where od.Discount > 0.0 _
    Select od), _
FreeShippingDiscount = ord.Freight
```

See Also

Query Examples

Making and Submitting Data Changes

5/1/2017 • 1 min to read • Edit Online

The topics in this section describe how to make and transmit changes to the database and how to handle optimistic concurrency conflicts.

NOTE

You can override LINQ to SQL default methods for `Insert`, `Update`, and `Delete` database operations. For more information, see Customizing Insert, Update, and Delete Operations.

Developers using Visual Studio can use the Object Relational Designer to develop stored procedures for the same purpose.

In This Section

How to: Insert Rows Into the Database
Describes how to insert rows in the database by adding objects to the object model.

How to: Update Rows in the Database
Describes how to update rows in the database by updating objects in the object model.

How to: Delete Rows From the Database
Describes how to delete rows in the database by deleting objects in the object model.

How to: Submit Changes to the Database
Describes how to send object-model changes to the database.

How to: Bracket Data Submissions by Using Transactions
Describes how to include operations in a transaction.

How to: Dynamically Create a Database
Describes how to generate databases dynamically, and typical scenarios for this approach.

How to: Manage Change Conflicts
Describes techniques for addressing optimistic concurrency issues.

How to: Insert Rows Into the Database

5/10/2017 • 1 min to read • Edit Online

You insert rows into a database by adding objects to the associated LINQ to SQL Table<TEntity> collection and then submitting the changes to the database. LINQ to SQL translates your changes into the appropriate SQL `INSERT` commands.

NOTE

You can override LINQ to SQL default methods for `Insert`, `Update`, and `Delete` database operations. For more information, see Customizing Insert, Update, and Delete Operations.

Developers using Visual Studio can use the Object Relational Designer to develop stored procedures for the same purpose.

The following steps assume that a valid DataContext connects you to the Northwind database. For more information, see How to: Connect to a Database.

To insert a row into the database

1. Create a new object that includes the column data to be submitted.
2. Add the new object to the LINQ to SQL `Table` collection associated with the target table in the database.
3. Submit the change to the database.

Example

The following code example creates a new object of type `Order` and populates it with appropriate values. It then adds the new object to the `Order` collection. Finally, it submits the change to the database as a new row in the `Orders` table.

```
// Create a new Order object.
Order ord = new Order
{
    OrderID = 12000,
    ShipCity = "Seattle",
    OrderDate = DateTime.Now
    // …
};

// Add the new object to the Orders collection.
db.Orders.InsertOnSubmit(ord);

// Submit the change to the database.
try
{
    db.SubmitChanges();
}
catch (Exception e)
{
    Console.WriteLine(e);
    // Make some adjustments.
    // ...
    // Try again.
    db.SubmitChanges();
}
```

```
' Create a new Order object.
Dim ord As New Order With _
{.OrderID = 12000, _
 .ShipCity = "Seattle", _
 .OrderDate = DateTime.Now}

' Add the new object to the Orders collection.
db.Orders.InsertOnSubmit(ord)

' Submit the change to the database.
Try
    db.SubmitChanges()
Catch e As Exception
    Console.WriteLine(e)
    ' Make some adjustments.
    ' ...
    ' Try again.
    db.SubmitChanges()
End Try
```

See Also

How to: Manage Change Conflicts
DataContext Methods (O/R Designer)
How to: Assign stored procedures to perform updates, inserts, and deletes (O/R Designer)
Making and Submitting Data Changes

How to: Update Rows in the Database

5/1/2017 • 1 min to read • Edit Online

You can update rows in a database by modifying member values of the objects associated with the LINQ to SQL Table<TEntity> collection and then submitting the changes to the database. LINQ to SQL translates your changes into the appropriate SQL `UPDATE` commands.

NOTE

You can override LINQ to SQL default methods for `Insert`, `Update`, and `Delete` database operations. For more information, see Customizing Insert, Update, and Delete Operations.

Developers using Visual Studio can use the Object Relational Designer to develop stored procedures for the same purpose.

The following steps assume that a valid DataContext connects you to the Northwind database. For more information, see How to: Connect to a Database.

To update a row in the database

1. Query the database for the row to be updated.
2. Make desired changes to member values in the resulting LINQ to SQL object.
3. Submit the changes to the database.

Example

The following example queries the database for order #11000, and then changes the values of `ShipName` and `ShipVia` in the resulting `Order` object. Finally, the changes to these member values are submitted to the database as changes in the `ShipName` and `ShipVia` columns.

```
// Query the database for the row to be updated.
var query =
    from ord in db.Orders
    where ord.OrderID == 11000
    select ord;

// Execute the query, and change the column values
// you want to change.
foreach (Order ord in query)
{
    ord.ShipName = "Mariner";
    ord.ShipVia = 2;
    // Insert any additional changes to column values.
}

// Submit the changes to the database.
try
{
    db.SubmitChanges();
}
catch (Exception e)
{
    Console.WriteLine(e);
    // Provide for exceptions.
}
```

```
' Query the database for the row to be updated.
Dim ordQuery = _
    From ord In db.Orders _
    Where ord.OrderID = 11000 _
    Select ord

' Execute the query, and change the column values
' you want to change.
For Each ord As Order In ordQuery
    ord.ShipName = "Mariner"
    ord.ShipVia = 2
    ' Insert any additional changes to column values.
Next

' Submit the changes to the database.
Try
    db.SubmitChanges()
Catch e As Exception
    Console.WriteLine(e)
    ' Make some adjustments.
    ' ...
    ' Try again
    db.SubmitChanges()
End Try
```

See Also

How to: Manage Change Conflicts
How to: Assign stored procedures to perform updates, inserts, and deletes (O/R Designer)
Making and Submitting Data Changes

How to: Delete Rows From the Database

5/1/2017 • 3 min to read • Edit Online

You can delete rows in a database by removing the corresponding LINQ to SQL objects from their table-related collection. LINQ to SQL translates your changes to the appropriate SQL `DELETE` commands.

LINQ to SQL does not support or recognize cascade-delete operations. If you want to delete a row in a table that has constraints against it, you must complete either of the following tasks:

- Set the `ON DELETE CASCADE` rule in the foreign-key constraint in the database.
- Use your own code to first delete the child objects that prevent the parent object from being deleted.

Otherwise, an exception is thrown. See the second code example later in this topic.

NOTE

You can override LINQ to SQL default methods for `Insert`, `Update`, and `Delete` database operations. For more information, see Customizing Insert, Update, and Delete Operations.

Developers using Visual Studio can use the Object Relational Designer to develop stored procedures for the same purpose.

The following steps assume that a valid DataContext connects you to the Northwind database. For more information, see How to: Connect to a Database.

To delete a row in the database

1. Query the database for the row to be deleted.
2. Call the DeleteOnSubmit method.
3. Submit the change to the database.

Example

This first code example queries the database for order details that belong to Order #11000, marks these order details for deletion, and submits these changes to the database.

```
// Query the database for the rows to be deleted.
var deleteOrderDetails =
    from details in db.OrderDetails
    where details.OrderID == 11000
    select details;

foreach (var detail in deleteOrderDetails)
{
    db.OrderDetails.DeleteOnSubmit(detail);
}

try
{
    db.SubmitChanges();
}
catch (Exception e)
{
    Console.WriteLine(e);
    // Provide for exceptions.
}
```

```
' Query the database for the rows to be deleted.
Dim deleteOrderDetails = _
    From details In db.OrderDetails() _
    Where details.OrderID = 11000 _
    Select details

For Each detail As OrderDetail In deleteOrderDetails
    db.OrderDetails.DeleteOnSubmit(detail)
Next

Try
    db.SubmitChanges()
Catch ex As Exception
    Console.WriteLine(ex)
    ' Provide for exceptions
End Try
```

Example

In this second example, the objective is to remove an order (#10250). The code first examines the `OrderDetails` table to see whether the order to be removed has children there. If the order has children, first the children and then the order are marked for removal. The DataContext puts the actual deletes in correct order so that delete commands sent to the database abide by the database constraints.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

db.Log = Console.Out;

// Specify order to be removed from database
int reqOrder = 10250;

// Fetch OrderDetails for requested order.
var ordDetailQuery =
    from odq in db.OrderDetails
    where odq.OrderID == reqOrder
    select odq;

foreach (var selectedDetail in ordDetailQuery)
{
    Console.WriteLine(selectedDetail.Product.ProductID);
    db.OrderDetails.DeleteOnSubmit(selectedDetail);
}

// Display progress.
Console.WriteLine("detail section finished.");
Console.ReadLine();

// Determine from Detail collection whether parent exists.
if (ordDetailQuery.Any())
{
    Console.WriteLine("The parent is presesnt in the Orders collection.");
    // Fetch Order.
    try
    {
        var ordFetch =
            (from ofetch in db.Orders
             where ofetch.OrderID == reqOrder
             select ofetch).First();
        db.Orders.DeleteOnSubmit(ordFetch);
        Console.WriteLine("{0} OrderID is marked for deletion.", ordFetch.OrderID);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
else
{
    Console.WriteLine("There was no parent in the Orders collection.");
}


// Display progress.
Console.WriteLine("Order section finished.");
Console.ReadLine();

try
{
    db.SubmitChanges();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    Console.ReadLine();
}

// Display progress.
Console.WriteLine("Submit finished.");
Console.ReadLine();
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

db.Log = Console.Out
' Specify order to be removed from database.
Dim reqOrder As Integer = 10252

' Fetch OrderDetails for requested order.
Dim ordDetailQuery = _
From odq In db.OrderDetails _
Where odq.OrderID = reqOrder _
Select odq

For Each selectedDetail As OrderDetail In ordDetailQuery
    Console.WriteLine(selectedDetail.Product.ProductID)
    db.OrderDetails.DeleteOnSubmit(selectedDetail)
Next

' Display progress.
Console.WriteLine("Detail section finished.")
Console.ReadLine()

' Determine from Detail collection whether parent exists.
If ordDetailQuery.Any Then
    Console.WriteLine("The parent is present in the Orders collection.")
    ' Fetch order.
    Try
        Dim ordFetch = _
        (From ofetch In db.Orders _
        Where ofetch.OrderID = reqOrder _
        Select ofetch).First()

        db.Orders.DeleteOnSubmit(ordFetch)
        Console.WriteLine("{0} OrderID is marked for deletion.,", ordFetch.OrderID)

    Catch ex As Exception
        Console.WriteLine(ex.Message)
        Console.ReadLine()
    End Try

Else
    Console.WriteLine("There was no parent in the Orders collection.")

End If


' Display progress.
Console.WriteLine("Order section finished.")
Console.ReadLine()

Try
    db.SubmitChanges()

Catch ex As Exception
    Console.WriteLine(ex.Message)
    Console.ReadLine()

End Try

' Display progress.
Console.WriteLine("Submit finished.")
Console.ReadLine()
```

See Also

How to: Manage Change Conflicts

How to: Assign stored procedures to perform updates, inserts, and deletes (O/R Designer)

Making and Submitting Data Changes

How to: Submit Changes to the Database

5/1/2017 • 1 min to read • Edit Online

Regardless of how many changes you make to your objects, changes are made only to in-memory replicas. You have made no changes to the actual data in the database. Your changes are not transmitted to the server until you explicitly call SubmitChanges on the DataContext.

When you make this call, the DataContext tries to translate your changes into equivalent SQL commands. You can use your own custom logic to override these actions, but the order of submission is orchestrated by a service of the DataContext known as the *change processor*. The sequence of events is as follows:

1. When you call SubmitChanges, LINQ to SQL examines the set of known objects to determine whether new instances have been attached to them. If they have, these new instances are added to the set of tracked objects.
2. All objects that have pending changes are ordered into a sequence of objects based on the dependencies between them. Objects whose changes depend on other objects are sequenced after their dependencies.
3. Immediately before any actual changes are transmitted, LINQ to SQL starts a transaction to encapsulate the series of individual commands.
4. The changes to the objects are translated one by one to SQL commands and sent to the server.

At this point, any errors detected by the database cause the submission process to stop, and an exception is raised. All changes to the database are rolled back as if no submissions ever occurred. The DataContext still has a full recording of all changes. You can therefore try to correct the problem and call SubmitChanges again, as in the code example that follows.

Example

When the transaction around the submission is completed successfully, the DataContext accepts the changes to the objects by ignoring the change-tracking information.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
// Make changes here.
try
{
    db.SubmitChanges();
}
catch (ChangeConflictException e)
{
    Console.WriteLine(e.Message);
    // Make some adjustments.
    // ...
    // Try again.
    db.SubmitChanges();
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

' Make changes here.
Sub MakeChanges()
    Try
        db.SubmitChanges()
    Catch e As ChangeConflictException
        Console.WriteLine(e.Message)
        ' Make some adjustments
        '...
        ' Try again.
        db.SubmitChanges()
    End Try
End Sub
```

See Also

How to: Detect and Resolve Conflicting Submissions
How to: Manage Change Conflicts
Downloading Sample Databases
Making and Submitting Data Changes

How to: Bracket Data Submissions by Using Transactions

5/1/2017 • 1 min to read • Edit Online

You can use TransactionScope to bracket your submissions to the database. For more information, see Transaction Support.

Example

The following code encloses the database submission in a TransactionScope.

```
        Northwnd db = new Northwnd(@"c:\northwnd.mdf");
        using (TransactionScope ts = new TransactionScope())
        {
            try
            {
                Product prod1 = db.Products.First(p => p.ProductID == 4);
                Product prod2 = db.Products.First(p => p.ProductID == 5);
                prod1.UnitsInStock -= 3;
                prod2.UnitsInStock -= 5;
                db.SubmitChanges();
ts.Complete();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
```

```
        Dim db As New Northwnd("c:\northwnd.mdf")
        Using ts = New TransactionScope()
            Try

                Dim prod1 = db.Products.First(Function(p) p.ProductID = 4)
                Dim prod2 = db.Products.First(Function(p) p.ProductID = 5)
                prod1.UnitsInStock -= 3
                prod2.UnitsInStock -= 5
                db.SubmitChanges()
ts.Complete()

            Catch e As Exception
                Console.WriteLine(e.Message)
            End Try
        End Using
```

See Also

Downloading Sample Databases
Making and Submitting Data Changes
Transaction Support

How to: Dynamically Create a Database

5/1/2017 • 2 min to read • Edit Online

In LINQ to SQL, an object model is mapped to a relational database. Mapping is enabled by using attribute-based mapping or an external mapping file to describe the structure of the relational database. In both scenarios, there is enough information about the relational database that you can create a new instance of the database using the System.Data.Linq.DataContext.CreateDatabase method.

The System.Data.Linq.DataContext.CreateDatabase method creates a replica of the database only to the extent of the information encoded in the object model. Mapping files and attributes from your object model might not encode everything about the structure of an existing database. Mapping information does not represent the contents of user-defined functions, stored procedures, triggers, or check constraints. This behavior is sufficient for a variety of databases.

You can use the System.Data.Linq.DataContext.CreateDatabase method in any number of scenarios, especially if a known data provider like Microsoft SQL Server 2008 is available. Typical scenarios include the following:

- You are building an application that automatically installs itself on a customer system.
- You are building a client application that needs a local database to save its offline state.

You can also use the System.Data.Linq.DataContext.CreateDatabase method with SQL Server by using an .mdf file or a catalog name, depending on your connection string. LINQ to SQL uses the connection string to define the database to be created and on which server the database is to be created.

NOTE

Whenever possible, use Windows Integrated Security to connect to the database so that passwords are not required in the connection string.

Example

The following code provides an example of how to create a new database named MyDVDs.mdf.

```
public class MyDVDs : DataContext
{
    public Table<DVD> DVDs;
    public MyDVDs(string connection) : base(connection) { }
}

[Table(Name = "DVDTable")]
public class DVD
{
    [Column(IsPrimaryKey = true)]
    public string Title;
    [Column]
    public string Rating;
}
```

```
Public Class MyDVDs
    Inherits DataContext
    Public DVDs As Table(Of DVD)
    Public Sub New(ByVal connection As String)
        MyBase.New(connection)
    End Sub
End Class

<Table(Name:="DVDTable")> _
Public Class DVD
    <Column(IsPrimaryKey:=True)> _
    Public Title As String
    <Column()> _
    Public Rating As String
End Class
```

Example

You can use the object model to create a database by doing the following:

```
public void CreateDatabase()
{
    MyDVDs db = new MyDVDs("c:\\mydvds.mdf");
    db.CreateDatabase();
}
```

```
Public Sub CreateDatabase()
    Dim db As New MyDVDs("c:\...\mydvds.mdf")
    db.CreateDatabase()
End Sub
```

Example

When building an application that automatically installs itself on a customer system, see if the database already exists and drop it before creating a new one. The DataContext class provides the DatabaseExists and DeleteDatabase methods to help you with this process.

The following example shows one way these methods can be used to implement this approach:

```
public void CreateDatabase2()
{
    MyDVDs db = new MyDVDs(@"c:\mydvds.mdf");
    if (db.DatabaseExists())
    {
        Console.WriteLine("Deleting old database...");
        db.DeleteDatabase();
    }
    db.CreateDatabase();
}
```

```
Public Sub CreateDatabase2()
    Dim db As MyDVDs = New MyDVDs("c:\...\mydvds.mdf")
    If db.DatabaseExists() Then
        Console.WriteLine("Deleting old database...")
        db.DeleteDatabase()
    End If
    db.CreateDatabase()
End Sub
```

See Also

Attribute-Based Mapping
External Mapping
SQL-CLR Type Mapping
Background Information
Making and Submitting Data Changes

How to: Manage Change Conflicts

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL provides a collection of APIs to help you discover, evaluate, and resolve concurrency conflicts.

In This Section

How to: Detect and Resolve Conflicting Submissions
Describes how to detect and resolve concurrency conflicts.

How to: Specify When Concurrency Exceptions are Thrown
Describes how to specify when you should be informed of concurrency conflicts.

How to: Specify Which Members are Tested for Concurrency Conflicts
Describes how to attribute members to specify whether they are checked for concurrency conflicts.

How to: Retrieve Entity Conflict Information
Describes how to gather information about entity conflicts.

How to: Retrieve Member Conflict Information
Describes how to gather information about member conflicts.

How to: Resolve Conflicts by Retaining Database Values
Describes how to overwrite current values with database values.

How to: Resolve Conflicts by Overwriting Database Values
Describes how to keep current values by overwriting database values.

How to: Resolve Conflicts by Merging with Database Values
Describes how to resolve a conflict by merging database and current values.

Related Sections

Optimistic Concurrency: Overview
Explains the terms that apply to optimistic concurrency in LINQ to SQL.

How to: Detect and Resolve Conflicting Submissions

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL provides many resources for detecting and resolving conflicts that stem from multi-user changes to the database. For more information, see How to: Manage Change Conflicts.

Example

The following example shows a `try` / `catch` block that catches a ChangeConflictException exception. Entity and member information for each conflict is displayed in the console window.

NOTE

You must include the `using System.Reflection` directive (`Imports System.Reflection` in Visual Basic) to support the information retrieval. For more information, see System.Reflection.

```
// using System.Reflection;
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

Customer newCust = new Customer();
newCust.City = "Auburn";
newCust.CustomerID = "AUBUR";
newCust.CompanyName = "AubCo";
db.Customers.InsertOnSubmit(newCust);

try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException e)
{
    Console.WriteLine("Optimistic concurrency error.");
    Console.WriteLine(e.Message);
    Console.ReadLine();
    foreach (ObjectChangeConflict occ in db.ChangeConflicts)
    {
        MetaTable metatable = db.Mapping.GetTable(occ.Object.GetType());
        Customer entityInConflict = (Customer)occ.Object;
        Console.WriteLine("Table name: {0}", metatable.TableName);
        Console.Write("Customer ID: ");
        Console.WriteLine(entityInConflict.CustomerID);
        foreach (MemberChangeConflict mcc in occ.MemberConflicts)
        {
            object currVal = mcc.CurrentValue;
            object origVal = mcc.OriginalValue;
            object databaseVal = mcc.DatabaseValue;
            MemberInfo mi = mcc.Member;
            Console.WriteLine("Member: {0}", mi.Name);
            Console.WriteLine("current value: {0}", currVal);
            Console.WriteLine("original value: {0}", origVal);
            Console.WriteLine("database value: {0}", databaseVal);
        }
    }
}
catch (Exception ee)
{
    // Catch other exceptions.
    Console.WriteLine(ee.Message);
}
finally
{
    Console.WriteLine("TryCatch block has finished.");
}
```

```
' Imports System.Reflection

Dim newCust As New Customer()
newCust.City = "Auburn"
newCust.CustomerID = "AUBUR"
newCust.CompanyName = "AubCo"
db.Customers.InsertOnSubmit(newCust)

Try
    db.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch e As ChangeConflictException
    Console.WriteLine("Optimistic concurrency error.")
    Console.WriteLine(e.Message)
    Console.ReadLine()
    For Each occ In db.ChangeConflicts

        Dim metatable As MetaTable = db.Mapping.GetTable(occ.Object.GetType())
        Dim entityInConflict = CType(occ.Object, Customer)
        Console.WriteLine("Table name: {0}", metatable.TableName)
        Console.Write("Customer ID: ")
        Console.WriteLine(entityInConflict.CustomerID)
        For Each mcc In occ.MemberConflicts

            Dim currVal = mcc.CurrentValue
            Dim origVal = mcc.OriginalValue
            Dim databaseVal = mcc.DatabaseValue
            Dim mi = mcc.Member
            Console.WriteLine("Member: {0}", mi.Name)
            Console.WriteLine("current value: {0}", currVal)
            Console.WriteLine("original value: {0}", origVal)
            Console.WriteLine("database value: {0}", databaseVal)
        Next
    Next

Catch ee As Exception
    ' Catch other exceptions.
    Console.WriteLine(ee.Message)
Finally
    Console.WriteLine("TryCatch block has finished.")
End Try
```

See Also

Making and Submitting Data Changes
How to: Manage Change Conflicts

How to: Specify When Concurrency Exceptions are Thrown

5/31/2017 • 1 min to read • Edit Online

In LINQ to SQL, a ChangeConflictException exception is thrown when objects do not update because of optimistic concurrency conflicts. For more information, see Optimistic Concurrency: Overview.

Before you submit your changes to the database, you can specify when concurrency exceptions should be thrown:

- Throw the exception at the first failure (FailOnFirstConflict).
- Finish all update tries, accumulate all failures, and report the accumulated failures in the exception (ContinueOnConflict).

When thrown, the ChangeConflictException exception provides access to a ChangeConflictCollection collection. This collection provides details for each conflict (mapped to a single failed update try), including access to the MemberConflicts collection. Each member conflict maps to a single member in the update that failed the concurrency check.

Example

The following code shows examples of both values.

```
Northwnd db = new Northwnd("...");

// Create, update, delete code.

db.SubmitChanges(ConflictMode.FailOnFirstConflict);
// or
db.SubmitChanges(ConflictMode.ContinueOnConflict);
```

```
Dim db As New Northwnd("...")

' Create, update, delete code.

db.SubmitChanges(ConflictMode.FailOnFirstConflict)
' or
db.SubmitChanges(ConflictMode.ContinueOnConflict)
```

See Also

How to: Manage Change Conflicts
Making and Submitting Data Changes

How to: Specify Which Members are Tested for Concurrency Conflicts

5/1/2017 • 1 min to read • Edit Online

Apply one of three enums to the LINQ to SQL UpdateCheck property on a ColumnAttribute attribute to specify which members are to be included in update checks for the detection of optimistic concurrency conflicts.

The UpdateCheck property (mapped at design time) is used together with run-time concurrency features in LINQ to SQL. For more information, see Optimistic Concurrency: Overview.

NOTE

Original member values are compared with the current database state as long as no member is designated as `IsVersion=true`. For more information, see IsVersion.

For code examples, see UpdateCheck.

To always use this member for detecting conflicts

1. Add the UpdateCheck property to the ColumnAttribute attribute.

2. Set the UpdateCheck property value to `Always`.

To never use this member for detecting conflicts

1. Add the UpdateCheck property to the ColumnAttribute attribute.

2. Set the UpdateCheck property value to `Never`.

To use this member for detecting conflicts only when the application has changed the value of the member

1. Add the UpdateCheck property to the ColumnAttribute attribute.

2. Set the UpdateCheck property value to `WhenChanged`.

Example

The following example specifies that `HomePage` objects should never be tested during update checks. For more information, see UpdateCheck.

```
[Column(Storage="_HomePage", DbType="NText", UpdateCheck=UpdateCheck.Never)]
public string HomePage
{
    get
    {
        return this._HomePage;
    }
    set
    {
        if ((this._HomePage != value))
 {
      this.OnHomePageChanging(value);
     this.SendPropertyChanging();
            this._HomePage = value;
     this.SendPropertyChanged("HomePage");
            this.OnHomePageChanged();
 }
    }
}
```

```
<Column(Storage:="_HomePage", DbType:="NText", UpdateCheck:=UpdateCheck.Never)>  _
Public Property HomePage() As String
    Get
        Return Me._HomePage
    End Get
    Set(ByVal value As String)
        If ((Me._HomePage <> value)  _
            = false) Then
     Me.OnHomePageChanging(value)
            Me.SendPropertyChanging
            Me._HomePage = value
            Me.SendPropertyChanged("HomePage")
            Me.OnHomePageChanged
        End If
    End Set
End Property
```

See Also

How to: Manage Change Conflicts
Making and Submitting Data Changes

How to: Retrieve Entity Conflict Information

5/1/2017 • 1 min to read • Edit Online

You can use objects of the ObjectChangeConflict class to provide information about conflicts revealed by ChangeConflictException exceptions. For more information, see Optimistic Concurrency: Overview.

Example

The following example iterates through a list of accumulated conflicts.

```
Northwnd db = new Northwnd("...");

try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}

catch (ChangeConflictException e)
{
    Console.WriteLine("Optimistic concurrency error.");
    Console.WriteLine(e.Message);
    foreach (ObjectChangeConflict occ in db.ChangeConflicts)
    {
        MetaTable metatable = db.Mapping.GetTable(occ.Object.GetType());
        Customer entityInConflict = (Customer)occ.Object;
        Console.WriteLine("Table name: {0}", metatable.TableName);
        Console.Write("Customer ID: ");
        Console.WriteLine(entityInConflict.CustomerID);
        Console.ReadLine();
    }
}
```

```
Dim db As New Northwnd("...")

Try
    db.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch ex As ChangeConflictException
    Console.WriteLine("Optimistic concurrency error.")
    Console.WriteLine(ex.Message)
    For Each occ As ObjectChangeConflict In db.ChangeConflicts
        Dim metatable As MetaTable = db.Mapping.GetTable(occ.Object.GetType())
        Dim entityInConflict = occ.Object

        Console.WriteLine("Table name: " & metatable.TableName)
        Console.Write("Customer ID: ")
        Console.WriteLine(entityInConflict.CustomerID)
        Console.ReadLine()
    Next
End Try
```

See Also

How to: Manage Change Conflicts

How to: Retrieve Member Conflict Information

5/1/2017 • 1 min to read • Edit Online

You can use the MemberChangeConflict class to retrieve information about individual members in conflict. In this same context you can provide for custom handling of the conflict for any member. For more information, see Optimistic Concurrency: Overview.

Example

The following code iterates through the ObjectChangeConflict objects. For each object, it then iterates through the MemberChangeConflict objects.

NOTE

Include System.Reflection in order to provide Member information.

```
// Add 'using System.Reflection' for this section.
Northwnd db = new Northwnd("...");

try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}

catch (ChangeConflictException e)
{
    Console.WriteLine("Optimistic concurrency error.");
    Console.WriteLine(e.Message);
    foreach (ObjectChangeConflict occ in db.ChangeConflicts)
    {
        MetaTable metatable = db.Mapping.GetTable(occ.Object.GetType());
        Customer entityInConflict = (Customer)occ.Object;
        Console.WriteLine("Table name: {0}", metatable.TableName);
        Console.Write("Customer ID: ");
        Console.WriteLine(entityInConflict.CustomerID);
        foreach (MemberChangeConflict mcc in occ.MemberConflicts)
        {
            object currVal = mcc.CurrentValue;
            object origVal = mcc.OriginalValue;
            object databaseVal = mcc.DatabaseValue;
            MemberInfo mi = mcc.Member;
            Console.WriteLine("Member: {0}", mi.Name);
            Console.WriteLine("current value: {0}", currVal);
            Console.WriteLine("original value: {0}", origVal);
            Console.WriteLine("database value: {0}", databaseVal);
            Console.ReadLine();
        }
    }
}
```

```
' Add 'Imports System.Reflection' for this section.
Dim db As New Northwnd("...")
'...
Try
    db.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch ex As ChangeConflictException
    Console.WriteLine("Optimistic concurrency error.")
    Console.WriteLine(ex.Message)
    For Each occ As ObjectChangeConflict In db.ChangeConflicts
        Dim metatable As MetaTable = db.Mapping.GetTable(occ.Object.GetType)
        Dim entityInConflict As Object = occ.Object

        Console.WriteLine("Table name: " & metatable.TableName)
        Console.Write("Customer ID: ")
        Console.WriteLine(entityInConflict.CustomerID)

        For Each mcc As MemberChangeConflict In occ.MemberConflicts
            Dim currVal = mcc.CurrentValue
            Dim origVal = mcc.OriginalValue
            Dim databaseVal = mcc.DatabaseValue
            Dim mi As MemberInfo = mcc.Member

            Console.WriteLine("Member: " & mi.Name)
            Console.WriteLine("current value: " & currVal)
            Console.WriteLine("original value: " & origVal)
            Console.WriteLine("database value: " & databaseVal)
            Console.ReadLine()
        Next
    Next
End Try
```

See Also

How to: Manage Change Conflicts

How to: Resolve Conflicts by Retaining Database Values

5/31/2017 • 1 min to read • Edit Online

To reconcile differences between expected and actual database values before you try to resubmit your changes, you can use OverwriteCurrentValues to retain the values found in the database. The current values in the object model are then overwritten. For more information, see Optimistic Concurrency: Overview.

NOTE

In all cases, the record on the client is first refreshed by retrieving the updated data from the database. This action makes sure that the next update try will not fail on the same concurrency checks.

Example

In this scenario, a ChangeConflictException exception is thrown when User1 tries to submit changes, because User2 has in the meantime changed the Assistant and Department columns. The following table shows the situation.

	MANAGER	ASSISTANT	DEPARTMENT
Original database state when queried by User1 and User2.	Alfreds	Maria	Sales
User1 prepares to submit these changes.	Alfred		Marketing
User2 has already submitted these changes.		Mary	Service

User1 decides to resolve this conflict by having the newer database values overwrite the current values in the object model.

When User1 resolves the conflict by using OverwriteCurrentValues, the result in the database is as follows in the table:

	MANAGER	ASSISTANT	DEPARTMENT
New state after conflict resolution.	Alfreds (original)	Mary (from User2)	Service (from User2)

The following example code shows how to overwrite current values in the object model with the database values. (No inspection or custom handling of individual member conflicts occurs.)

```
Northwnd db = new Northwnd("...");
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}

catch (ChangeConflictException e)
{
    Console.WriteLine(e.Message);
    foreach (ObjectChangeConflict occ in db.ChangeConflicts)
    {
        // All database values overwrite current values.
        occ.Resolve(RefreshMode.OverwriteCurrentValues);
    }
}
```

```
Dim db As New Northwnd("...")

Try
    db.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch ex As ChangeConflictException
    Console.WriteLine(ex.Message)

    For Each occ As ObjectChangeConflict In db.ChangeConflicts
        ' All database values overwrite current values.
        occ.Resolve(Data.Linq.RefreshMode.OverwriteCurrentValues)
    Next

End Try
```

See Also

How to: Manage Change Conflicts

How to: Resolve Conflicts by Overwriting Database Values

5/31/2017 • 1 min to read • Edit Online

To reconcile differences between expected and actual database values before you try to resubmit your changes, you can use KeepCurrentValues to overwrite database values. For more information, see Optimistic Concurrency: Overview.

NOTE

In all cases, the record on the client is first refreshed by retrieving the updated data from the database. This action makes sure that the next update try will not fail on the same concurrency checks.

Example

In this scenario, an ChangeConflictException exception is thrown when User1 tries to submit changes, because User2 has in the meantime changed the Assistant and Department columns. The following table shows the situation.

	MANAGER	ASSISTANT	DEPARTMENT
Original database state when queried by User1 and User2.	Alfreds	Maria	Sales
User1 prepares to submit these changes.	Alfred		Marketing
User2 has already submitted these changes.		Mary	Service

User1 decides to resolve this conflict by overwriting database values with the current client member values.

When User1 resolves the conflict by using KeepCurrentValues, the result in the database is as in following table:

	MANAGER	ASSISTANT	DEPARTMENT
New state after conflict resolution.	Alfred (from User1)	Maria (original)	Marketing (from User1)

The following example code shows how to overwrite database values with the current client member values. (No inspection or custom handling of individual member conflicts occurs.)

```
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}

catch (ChangeConflictException e)
{
    Console.WriteLine(e.Message);
    foreach (ObjectChangeConflict occ in db.ChangeConflicts)
    {
        //No database values are merged into current.
        occ.Resolve(RefreshMode.KeepCurrentValues);
    }
}
```

```
Try
    db.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch ex As ChangeConflictException
    Console.WriteLine(ex.Message)

    For Each occ As ObjectChangeConflict In db.ChangeConflicts
        ' No database values are merged into current.
        occ.Resolve(Data.Linq.RefreshMode.KeepCurrentValues)
    Next

End Try
```

See Also

How to: Manage Change Conflicts

How to: Resolve Conflicts by Merging with Database Values

5/31/2017 • 1 min to read • Edit Online

To reconcile differences between expected and actual database values before you try to resubmit your changes, you can use KeepChanges to merge database values with the current client member values. For more information, see Optimistic Concurrency: Overview.

NOTE

In all cases, the record on the client is first refreshed by retrieving the updated data from the database. This action makes sure that the next update try will not fail on the same concurrency checks.

Example

In this scenario, a ChangeConflictException exception is thrown when User1 tries to submit changes, because User2 has in the meantime changed the Assistant and Department columns. The following table shows the situation.

	MANAGER	ASSISTANT	DEPARTMENT
Original database state when queried by User1 and User2.	Alfreds	Maria	Sales
User1 prepares to submit these changes.	Alfred		Marketing
User2 has already submitted these changes.		Mary	Service

User1 decides to resolve this conflict by merging database values with the current client member values. The result will be that database values are overwritten only when the current changeset has also modified that value.

When User1 resolves the conflict by using KeepChanges, the result in the database is as in the following table:

	MANAGER	ASSISTANT	DEPARTMENT
New state after conflict resolution.	Alfred (from User1)	Mary (from User2)	Marketing (from User1)

The following example shows how to merge database values with the current client member values (unless the client has also changed that value). No inspection or custom handling of individual member conflicts occurs.

```
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}

catch (ChangeConflictException e)
{
    Console.WriteLine(e.Message);
    // Automerge database values for members that client
    // has not modified.
    foreach (ObjectChangeConflict occ in db.ChangeConflicts)
    {
        occ.Resolve(RefreshMode.KeepChanges);
    }
}

// Submit succeeds on second try.
db.SubmitChanges(ConflictMode.FailOnFirstConflict);
```

```
Try
    db.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch ex As ChangeConflictException
    Console.WriteLine(ex.Message)

    For Each occ As ObjectChangeConflict In db.ChangeConflicts
        ' Automerge database values into current for members
        ' that client has not modified.
        occ.Resolve(Data.Linq.RefreshMode.KeepChanges)
    Next

End Try

' Submit succeeds on second try.
db.SubmitChanges(ConflictMode.FailOnFirstConflict)
```

See Also

How to: Resolve Conflicts by Overwriting Database Values
How to: Resolve Conflicts by Retaining Database Values
How to: Manage Change Conflicts

Debugging Support

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL provides general debugging support for LINQ to SQL projects. Also see Debugging LINQ or Debugging LINQ.

LINQ to SQL also provides special tools for viewing SQL code. For more information, see the topics in this section.

In This Section

How to: Display Generated SQL
Describes how to use DataContext properties to view query activity.

How to: Display a ChangeSet
Describes how to show changes being sent to the database.

How to: Display LINQ to SQL Commands
Describes how to display SQL commands and other information.

Troubleshooting
Presents common scenarios whose causes might be hard to determine.

See Also

Programming Guide

How to: Display Generated SQL

5/1/2017 • 1 min to read • Edit Online

You can view the SQL code generated for queries and change processing by using the Log property. This approach can be useful for understanding LINQ to SQL functionality and for debugging specific problems.

Example

The following example uses the Log property to display SQL code in the console window before the code is executed. You can use this property with query, insert, update, and delete commands.

The lines from the console window are what you see when you execute the Visual Basic or C# code that follows.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName], [t0].[ContactT
itle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Coun
try], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0
-- @p0: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20810.0
```

```
AROUT
BSBEV
CONSH
EASTC
NORTS
SEVES
```

```
db.Log = Console.Out;
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach(Customer custObj in custQuery)
{
    Console.WriteLine(custObj.CustomerID);
}
```

```
db.Log = Console.Out
Dim custQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust

For Each custObj In custQuery
    Console.WriteLine(custObj.CustomerID)
Next
```

See Also

Debugging Support

How to: Display a ChangeSet

5/1/2017 • 1 min to read • Edit Online

You can view changes tracked by a DataContext by using GetChangeSet.

Example

The following example retrieves customers whose city is London, changes the city to Paris, and submits the changes back to the database.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

var custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custQuery)
{
    Console.WriteLine("CustomerID: {0}", custObj.CustomerID);
    Console.WriteLine("\tOriginal value: {0}", custObj.City);
    custObj.City = "Paris";
    Console.WriteLine("\tUpdated value: {0}", custObj.City);
}

ChangeSet cs = db.GetChangeSet();
Console.Write("Total changes: {0}", cs);
// Freeze the console window.
Console.ReadLine();

db.SubmitChanges();
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

Dim custQuery = _
    From cust In db.Customers _
    Where (cust.City = "London") _
    Select cust

For Each custObj As Customer In custQuery
    Console.WriteLine("CustomerID: {0}", custObj.CustomerID)
    Console.WriteLine(vbTab & "Original value: {0}", custObj.City)
    custObj.City = "Paris"
    Console.WriteLine(vbTab & "Updated value: {0}", custObj.City)
Next

Dim cs As ChangeSet = db.GetChangeSet()
Console.Write("Total changes: {0}", cs)
' Freeze the console window.
Console.ReadLine()

db.SubmitChanges()
```

Output from this code appears similar to the following. Note that the summary at the end shows that eight changes were made.

```
CustomerID: AROUT
```

`Original value: London`

`Updated value: Paris`

`CustomerID: BSBEV`

`Original value: London`

`Updated value: Paris`

`CustomerID: CONSH`

`Original value: London`

`Updated value: Paris`

`CustomerID: EASTC`

`Original value: London`

`Updated value: Paris`

`CustomerID: NORTS`

`Original value: London`

`Updated value: Paris`

`CustomerID: PARIS`

`Original value: London`

`Updated value: Paris`

`CustomerID: SEVES`

`Original value: London`

`Updated value: Paris`

`CustomerID: SPECD`

`Original value: London`

`Updated value: Paris`

``

`Total changes: {Added: 0, Removed: 0, Modified: 8}`

See Also

Debugging Support

How to: Display LINQ to SQL Commands

5/16/2017 • 1 min to read • Edit Online

Use GetCommand to display SQL commands and other information.

Example

In the following example, the console window displays the output from the query, followed by the SQL commands that are generated, the type of commands, and the type of connection.

```
// using System.Data.Common;
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

var q =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

Console.WriteLine("Customers from London:");
foreach (var z in q)
{
    Console.WriteLine("\t {0}",z.ContactName);
}

DbCommand dc = db.GetCommand(q);
Console.WriteLine("\nCommand Text: \n{0}",dc.CommandText);
Console.WriteLine("\nCommand Type: {0}",dc.CommandType);
Console.WriteLine("\nConnection: {0}",dc.Connection);

Console.ReadLine();
```

```
' Imports System.Data.Common
Dim db As New Northwnd("c:\northwnd.mdf")

Dim q = _
From cust In db.Customers _
Where cust.City = "London" _
Select cust

Console.WriteLine("Customers from London:")
For Each z As Customer In q
    Console.WriteLine(vbTab & z.ContactName)
Next

Dim dc As DbCommand = db.GetCommand(q)
Console.WriteLine(vbNewLine & "Command Text: " & vbNewLine & dc.CommandText)
Console.WriteLine(vbNewLine & "Command Type: {0}", dc.CommandType)
Console.WriteLine(vbNewLine & "Connection: {0}", dc.Connection)

Console.ReadLine()
```

Output appears as follows:

```
Customers from London:
    Thomas Hardy
    Victoria Ashworth
    Elizabeth Brown
    Ann Devon
    Simon Crowther
    Marie Bertrand
    Hari Kumar
    Dominique Perrier
```

```
Command Text:
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName], [t0].[ContactT
itle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Coun
try], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0

Command Type: Text

Connection: System.Data.SqlClient.SqlConnection
```

See Also

Debugging Support

Troubleshooting

5/1/2017 • 3 min to read • Edit Online

The following information exposes some issues you might encounter in your LINQ to SQL applications, and provides suggestions to avoid or otherwise reduce the effect of these issues.

Additional issues are addressed in Frequently Asked Questions.

Unsupported Standard Query Operators

LINQ to SQL does not support all standard query operator methods (for example, ElementAt). As a result, projects that compile can still produce run-time errors. For more information, see Standard Query Operator Translation.

Memory Issues

If a query involves an in-memory collection and LINQ to SQL Table<TEntity>, the query might be executed in memory, depending on the order in which the two collections are specified. If the query must be executed in memory, then the data from the database table will need to be retrieved.

This approach is inefficient and could result in significant memory and processor usage. Try to avoid such multi-domain queries.

File Names and SQLMetal

To specify an input file name, add the name to the command line as the input file. Including the file name in the connection string (using the **/conn** option) is not supported. For more information, see SqlMetal.exe (Code Generation Tool).

Class Library Projects

The Object Relational Designer creates a connection string in the `app.config` file of the project. In class library projects, the `app.config` file is not used. LINQ to SQL uses the Connection String provided in the design-time files. Changing the value in `app.config` does not change the database to which your application connects.

Cascade Delete

LINQ to SQL does not support or recognize cascade-delete operations. If you want to delete a row in a table that has constraints against it, you must do either of the following:

- Set the `ON DELETE CASCADE` rule in the foreign-key constraint in the database.
- Use your own code to first delete the child objects that prevent the parent object from being deleted.

Otherwise, a SqlException exception is thrown.

For more information, see How to: Delete Rows From the Database.

Expression Not Queryable

If you get the "Expression [expression] is not queryable; are you missing an assembly reference?" error, make sure of the following:

- Your application is targeting .NET Compact Framework 3.5.
- You have a reference to `System.Core.dll` and `System.Data.Linq.dll`.
- You have an `Imports` (Visual Basic) or `using` (C#) directive for System.Linq and System.Data.Linq.

DuplicateKeyException

In the course of debugging a LINQ to SQL project, you might traverse an entity's relations. Doing so brings these items into the cache, and LINQ to SQL becomes aware of their presence. If you then try to execute Attach or InsertOnSubmit or a similar method that produces multiple rows that have the same key, a DuplicateKeyException is thrown.

String Concatenation Exceptions

Concatenation on operands mapped to `[n]text` and other `[n][var]char` is not supported. An exception is thrown for concatenation of strings mapped to the two different sets of types. For more information, see System.String Methods.

Skip and Take Exceptions in SQL Server 2000

You must use identity members (IsPrimaryKey) when you use Take or Skip against a SQL Server 2000 database. The query must be against a single table (that is, not a join), or be a Distinct, Except, Intersect, or Union operation, and must not include a Concat operation. For more information, see the "SQL Server 2000 Support" section in Standard Query Operator Translation.

This requirement does not apply to SQL Server 2005.

GroupBy InvalidOperationException

This exception is thrown when a column value is null in a GroupBy query that groups by a `boolean` expression, such as `group x by (Phone==@phone)`. Because the expression is a `boolean`, the key is inferred to be `boolean`, not `nullable``boolean`. When the translated comparison produces a null, an attempt is made to assign a `nullable``boolean` to a `boolean`, and the exception is thrown.

To avoid this situation (assuming you want to treat nulls as false), use an approach such as the following:

```
GroupBy="(Phone != null) && (Phone=@Phone)"
```

OnCreated() Partial Method

The generated method `OnCreated()` is called each time the object constructor is called, including the scenario in which LINQ to SQL calls the constructor to make a copy for original values. Take this behavior into account if you implement the `OnCreated()` method in your own partial class.

See Also

Debugging Support
Frequently Asked Questions

Background Information

5/1/2017 • 2 min to read • Edit Online

The topics in this section pertain to concepts and procedures that extend beyond the basics about using LINQ to SQL.

Follow these steps to find additional examples of LINQ to SQL code and applications:

- Search the MSDN Library for specific issues.
- Participate in the LINQ Forum, where you can discuss more complex topics in detail with experts.
- Study the whitepaper that details LINQ to SQL technology, complete with Visual Basic and C# code examples. For more information, see LINQ to SQL: .NET Language-Integrated Query for Relational Data.

In This Section

ADO.NET and LINQ to SQL
Describes the relationship of ADO.NET and LINQ to SQL.

Analyzing LINQ to SQL Source Code
Describes how to analyze LINQ to SQL mapping by generating and viewing source code from the Northwind sample database.

Customizing Insert, Update, and Delete Operations
Describes how to add validation code and other customizations.

Data Binding
Describes how LINQ to SQL uses IListSource to support data binding.

Inheritance Support
Describes the role of inheritance in the LINQ to SQL object model, and how to use related operators in your queries.

Local Method Calls
Describes LINQ to SQL support for local method calls.

N-Tier and Remote Applications with LINQ to SQL
Provides detailed information for multi-tier applications that use LINQ to SQL.

Object Identity
Describes object identity in the LINQ to SQL object model, and explains how this feature differs from object identity in a database.

The LINQ to SQL Object Model
Describes the object model and its relationship to the relational data model.

Object States and Change-Tracking
Provides detailed information about how LINQ to SQL tracks changes.

Optimistic Concurrency: Overview
Describes optimistic concurrency and defines terms.

Query Concepts
Describes aspects of queries in LINQ to SQL that differ from queries in LINQ.

Retrieving Objects from the Identity Cache
Describes the types of queries that retrieve objects from the identity cache.

Security in LINQ to SQL
Describes the correct approach to security in database connections.

Serialization
Describes the serialization process in LINQ to SQL applications.

Stored Procedures
Describes how to map stored procedures at design time and how to call them from your application.

Transaction Support
Outlines the three models of transaction that LINQ to SQL supports.

SQL-CLR Type Mismatches
Describes the challenges of mingling different type systems.

SQL-CLR Custom Type Mappings
Provides guidance on customizing type mappings.

User-Defined Functions
Describes how to map user-defined functions at design time and how to call them from your application.

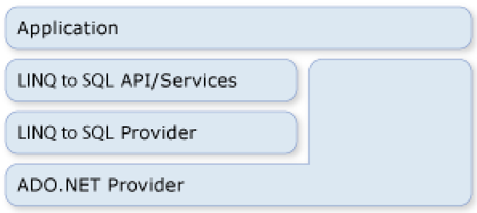
Related Sections

Programming Guide
Includes links to sections that explain various aspects of the LINQ to SQL.

ADO.NET and LINQ to SQL

5/1/2017 • 3 min to read • Edit Online

LINQ to SQL is part of the ADO.NET family of technologies. It is based on services provided by the ADO.NET provider model. You can therefore mix LINQ to SQL code with existing ADO.NET applications and migrate current ADO.NET solutions to LINQ to SQL. The following illustration provides a high-level view of the relationship.



Connections

You can supply an existing ADO.NET connection when you create a LINQ to SQL DataContext. All operations against the DataContext (including queries) use this provided connection. If the connection is already open, LINQ to SQL leaves it as is when you are finished with it.

```
string connString = @"Data Source=.\SQLEXPRESS;AttachDbFilename=c:\northwind.mdf;
    Integrated Security=True; Connect Timeout=30; User Instance=True";
SqlConnection nwindConn = new SqlConnection(connString);
nwindConn.Open();

Northwnd interop_db = new Northwnd(nwindConn);

SqlTransaction nwindTxn = nwindConn.BeginTransaction();

try
{
    SqlCommand cmd = new SqlCommand(
        "UPDATE Products SET QuantityPerUnit = 'single item' WHERE ProductID = 3");
    cmd.Connection = nwindConn;
    cmd.Transaction = nwindTxn;
    cmd.ExecuteNonQuery();

    interop_db.Transaction = nwindTxn;

    Product prod1 = interop_db.Products
        .First(p => p.ProductID == 4);
    Product prod2 = interop_db.Products
        .First(p => p.ProductID == 5);
    prod1.UnitsInStock -= 3;
    prod2.UnitsInStock -= 5;

    interop_db.SubmitChanges();

    nwindTxn.Commit();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine("Error submitting changes... all changes rolled back.");
}

nwindConn.Close();
```

```
Dim conString = "Data Source=.\SQLEXPRESS;AttachDbFilename=c:\northwind.mdf; Integrated Security=True;Connect
Timeout=30;User Instance=True"
Dim northwindCon = New SqlConnection(conString)
northwindCon.Open()

Dim db = New Northwnd("...")
Dim northwindTransaction = northwindCon.BeginTransaction()

Try
    Dim cmd = New SqlCommand( _
            "UPDATE Products SET QuantityPerUnit = 'single item' " & _
            "WHERE ProductID = 3")
    cmd.Connection = northwindCon
    cmd.Transaction = northwindTransaction
    cmd.ExecuteNonQuery()

    db.Transaction = northwindTransaction

    Dim prod1 = (From prod In db.Products _
  Where prod.ProductID = 4).First
    Dim prod2 = (From prod In db.Products _
  Where prod.ProductID = 5).First
    prod1.UnitsInStock -= 3
    prod2.UnitsInStock -= 5

    db.SubmitChanges()

    northwindTransaction.Commit()

Catch e As Exception

    Console.WriteLine(e.Message)
    Console.WriteLine("Error submitting changes... " & _
"all changes rolled back.")
End Try

northwindCon.Close()
```

You can always access the connection and close it yourself by using the Connection property, as in the following code:

```
db.Connection.Close();
```

```
db.Connection.Close()
```

Transactions

You can supply your DataContext with your own database transaction when your application has already initiated the transaction and you want your DataContext to be involved.

The preferred method of doing transactions with the .NET Framework is to use the TransactionScope object. By using this approach, you can make distributed transactions that work across databases and other memory-resident resource managers. Transaction scopes require few resources to start. They promote themselves to distributed transactions only when there are multiple connections within the scope of the transaction.

```
using (TransactionScope ts = new TransactionScope())
{
    db.SubmitChanges();
    ts.Complete();
}
```

```
Using ts As New TransactionScope()
    db.SubmitChanges()
    ts.Complete()
End Using
```

You cannot use this approach for all databases. For example, the SqlClient connection cannot promote system transactions when it works against a SQL Server 2000 server. Instead, it automatically enlists to a full, distributed transaction whenever it sees a transaction scope being used.

Direct SQL Commands

At times you can encounter situations where the ability of the DataContext to query or submit changes is insufficient for the specialized task you want to perform. In these circumstances you can use the ExecuteQuery method to issue SQL commands to the database and convert the query results to objects.

For example, assume that the data for the `Customer` class is spread over two tables (customer1 and customer2). The following query returns a sequence of `Customer` objects:

```
            IEnumerable<Customer> results = db.ExecuteQuery<Customer>(
    @"select c1.custid as CustomerID, c2.custName as ContactName
        from customer1 as c1, customer2 as c2
        where c1.custid = c2.custid"
);
```

```
    Dim results As IEnumerable(Of Customer) = _
db.ExecuteQuery(Of Customer)( _
"SELECT [c1].custID as CustomerID," & _
    "[c2].custName as ContactName" & _
    "FROM customer1 AS [c1], customer2 as [c2]" & _
    "WHERE [c1].custid = [c2].custid")
```

As long as the column names in the tabular results match column properties of your entity class, LINQ to SQL creates your objects out of any SQL query.

Parameters

The ExecuteQuery method accepts parameters. The following code executes a parameterized query:

```
            IEnumerable<Customer> results = db.ExecuteQuery<Customer>(
    "select contactname from customers where city = {0}",
    "London"
);
```

```
    Dim results As IEnumerable(Of Customer) = _
db.ExecuteQuery(Of Customer)( _
    "SELECT contactname FROM customers WHERE city = {0}, 'London'")
End Sub
```

NOTE

Parameters are expressed in the query text by using the same curly notation used by `Console.WriteLine()` and `String.Format()`. `String.Format()` takes the query string you provide and substitutes the curly-braced parameters with generated parameter names such as `@p0`, `@p1` …, `@p(n)`.

See Also

Background Information

How to: Reuse a Connection Between an ADO.NET Command and a DataContext

Analyzing LINQ to SQL Source Code

5/1/2017 • 1 min to read • Edit Online

By using the following steps, you can produce LINQ to SQL source code from the Northwind sample database. You can compare elements of the object model with elements of the database to better see how different items are mapped.

NOTE

Developers using Visual Studio can use the O/R Designer to produce this code.

1. If you do not already have the Northwind sample database on your development computer, you can download it free of charge. For more information, see Downloading Sample Databases.
2. Use the SqlMetal command-line tool to generate a Visual Basic or C# source file. For more information, see SqlMetal.exe (Code Generation Tool). By typing the following commands at a command prompt, you can generate Visual Basic and C# source files that include stored procedures and functions:
   - `sqlmetal /code:northwind.vb /language:vb "c:\northwnd.mdf" /sprocs /functions /pluralize`
   - `sqlmetal /code:northwind.cs /language:csharp "c:\northwnd.mdf" /sprocs /functions /pluralize`

See Also

Reference
Background Information

Customizing Insert, Update, and Delete Operations

5/1/2017 • 1 min to read • Edit Online

By default, LINQ to SQL generates dynamic SQL to implement insert, read, update, and delete operations. In practice, however, you typically customize your application to suit your business needs.

NOTE

If you are using Visual Studio, you can use the Object Relational Designer to customize insert, update, and delete actions.

This section of topics describes the techniques that LINQ to SQL provides for customizing insert, read, update, and delete operations in your application.

In This Section

Customizing Operations: Overview
Describes the various techniques LINQ to SQL provides for customizing insert, read, update, and delete operations.

Insert, Update, and Delete Operations
Describes the LINQ to SQL default processes for manipulating database data.

Responsibilities of the Developer In Overriding Default Behavior
Describes the role of the developer in implementing requirements not enforced by LINQ to SQL.

Adding Business Logic By Using Partial Methods
Describes how to use partial methods to override autogenerated methods.

Customizing Operations: Overview

5/1/2017 • 1 min to read • Edit Online

By default, LINQ to SQL generates dynamic SQL for insert, update, and delete operations based on mapping. However, in practice you typically want to add your own business logic to provide for security, validation, and so forth.

LINQ to SQL techniques for customizing these operations include the following.

Loading Options

In your queries, you can control how much data related to your main target is retrieved when you connect to the database. This functionality is implemented largely by using DataLoadOptions. For more information, see Deferred versus Immediate Loading.

Partial Methods

In its default mapping, LINQ to SQL provides partial methods to help you implement your business logic. For more information, see Adding Business Logic By Using Partial Methods.

Stored Procedures and User-Defined Functions

LINQ to SQL supports the use of stored procedures and user-defined functions. Stored procedures are frequently used to customize operations. For more information, see Stored Procedures.

See Also

Customizing Insert, Update, and Delete Operations

Insert, Update, and Delete Operations

5/1/2017 • 2 min to read • Edit Online

You perform `Insert`, `Update`, and `Delete` operations in LINQ to SQL by adding, changing, and removing objects in your object model. By default, LINQ to SQL translates your actions to SQL and submits the changes to the database.

LINQ to SQL offers maximum flexibility in manipulating and persisting changes that you made to your objects. As soon as entity objects are available (either by retrieving them through a query or by constructing them anew), you can change them as typical objects in your application. That is, you can change their values, you can add them to your collections, and you can remove them from your collections. LINQ to SQL tracks your changes and is ready to transmit them back to the database when you call SubmitChanges.

NOTE

LINQ to SQL does not support or recognize cascade-delete operations. If you want to delete a row in a table that has constraints against it, you must either set the `ON DELETE CASCADE` rule in the foreign-key constraint in the database, or use your own code to first delete the child objects that prevent the parent object from being deleted. Otherwise, an exception is thrown. For more information, see How to: Delete Rows From the Database.

The following excerpts use the `Customer` and `Order` classes from the Northwind sample database. Class definitions are not shown for brevity.

```
Northwnd db = new Northwnd(@"c:\Northwnd.mdf");

// Query for a specific customer.
var cust =
    (from c in db.Customers
     where c.CustomerID == "ALFKI"
     select c).First();

// Change the name of the contact.
cust.ContactName = "New Contact";

// Create and add a new Order to the Orders collection.
Order ord = new Order { OrderDate = DateTime.Now };
cust.Orders.Add(ord);

// Delete an existing Order.
Order ord0 = cust.Orders[0];

// Removing it from the table also removes it from the Customer’s list.
db.Orders.DeleteOnSubmit(ord0);

// Ask the DataContext to save all the changes.
db.SubmitChanges();
```

```
Dim db As New Northwnd("…\Northwnd.mdf")

Dim cust As Customer = _
(From c In db.Customers _
Where c.CustomerID = "ALFKI" _
Select c) _
.First()

' Change the name of the contact.
cust.ContactName = "New Contact"

' Create and add a new Order to Orders collection.
Dim ord As New Order With {.OrderDate = DateTime.Now}
cust.Orders.Add(ord)

' Delete an existing Order.
Dim ord0 As Order = cust.Orders(0)

' Removing it from the table also removes it from
' the Customer’s list.
db.Orders.DeleteOnSubmit(ord0)

' Ask the DataContext to save all the changes.
db.SubmitChanges()
```

When you call SubmitChanges, LINQ to SQL automatically generates and executes the SQL commands that it must have to transmit your changes back to the database.

NOTE

You can override this behavior by using your own custom logic, typically by way of a stored procedure. For more information, see Responsibilities of the Developer In Overriding Default Behavior.

Developers using Visual Studio can use the Object Relational Designer to develop stored procedures for this purpose.

See Also

Downloading Sample Databases
Customizing Insert, Update, and Delete Operations

Responsibilities of the Developer In Overriding Default Behavior

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL does not enforce the following requirements, but behavior is undefined if these requirements are not satisfied.

- The overriding method must not call SubmitChanges or Attach. LINQ to SQL throws an exception if these methods are called in an override method.
- Override methods cannot be used to start, commit, or stop a transaction. The SubmitChanges operation is performed under a transaction. An inner nested transaction can interfere with the outer transaction. Load override methods can start a transaction only after they determine that the operation is not being performed in a Transaction.
- Override methods are expected to follow the applicable optimistic concurrency mapping. The override method is expected to throw a ChangeConflictException when an optimistic concurrency conflict occurs. LINQ to SQL catches this exception so that you can correctly process the SubmitChanges option provided on SubmitChanges.
- Create (`Insert`) and `Update` override methods are expected to flow back the values for database-generated columns to corresponding object members when the operation is successfully completed.

  For example, if `Order.OrderID` is mapped to an identity column (*autoincrement* primary key), then the `InsertOrder()` override method must retrieve the database-generated ID and set the `Order.OrderID` member to that ID. Likewise, timestamp members must be updated to the database-generated timestamp values to make sure that the updated objects are consistent. Failure to propagate the database-generated values can cause an inconsistency between the database and the objects tracked by the DataContext.
- It is the user's responsibility to invoke the correct dynamic API. For example, in the update override method, only the ExecuteDynamicUpdate can be called. LINQ to SQL does not detect or verify whether the invoked dynamic method matches the applicable operation. If an inapplicable method is called (for example, ExecuteDynamicDelete for an object to be updated), the results are undefined.
- Finally, the overriding method is expected to perform the stated operation. The semantics of LINQ to SQL operations such as eager loading, deferred loading, and SubmitChanges) require the overrides to provide the stated service. For example, a load override that just returns an empty collection without checking the contents in the database will likely lead to inconsistent data.

See Also

Customizing Insert, Update, and Delete Operations

Adding Business Logic By Using Partial Methods

5/1/2017 • 3 min to read • Edit Online

You can customize Visual Basic and C# generated code in your LINQ to SQL projects by using *partial methods*. The code that LINQ to SQL generates defines signatures as one part of a partial method. If you want to implement the method, you can add your own partial method. If you do not add your own implementation, the compiler discards the partial methods signature and calls the default methods in LINQ to SQL.

NOTE

If you are using Visual Studio, you can use the Object Relational Designer to add validation and other customizations to entity classes.

For example, the default mapping for the `Customer` class in the Northwind sample database includes the following partial method:

```
partial void OnAddressChanged();
```

```
Partial Private Sub OnAddressChanged()
End Sub
```

You can implement your own method by adding code such as the following to your own partial `Customer` class:

```
public partial class Customer
{
    partial void OnAddressChanged();
    partial void OnAddressChanged()
    {
        // Insert business logic here.
    }
}
```

```
Partial Class Customer
    Private Sub OnAddressChanged()
        ' Insert business logic here.
    End Sub
End Class
```

This approach is typically used in LINQ to SQL to override default methods for `Insert`, `Update`, `Delete`, and to validate properties during object life-cycle events.

For more information, see Partial Methods (Visual Basic) or partial (Method) (C# Reference) (C#).

Example

Description

The following example shows `ExampleClass` first as it might be defined by a code-generating tool such as SQLMetal, and then how you might implement only one of the two methods.

Code

```
// Code-generating tool defines a partial class, including
// two partial methods.
partial class ExampleClass
{
    partial void onFindingMaxOutput();
    partial void onFindingMinOutput();
}

// Developer implements one of the partial methods. Compiler
// discards the signature of the other method.
partial class ExampleClass
{
    partial void onFindingMaxOutput()
    {
        Console.WriteLine("Maximum has been found.");
    }
}
```

```
' Code-generating tool defines a partial class, including
' two partial methods.
Partial Class ExampleClass
    Partial Private Sub OnFindingMaxOutput()
    End Sub

    Partial Private Sub OnFindingMinOutput()
    End Sub

    Sub ExportResults()
        OnFindingMaxOutput()
        OnFindingMinOutput()
    End Sub
End Class

' Developer implements one of the partial methods. Compiler
' discards the other method.
Class ExampleClass
    Private Sub OnFindingMaxOutput()
        Console.WriteLine("Maximum has been found.")
    End Sub
End Class
```

Example

Description

The following example uses the relationship between `Shipper` and `Order` entities. Note among the methods the partial methods, `InsertShipper` and `DeleteShipper`. These methods override the default partial methods supplied by LINQ to SQL mapping.

Code

```
public static int LoadOrdersCalled = 0;
private IEnumerable<Order> LoadOrders(Shipper shipper)
{
    LoadOrdersCalled++;
    return this.Orders.Where(o => o.ShipVia == shipper.ShipperID);
}

public static int LoadShipperCalled = 0;
private Shipper LoadShipper(Order order)
{
    LoadShipperCalled++;
    return this.Shippers.Single(s => s.ShipperID == order.ShipVia);
}

public static int InsertShipperCalled = 0;
partial void InsertShipper(Shipper shipper)
{
    InsertShipperCalled++;
    // Call a Web service to perform an insert operation.
    InsertShipperService(shipper);
}

public static int UpdateShipperCalled = 0;
private void UpdateShipper(Shipper original, Shipper current)
{
    Shipper shipper = new Shipper();
    UpdateShipperCalled++;
    // Call a Web service to update shipper.
    InsertShipperService(shipper);
}

public static bool DeleteShipperCalled;
partial void DeleteShipper(Shipper shipper)
{
    DeleteShipperCalled = true;
}
```

```
Public Shared LoadOrdersCalled As Integer = 0
Private Function LoadOrders(ByVal shipper As Shipper) As  _
    IEnumerable(Of Order)
    LoadOrdersCalled += 1
    Return Me.Orders.Where(Function(o) o.ShipVia = _
        shipper.ShipperID)
End Function

Public Shared LoadShipperCalled As Integer = 0
Private Function LoadShipper(ByVal order As Order) As Shipper
    LoadShipperCalled += 1
    Return Me.Shippers.Single(Function(s) s.ShipperID = _
        order.ShipVia)
End Function

Public Shared InsertShipperCalled As Integer = 0
Private Sub InsertShipper(ByVal instance As Shipper)
    InsertShipperCalled += 1
    ' Call a Web service to perform an insert operation.
    InsertShipperService(shpr:=Nothing)
End Sub

Public Shared UpdateShipperCalled As Integer = 0
Private Sub UpdateShipper(ByVal original As Shipper, ByVal current _
    As Shipper)
    UpdateShipperCalled += 1
    ' Call a Web service to update shipper.
    InsertShipperService(shpr:=Nothing)
End Sub

Public Shared DeleteShipperCalled As Boolean
Private Sub DeleteShipper(ByVal instance As Shipper)
    DeleteShipperCalled = True
End Sub
```

See Also

Making and Submitting Data Changes
Customizing Insert, Update, and Delete Operations

Data Binding

5/1/2017 • 6 min to read • Edit Online

LINQ to SQL supports binding to common controls, such as grid controls. Specifically, LINQ to SQL defines the basic patterns for binding to a data grid and handling master-detail binding, both with regard to display and updating.

Underlying Principle

LINQ to SQL translates LINQ queries to SQL for execution on a database. The results are strongly typed `IEnumerable`. Because these objects are ordinary common language runtime (CLR) objects, ordinary object data binding can be used to display the results. On the other hand, change operations (inserts, updates, and deletes) require additional steps.

Operation

Implicitly binding to Windows Forms controls is accomplished by implementing IListSource. Data sources generic Table<TEntity> (`Table<T>` in C# or `Table(Of T)` in Visual Basic) and generic `DataQuery` have been updated to implement IListSource. User interface (UI) data-binding engines (Windows Forms and Windows Presentation Foundation) both test whether their data source implements IListSource. Therefore, writing a direct affectation of a query to a data source of a control implicitly calls LINQ to SQL collection generation, as in the following example:

```
DataGrid dataGrid1 = new DataGrid();
DataGrid dataGrid2 = new DataGrid();
DataGrid dataGrid3 = new DataGrid();

var custQuery =
    from cust in db.Customers
    select cust;
dataGrid1.DataSource = custQuery;
dataGrid2.DataSource = custQuery;
dataGrid2.DataMember = "Orders";

BindingSource bs = new BindingSource();
bs.DataSource = custQuery;
dataGrid3.DataSource = bs;
```

```
Dim dataGrid1 As New DataGrid()
Dim dataGrid2 As New DataGrid()
Dim dataGrid3 As New DataGrid()

Dim custQuery = _
    From cust In db.Customers _
    Select cust

dataGrid1.DataSource = custQuery
dataGrid2.DataSource = custQuery
dataGrid2.DataMember = "Orders"

Dim bs = _
    New BindingSource()
bs.DataSource = custQuery
dataGrid3.DataSource = bs
```

The same occurs with Windows Presentation Foundation:

```
ListView listView1 = new ListView();
var custQuery2 =
    from cust in db.Customers
    select cust;

ListViewItem ItemsSource = new ListViewItem();
ItemsSource = (ListViewItem)custQuery2;
```

```
Dim listView1 As New ListView()
Dim custQuery2 = _
From cust In db.Customers _
Select cust

Dim ItemsSource As New ListViewItem
ItemsSource = custQuery2
```

Collection generations are implemented by generic Table<TEntity> and generic `DataQuery` in GetList.

IListSource Implementation

LINQ to SQL implements IListSource in two locations:

- The data source is a Table<TEntity>: LINQ to SQL browses the table to fill a `DataBindingList` collection that keeps a reference on the table.
- The data source is an IQueryable<T>. There are two scenarios:
    - If LINQ to SQL finds the underlying Table<TEntity> from the IQueryable<T>, the source allows for edition and the situation is the same as in the first bullet point.
    - If LINQ to SQL cannot find the underlying Table<TEntity>, the source does not allow for edition (for example, `groupby`). LINQ to SQL browses the query to fill a generic `SortableBindingList`, which is a simple BindingList<T> that implements the sorting feature for T entities for a given property.

Specialized Collections

For many features described earlier in this document, BindingList<T> has been specialized to some different classes. These classes are generic `SortableBindingList` and generic `DataBindingList`. Both are declared as internal.

Generic SortableBindingList

This class inherits from BindingList<T>, and is a sortable version of BindingList<T>. Sorting is an in-memory solution and never contacts the database itself. BindingList<T> implements IBindingList but does not support sorting by default. However, BindingList<T> implements IBindingList with virtual *core* methods. You can easily override these methods. Generic `SortableBindingList` overrides SupportsSortingCore, SortPropertyCore, SortDirectionCore, and ApplySortCore. `ApplySortCore` is called by ApplySort and sorts the list of T items for a given property.

An exception is raised if the property does not belong to T.

To achieve sorting, LINQ to SQL creates a generic `SortableBindingList.PropertyComparer` class that inherits from generic IComparer.Compare and implements a default comparer for a given type T, a `PropertyDescriptor`, and a direction. This class dynamically creates a `Comparer` of T where T is the `PropertyType` of the `PropertyDescriptor`. Then, the default comparer is retrieved from the static generic `Comparer`. A default instance is obtained by using reflection.

Generic `SortableBindingList` is also the base class for `DataBindingList`. Generic `SortableBindingList` offers two virtual methods for suspending or resuming items add/remove tracking. Those two methods can be used for base features such as sorting, but will really be implemented by upper classes like generic `DataBindingList`.

Generic DataBindingList

This class inherits from generic `SortableBindingLIst`. Generic `DataBindingList` keeps a reference on the underlying generic `Table` of the generic `IQueryable` used for the initial filling of the collection. Generic `DatabindingList` adds tracking for item add/remove to the collection by overriding `InsertItem` () and `RemoveItem` (). It also implements the abstract suspend/resume tracking feature to make tracking conditional. This feature makes generic `DataBindingList` take advantage of all the polymorphic usage of the tracking feature of the parent classes.

Binding to EntitySets

Binding to `EntitySet` is a special case because `EntitySet` is already a collection that implements IBindingList. LINQ to SQL adds sorting and canceling (ICancelAddNew) support. An `EntitySet` class uses an internal list to store entities. This list is a low-level collection based on a generic array, the generic `ItemList` class.

Adding a Sorting Feature

Arrays offer a sort method (`Array.Sort()`) that you can be used with a `Comparer` of T. LINQ to SQL uses the generic `SortableBindingList.PropertyComparer` class described earlier in this topic to obtain this `Comparer` for the property and the direction to be sorted on. An `ApplySort` method is added to generic `ItemList` to call this feature.

On the `EntitySet` side, you now have to declare sorting support:

- SupportsSorting returns `true`.
- ApplySort calls `entities.ApplySort()` and then `OnListChanged()`.
- SortDirection and SortProperty properties expose the current sorting definition, which is stored in local members.

When you use a System.Windows.Forms.BindingSource and bind an EntitySet<TEntity> to the System.Windows.Forms.BindingSource.DataSource, you must call EntitySet<Tentity>.GetNewBindingList to update BindingSource.List.

If you use a System.Windows.Forms.BindingSource and set the BindingSource.DataMember property and set BindingSource.DataSource to a class that has a property named in the BindingSource.DataMember that exposes the EntitySet<TEntity>, you don’t have to call EntitySet<Tentity>.GetNewBindingList to update the BindingSource.List but you lose Sorting capability.

Caching

LINQ to SQL queries implement GetList. When the Windows Forms BindingSource class meets this interface, it calls GetList() threes time for a single connection. To work around this situation, LINQ to SQL implements a cache per instance to store and always return the same generated collection.

Cancellation

IBindingList defines an AddNew method that is used by controls to create a new item from a bound collection. The `DataGridView` control shows this feature very well when the last visible row contains a star in its header. The star shows you that you can add a new item.

In addition to this feature, a collection can also implement ICancelAddNew. This feature allows for the controls to cancel or validate that the new edited item has been validated or not.

ICancelAddNew is implemented in all LINQ to SQL databound collections (generic `SortableBindingList` and generic `EntitySet`). In both implementations the code performs as follows:

- Lets items be inserted and then removed from the collection.
- Does not track changes as long as the UI does not commit the edition.
- Does not track changes as long as the edition is canceled (CancelNew).
- Allows tracking when the edition is committed (EndNew).
- Lets the collection behave normally if the new item does not come from AddNew.

Troubleshooting

This section calls out several items that might help troubleshoot your LINQ to SQL data binding applications.

- You must use properties; using only fields is not sufficient. Windows Forms require this usage.
- By default, `image`, `varbinary`, and `timestamp` database types map to byte array. Because `ToString()` is not supported in this scenario, these objects cannot be displayed.
- A class member mapped to a primary key has a setter, but LINQ to SQL does not support object identity change. Therefore, the primary/unique key that is used in mapping cannot be updated in the database. A change in the grid causes an exception when you call SubmitChanges.
- If an entity is bound in two separate grids (for example, one master and another detail), a `Delete` in the master grid is not propagated to the detail grid.

See Also

Background Information

Inheritance Support

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL supports *single-table mapping*. In other words, a complete inheritance hierarchy is stored in a single database table. The table contains the flattened union of all possible data columns for the whole hierarchy. (A union is the result of combining two tables into one table that has the rows that were present in either of the original tables.) Each row has nulls in the columns that do not apply to the type of the instance represented by the row.

The single-table mapping strategy is the simplest representation of inheritance and provides good performance characteristics for many different categories of queries.

To implement this mapping in LINQ to SQL, you must specify the attributes and attribute properties on the root class of the inheritance hierarchy. For more information, see How to: Map Inheritance Hierarchies.

Developers using Visual Studio can also use the Object Relational Designer to map inheritance hierarchies.

See Also

Background Information

Local Method Calls

5/1/2017 • 1 min to read • Edit Online

A local method call is one that is executed within the object model. A remote method call is one that LINQ to SQL translates to SQL and transmits to the database engine for execution. Local method calls are needed when LINQ to SQL cannot translate the call into SQL. Otherwise, an InvalidOperationException is thrown.

Example 1

In the following example, an `Order` class is mapped to the Orders table in the Northwind sample database. A local instance method has been added to the class.

In Query 1, the constructor for the `Order` class is executed locally. In Query 2, if LINQ to SQL tried to translate `LocalInstanceMethod()` into SQL, the attempt would fail and an InvalidOperationException exception would be thrown. But because LINQ to SQL provides support for local method calls, Query2 will not throw an exception.

```
// Query 1.
var q1 =
    from ord in db.Orders
    where ord.EmployeeID == 9
    select ord;

foreach (var ordObj in q1)
{
    Console.WriteLine("{0}, {1}", ordObj.OrderID,
        ordObj.ShipVia.Value);
}
```

```
' Query 1.
Dim q0 = _
    From ord In db.Orders _
    Where ord.EmployeeID = 9 _
    Select ord

For Each ordObj In q0
    Console.WriteLine("{0}, {1}", ordObj.OrderID, _
        ordObj.ShipVia.Value)
Next
```

```
// Query 2.
public int LocalInstanceMethod(int x)
{
    return x + 1;
}

void q2()
{
    var q2 =
    from ord in db.Orders
    where ord.EmployeeID == 9
    select new
    {
        member0 = ord.OrderID,
        member1 = ord.LocalInstanceMethod(ord.ShipVia.Value)
    };
}
```

```
' Query 2.
Public Function LocalInstanceMethod(ByVal x As Integer) As Integer
    Return x + 1
End Function

Sub q2()
    Dim db As New Northwnd("")
    Dim q2 = _
    From ord In db.Orders _
    Where ord.EmployeeID = 9 _
    Select member0 = ord.OrderID, member1 = ord.LocalInstanceMethod(ord.ShipVia.Value)
End Sub
```

See Also

Background Information

N-Tier and Remote Applications with LINQ to SQL

5/1/2017 • 1 min to read • Edit Online

You can create n-tier or multitier applications that use LINQ to SQL. Typically, the LINQ to SQL data context, entity classes, and query construction logic are located on the middle tier as the data access layer (DAL). Business logic and any non-persistent data can be implemented completely in partial classes and methods of entities and the data context, or it can be implemented in separate classes.

The client or presentation layer calls methods on the middle-tier's remote interface, and the DAL on that tier will execute queries or stored procedures that are mapped to DataContext methods. The middle tier returns the data to clients typically as XML representations of entities or proxy objects.

On the middle tier, entities are created by the data context, which tracks their state, and manages deferred loading from and submission of changes to the database. These entities are "attached" to the `DataContext`. However, after the entities are sent to another tier through serialization, they become detached, which means the `DataContext` is no longer tracking their state. Entities that the client sends back for updates must be reattached to the data context before LINQ to SQL can submit the changes to the database. The client is responsible for providing original values and/or timestamps back to the middle tier if those are required for optimistic concurrency checks.

In ASP.NET applications, the LinqDataSource manages most of this complexity. For more information, see NIB: LinqDataSource Web Server Control Overview.

Additional Resources

For more information about how to implement n-tier applications that use LINQ to SQL, see the following topics:

- LINQ to SQL N-Tier with ASP.NET
- LINQ to SQL N-Tier with Web Services
- LINQ to SQL with Tightly-Coupled Client-Server Applications
- Implementing N-Tier Business Logic
- Data Retrieval and CUD Operations in N-Tier Applications (LINQ to SQL)

For more information about n-tier applications that use ADO.NET DataSets, see Work with datasets in n-tier applications.

See Also

Background Information

LINQ to SQL N-Tier with ASP.NET

5/1/2017 • 1 min to read • Edit Online

In ASP.NET applications that use LINQ to SQL, you use the LinqDataSource Web server control. The control handles most of the logic that it must have to query against LINQ to SQL, pass the data to the browser, retrieve it, and submit it to the LINQ to SQL DataContext which then updates the database. You just configure the control in the markup, and the control handles all the data transfer between LINQ to SQL and the browser. Because the control handles the interactions with the presentation tier, and LINQ to SQL handles the communication with the data tier, your main focus in ASP.NET multitier applications is on writing your custom business logic.

For more information about `LINQDataSource`, see NIB: LinqDataSource Web Server Control Overview.

See Also

N-Tier and Remote Applications with LINQ to SQL

LINQ to SQL N-Tier with Web Services

5/10/2017 • 4 min to read • Edit Online

LINQ to SQL is designed especially for use on the middle tier in a loosely-coupled data access layer (DAL) such as a Web service. If the presentation tier is an ASP.NET Web page, then you use the LinqDataSource Web server control to manage the data transfer between the user interface and LINQ to SQL on the middle-tier. If the presentation tier is not an ASP.NET page, then both the middle-tier and the presentation tier must do some additional work to manage the serialization and deserialization of data.

Setting up LINQ to SQL on the Middle Tier

In a Web service or n-tier application, the middle tier contains the data context and the entity classes. You can create these classes manually, or by using either SQLMetal.exe or the Object Relational Designer as described elsewhere in the documentation. At design time, you have the option to make the entity classes serializable. For more information, see How to: Make Entities Serializable. Another option is to create a separate set of classes that encapsulate the data to be serialized, and then project into those serializable types when you return data in your LINQ queries.

You then define the interface with the methods that the clients will call to retrieve, insert and update data. The interface methods wrap your LINQ queries. You can use any kind of serialization mechanism to handle the remote method calls and the serialization of data. The only requirement is that if you have cyclic or bi-directional relationships in your object model, such as that between Customers and Orders in the standard Northwind object model, then you must use a serializer that supports it. The Windows Communication Foundation (WCF) DataContractSerializer supports bi-directional relationships but the XmlSerializer that is used with non-WCF Web services does not. If you select to use the XmlSerializer, then you must make sure that your object model has no cyclic relationships.

For more information about Windows Communication Foundation, see Windows Communication Foundation Services and WCF Data Services in Visual Studio.

Implement your business rules or other domain-specific logic by using the partial classes and methods on the DataContext and entity classes to hook into LINQ to SQL runtime events. For more information, see Implementing N-Tier Business Logic.

Defining the Serializable Types

The client or presentation tier must have type definitions for the classes that it will be receiving from the middle tier. Those types may be the entity classes themselves, or special classes that wrap only certain fields from the entity classes for remoting. In any case, LINQ to SQL is completely unconcerned about how the presentation tier acquires those type definitions. For example, the presentation tier could use WCF to generate the types automatically, or it could have a copy of a DLL in which those types are defined, or it could just define its own versions of the types.

Retrieving and Inserting Data

The middle tier defines an interface that specifies how the presentation tier accesses the data. For example `GetProductByID(int productID)`, or `GetCustomers()`. On the middle tier, the method body typically creates a new instance of the DataContext, executes a query against one or more of its table. The middle tier then returns the result as an IEnumerable<T>, where `T` is either an entity class or another type that is used for serialization. The presentation tier never sends or receives query variables directly to or from the middle tier. The two tiers exchange values, objects, and collections of concrete data. After it has received a collection, the presentation tier can use LINQ to Objects to query it if necessary.

When inserting data, the presentation tier can construct a new object and send it to the middle tier, or it can have the middle tier construct the object based on values that it provides. In general, retrieving and inserting data in n-tier applications does not differ much from the process in 2-tier applications. For more information, see Querying the Database and Making and Submitting Data Changes.

Tracking Changes for Updates and Deletes

LINQ to SQL supports optimistic concurrency based on timestamps (also named RowVersions) and on original values. If the database tables have timestamps, then updates and deletions require little extra work on either the middle-tier or presentation tier. However, if you must use original values for optimistic concurrency checks, then the presentation tier is responsible for tracking those values and sending them back when it makes updates. This is because changes that were made to entities on the presentation tier are not tracked on the middle tier. In fact, the original retrieval of an entity, and the eventual update made to it are typically performed by two completely separate instances of the DataContext.

The greater the number of changes that the presentation tier makes, the more complex it becomes to track those changes and package them back to the middle tier. The implementation of a mechanism for communicating changes is completely up to the application. The only requirement is that LINQ to SQL must be given those original values that are required for optimistic concurrency checks.

For more information, see Data Retrieval and CUD Operations in N-Tier Applications (LINQ to SQL).

See Also

N-Tier and Remote Applications with LINQ to SQL
NIB: LinqDataSource Web Server Control Overview

LINQ to SQL with Tightly-Coupled Client-Server Applications

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL can be used on the middle tier with tightly-coupled Smart Clients on the presentation layer. In scenarios that involve read-only data access, no optimistic concurrency checks, or optimistic concurrency with timestamps, there is not much more complexity than with non-remote scenarios. However, when a database requires optimistic concurrency checks with original values, LINQ to SQL does not provide the level of support for round-tripping of data that you find in DataSets. However, a LINQ to SQL middle tier can exchange data with clients on any platform.

LINQ to SQL in Visual Studio 2008 provides no infrastructure for tracking entity state after they have been serialized to a client. LINQ to SQL enables service-oriented architectures where interactions between the data and presentation layers are small and relatively atomic, but does not perform any round-tripping of original values. Therefore, if you want to use a tightly-coupled Smart Client with LINQ to SQL, and a database uses optimistic concurrency with original values, you will have to implement your own mechanism for communicating changes between the presentation tier and middle tier. It is up to the system designer to decide whether it makes sense to do this bit of extra work in exchange for the benefits that LINQ to SQL provides on the middle tier. On the other hand, if the database has timestamps, then there is no need for custom change-tracking logic.

See Also

N-Tier and Remote Applications with LINQ to SQL
LINQ to SQL N-Tier with Web Services
Work with datasets in n-tier applications

Implementing Business Logic (LINQ to SQL)

5/10/2017 • 5 min to read • Edit Online

The term "business logic" in this topic refers to any custom rules or validation tests that you apply to data before it is inserted, updated or deleted from the database. Business logic is also sometimes referred to as "business rules" or "domain logic." In n-tier applications it is typically designed as a logical layer so that it can be modified independently of the presentation layer or data access layer. The business logic can be invoked by the data access layer before or after any update, insertion, or deletion of data in the database.

The business logic can be as simple as a schema validation to make sure that the type of the field is compatible with the type of the table column. Or it can consist of a set of objects that interact in arbitrarily complex ways. The rules may be implemented as stored procedures on the database or as in-memory objects. However the business logic is implemented, LINQ to SQL enables you use partial classes and partial methods to separate the business logic from the data access code.

How LINQ to SQL Invokes Your Business Logic

When you generate an entity class at design time, either manually or by using the Object Relational Designer or SQLMetal, it is defined as a partial class. This means that, in a separate code file, you can define another part of the entity class that contains your custom business logic. At compile time, the two parts are merged into a single class. But if you have to regenerate your entity classes by using the Object Relational Designer or SQLMetal, you can do so and your part of the class will not be modified.

The partial classes that define entities and the DataContext contain partial methods. These are the extensibility points that you can use to apply your business logic before and after any update, insert, or delete for an entity or entity property. Partial methods can be thought of as compile-time events. The code-generator defines a method signature and calls the methods in the get and set property accessors, the `DataContext` constructor, and in some cases behind the scenes when SubmitChanges is called. However, if you do not implement a particular partial method, then all the references to it and the definition are removed at compile time.

In the implementing definition that you write in your separate code file, you can perform whatever custom logic is required. You can use your partial class itself as your domain layer, or you can call from your implementing definition of the partial method into a separate object or objects. Either way, your business logic is cleanly separated from both your data access code and your presentation layer code.

A Closer Look at the Extensibility Points

The following example shows part of the code generated by the Object Relational Designer for the `DataContext` class that has two tables: `Customers` and `Orders`. Note that Insert, Update, and Delete methods are defined for each table in the class.

```
Partial Public Class Northwnd
    Inherits System.Data.Linq.DataContext

    Private Shared mappingSource As _
        System.Data.Linq.Mapping.MappingSource = New _
        AttributeMappingSource

    #Region "Extensibility Method Definitions"
    Partial Private Sub OnCreated()
    End Sub
    Partial Private Sub InsertCustomer(instance As Customer)
    End Sub
    Partial Private Sub UpdateCustomer(instance As Customer)
    End Sub
    Partial Private Sub DeleteCustomer(instance As Customer)
    End Sub
    Partial Private Sub InsertOrder(instance As [Order])
    End Sub
    Partial Private Sub UpdateOrder(instance As [Order])
    End Sub
    Partial Private Sub DeleteOrder(instance As [Order])
    End Sub
    #End Region
```

```
public partial class MyNorthWindDataContext : System.Data.Linq.DataContext
    {
        private static System.Data.Linq.Mapping.MappingSource mappingSource = new AttributeMappingSource();

        #region Extensibility Method Definitions
        partial void OnCreated();
        partial void InsertCustomer(Customer instance);
        partial void UpdateCustomer(Customer instance);
        partial void DeleteCustomer(Customer instance);
        partial void InsertOrder(Order instance);
        partial void UpdateOrder(Order instance);
        partial void DeleteOrder(Order instance);
        #endregion
```

If you implement the Insert, Update and Delete methods in your partial class, the LINQ to SQL runtime will call them instead of its own default methods when SubmitChanges is called. This enables you to override the default behavior for create / read / update / delete operations. For more information, see Walkthrough: Customizing the insert, update, and delete behavior of entity classes.

The `OnCreated` method is called in the class constructor.

```
Public Sub New(ByVal connection As String)
    MyBase.New(connection, mappingSource)
    OnCreated()
End Sub
```

```
public MyNorthWindDataContext(string connection) :
            base(connection, mappingSource)
        {
            OnCreated();
        }
```

The entity classes have three methods that are called by the LINQ to SQL runtime when the entity is created, loaded, and validated (when `SubmitChanges` is called). The entity classes also have two partial methods for each property, one that is called before the property is set, and one that is called after. The following code example shows some of the methods generated for the `Customer` class:

```
#Region "Extensibility Method Definitions"
    Partial Private Sub OnLoaded()
    End Sub
    Partial Private Sub OnValidate(action As _
        System.Data.Linq.ChangeAction)
    End Sub
    Partial Private Sub OnCreated()
    End Sub
    Partial Private Sub OnCustomerIDChanging(value As String)
    End Sub
    Partial Private Sub OnCustomerIDChanged()
    End Sub
    Partial Private Sub OnCompanyNameChanging(value As String)
    End Sub
    Partial Private Sub OnCompanyNameChanged()
    End Sub
' ...Additional Changing/Changed methods for each property.
```

```
#region Extensibility Method Definitions
    partial void OnLoaded();
    partial void OnValidate();
    partial void OnCreated();
    partial void OnCustomerIDChanging(string value);
    partial void OnCustomerIDChanged();
    partial void OnCompanyNameChanging(string value);
    partial void OnCompanyNameChanged();
// ...additional Changing/Changed methods for each property
```

The methods are called in the property set accessor as shown in the following example for the `CustomerID` property:

```
Public Property CustomerID() As String
    Set
        If (String.Equals(Me._CustomerID, value) = False) Then
            Me.OnCustomerIDChanging(value)
            Me.SendPropertyChanging()
            Me._CustomerID = value
            Me.SendPropertyChanged("CustomerID")
            Me.OnCustomerIDChanged()
        End If
    End Set
End Property
```

```
public string CustomerID
{
    set
    {
        if ((this._CustomerID != value))
        {
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
            this.OnCustomerIDChanged();
        }
     }
}
```

In your part of the class, you write an implementing definition of the method. In Visual Studio, after you type `partial` you will see IntelliSense for the method definitions in the other part of the class.

```
Partial Public Class Customer
    Private Sub OnCustomerIDChanging(value As String)
        ' Perform custom validation logic here.
    End Sub
End Class
```

```
partial class Customer
    {
        partial void OnCustomerIDChanging(string value)
        {
            //Perform custom validation logic here.
        }
    }
```

For more information about how to add business logic to your application by using partial methods, see the following topics:

How to: Add validation to entity classes

Walkthrough: Customizing the insert, update, and delete behavior of entity classes

Walkthrough: Adding Validation to Entity Classes

See Also

Partial Classes and Methods
Partial Methods
LINQ to SQL Tools in Visual Studio
SqlMetal.exe (Code Generation Tool)

Data Retrieval and CUD Operations in N-Tier Applications (LINQ to SQL)

5/1/2017 • 12 min to read • Edit Online

When you serialize entity objects such as Customers or Orders to a client over a network, those entities are detached from their data context. The data context no longer tracks their changes or their associations with other objects. This is not an issue as long as the clients are only reading the data. It is also relatively simple to enable clients to add new rows to a database. However, if your application requires that clients be able to update or delete data, then you must attach the entities to a new data context before you call System.Data.Linq.DataContext.SubmitChanges. In addition, if you are using an optimistic concurrency check with original values, then you will also need a way to provide the database both the original entity and the entity as modified. The `Attach` methods are provided to enable you to put entities into a new data context after they have been detached.

Even if you are serializing proxy objects in place of the LINQ to SQL entities, you still have to construct an entity on the data access layer (DAL), and attach it to a new System.Data.Linq.DataContext, in order to submit the data to the database.

LINQ to SQL is completely indifferent about how entities are serialized. For more information about how to use the Object Relational Designer and SQLMetal tools to generate classes that are serializable by using Windows Communication Foundation (WCF), see How to: Make Entities Serializable.

NOTE

Only call the `Attach` methods on new or deserialized entities. The only way for an entity to be detached from its original data context is for it to be serialized. If you try to attach an undetached entity to a new data context, and that entity still has deferred loaders from its previous data context, LINQ to SQL will thrown an exception. An entity with deferred loaders from two different data contexts could cause unwanted results when you perform insert, update, and delete operations on that entity. For more information about deferred loaders, see Deferred versus Immediate Loading.

Retrieving Data

Client Method Call

The following examples show a sample method call to the DAL from a Windows Forms client. In this example, the DAL is implemented as a Windows Service Library:

```vb
Private Function GetProdsByCat_Click(ByVal sender As Object, ByVal e _
    As EventArgs)

    ' Create the WCF client proxy.
    Dim proxy As New NorthwindServiceReference.Service1Client

    ' Call the method on the service.
    Dim products As NorthwindServiceReference.Product() = _
        proxy.GetProductsByCategory(1)

    ' If the database uses original values for concurrency checks,
    ' the client needs to store them and pass them back to the
    ' middle tier along with the new values when updating data.

    For Each v As NorthwindClient1.NorthwindServiceReference.Product _
        In products
        ' Persist to a List(Of Product) declared at class scope.
        ' Additional change-tracking logic is the responsibility
        ' of the presentation tier and/or middle tier.
        originalProducts.Add(v)
    Next

    ' (Not shown) Bind the products list to a control
    ' and/or perform whatever processing is necessary.
End Function
```

```csharp
private void GetProdsByCat_Click(object sender, EventArgs e)
{
    // Create the WCF client proxy.
    NorthwindServiceReference.Service1Client proxy =
    new NorthwindClient.NorthwindServiceReference.Service1Client();

    // Call the method on the service.
    NorthwindServiceReference.Product[] products =
    proxy.GetProductsByCategory(1);

    // If the database uses original values for concurrency checks,
    // the client needs to store them and pass them back to the
    // middle tier along with the new values when updating data.
    foreach (var v in products)
    {
        // Persist to a list<Product> declared at class scope.
        // Additional change-tracking logic is the responsibility
        // of the presentation tier and/or middle tier.
        originalProducts.Add(v);
    }

    // (Not shown) Bind the products list to a control
    // and/or perform whatever processing is necessary.
    }
```

Middle Tier Implementation

The following example shows an implementation of the interface method on the middle tier. The following are the two main points to note:

- The DataContext is declared at method scope.
- The method returns an IEnumerable collection of the actual results. The serializer will execute the query to send the results back to the client/presentation tier. To access the query results locally on the middle tier, you can force execution by calling `ToList` or `ToArray` on the query variable. You can then return that list or array as an `IEnumerable`.

```
Public Function GetProductsByCategory(ByVal categoryID As Integer) _
    As IEnumerable(Of Product)

    Dim db As New NorthwindClasses1DataContext(connectionString)
    Dim productQuery = _
    From prod In db.Products _
    Where prod.CategoryID = categoryID _
    Select prod

    Return productQuery.AsEnumerable()

End Function
```

```
public IEnumerable<Product> GetProductsByCategory(int categoryID)
{
    NorthwindClasses1DataContext db =
    new NorthwindClasses1DataContext(connectionString);

    IEnumerable<Product> productQuery =
    from prod in db.Products
    where prod.CategoryID == categoryID
    select prod;

    return productQuery.AsEnumerable();
}
```

An instance of a data context should have a lifetime of one "unit of work." In a loosely-coupled environment, a unit of work is typically small, perhaps one optimistic transaction, including a single call to `SubmitChanges`. Therefore, the data context is created and disposed at method scope. If the unit of work includes calls to business rules logic, then generally you will want to keep the `DataContext` instance for that whole operation. In any case, `DataContext` instances are not intended to be kept alive for long periods of time across arbitrary numbers of transactions.

This method will return Product objects but not the collection of Order_Detail objects that are associated with each Product. Use the DataLoadOptions object to change this default behavior. For more information, see How to: Control How Much Related Data Is Retrieved.

Inserting Data

To insert a new object, the presentation tier just calls the relevant method on the middle tier interface, and passes in the new object to insert. In some cases, it may be more efficient for the client to pass in only some values and have the middle tier construct the full object.

Middle Tier Implementation

On the middle tier, a new DataContext is created, the object is attached to the DataContext by using the InsertOnSubmit method, and the object is inserted when SubmitChanges is called. Exceptions, callbacks, and error conditions can be handled just as in any other Web service scenario.

```
' No call to Attach is necessary for inserts.
Public Sub InsertOrder(ByVal o As Order)

    Dim db As New NorthwindClasses1DataContext(connectionString)
    db.Orders.InsertOnSubmit(o)

    ' Exception handling not shown.
    db.SubmitChanges()

End Sub
```

```
// No call to Attach is necessary for inserts.
    public void InsertOrder(Order o)
    {
        NorthwindClasses1DataContext db = new NorthwindClasses1DataContext(connectionString);
        db.Orders.InsertOnSubmit(o);

        // Exception handling not shown.
        db.SubmitChanges();
    }
```

Deleting Data

To delete an existing object from the database, the presentation tier calls the relevant method on the middle tier interface, and passes in its copy that includes original values of the object to be deleted.

Delete operations involve optimistic concurrency checks, and the object to be deleted must first be attached to the new data context. In this example, the `Boolean` parameter is set to `false` to indicate that the object does not have a timestamp (RowVersion). If your database table does generate timestamps for each record, then concurrency checks are much simpler, especially for the client. Just pass in either the original or modified object and set the `Boolean` parameter to `true`. In any case, on the middle tier it is typically necessary to catch the ChangeConflictException. For more information about how to handle optimistic concurrency conflicts, see Optimistic Concurrency: Overview.

When deleting entities that have foreign key constraints on associated tables, you must first delete all the objects in its EntitySet<TEntity> collections.

```
' Attach is necessary for deletes.
Public Sub DeleteOrder(ByVal order As Order)
    Dim db As New NorthwindClasses1DataContext(connectionString)

    db.Orders.Attach(order, False)
    ' This will throw an exception if the order has order details.
    db.Orders.DeleteOnSubmit(order)

    Try
        ' ConflictMode is an optional parameter.
        db.SubmitChanges(ConflictMode.ContinueOnConflict)

    Catch ex As ChangeConflictException
        ' Get conflict information, and take actions
        ' that are appropriate for your application.
        ' See MSDN Article "How to: Manage Change
        ' Conflicts (LINQ to SQL).

    End Try
End Sub
```

```
// Attach is necessary for deletes.
public void DeleteOrder(Order order)
{
    NorthwindClasses1DataContext db = new NorthwindClasses1DataContext(connectionString);

    db.Orders.Attach(order, false);
    // This will throw an exception if the order has order details.
    db.Orders.DeleteOnSubmit(order);
    try
    {
        // ConflictMode is an optional parameter.
        db.SubmitChanges(ConflictMode.ContinueOnConflict);
    }
    catch (ChangeConflictException e)
    {
       // Get conflict information, and take actions
       // that are appropriate for your application.
       // See MSDN Article How to: Manage Change Conflicts (LINQ to SQL).
    }
}
```

Updating Data

LINQ to SQL supports updates in these scenarios involving optimistic concurrency:

- Optimistic concurrency based on timestamps or RowVersion numbers.
- Optimistic concurrency based on original values of a subset of entity properties.
- Optimistic concurrency based on the complete original and modified entities.

You can also perform updates or deletes on an entity together with its relations, for example a Customer and a collection of its associated Order objects. When you make modifications on the client to a graph of entity objects and their child (`EntitySet`) collections, and the optimistic concurrency checks require original values, the client must provide those original values for each entity and EntitySet<TEntity> object. If you want to enable clients to make a set of related updates, deletes, and insertions in a single method call, you must provide the client a way to indicate what type of operation to perform on each entity. On the middle tier, you then must call the appropriate Attach method and then InsertOnSubmit, DeleteAllOnSubmit, or InsertOnSubmit (without `Attach`, for insertions) for each entity before you call SubmitChanges. Do not retrieve data from the database as a way to obtain original values before you try updates.

For more information about optimistic concurrency, see Optimistic Concurrency: Overview. For detailed information about resolving optimistic concurrency change conflicts, see How to: Manage Change Conflicts.

The following examples demonstrate each scenario:

Optimistic concurrency with timestamps

```
' Assume that "customer" has been sent by client.
' Attach with "true" to say this is a modified entity
' and it can be checked for optimistic concurrency
' because it has a column that is marked with the
' "RowVersion" attribute.

db.Customers.Attach(customer, True)

Try
    ' Optional: Specify a ConflictMode value
    ' in call to SubmitChanges.
    db.SubmitChanges()
Catch ex As ChangeConflictException
    ' Handle conflict based on options provided.
    ' See MSDN article "How to: Manage Change
    ' Conflicts (LINQ to SQL)".
End Try
```

```
// Assume that "customer" has been sent by client.
// Attach with "true" to say this is a modified entity
// and it can be checked for optimistic concurrency because
//  it has a column that is marked with "RowVersion" attribute
db.Customers.Attach(customer, true)
try
{
    // Optional: Specify a ConflictMode value
    // in call to SubmitChanges.
    db.SubmitChanges();
}
catch(ChangeConflictException e)
{
    // Handle conflict based on options provided
    // See MSDN article How to: Manage Change Conflicts (LINQ to SQL).
}
```

With Subset of Original Values

In this approach, the client returns the complete serialized object, together with the values to be modified.

```vb
Public Sub UpdateProductInventory(ByVal p As Product, ByVal _
    unitsInStock As Short?, ByVal unitsOnOrder As Short?)

    Using db As New NorthwindClasses1DataContext(connectionString)
        ' p is the original unmodified product
        ' that was obtained from the database.
        ' The client kept a copy and returns it now.
        db.Products.Attach(p, False)

        ' Now that the original values are in the data context,
        ' apply the changes.
        p.UnitsInStock = unitsInStock
        p.UnitsOnOrder = unitsOnOrder

        Try
            ' Optional: Specify a ConflictMode value
            ' in call to SubmitChanges.
            db.SubmitChanges()

        Catch ex As Exception
            ' Handle conflict based on options provided.
            ' See MSDN article "How to: Manage Change Conflicts
            ' (LINQ to SQL)".
        End Try
    End Using
End Sub
```

```csharp
public void UpdateProductInventory(Product p, short? unitsInStock, short? unitsOnOrder)
{
    using (NorthwindClasses1DataContext db = new NorthwindClasses1DataContext(connectionString))
    {
        // p is the original unmodified product
        // that was obtained from the database.
        // The client kept a copy and returns it now.
        db.Products.Attach(p, false);

        // Now that the original values are in the data context, apply the changes.
        p.UnitsInStock = unitsInStock;
        p.UnitsOnOrder = unitsOnOrder;
        try
        {
             // Optional: Specify a ConflictMode value
             // in call to SubmitChanges.
             db.SubmitChanges();
        }
        catch (ChangeConflictException e)
        {
            // Handle conflict based on provided options.
            // See MSDN article How to: Manage Change Conflicts
            // (LINQ to SQL).
        }
    }
}
```

With Complete Entities

```
Public Sub UpdateProductInfo(ByVal newProd As Product, ByVal _
    originalProd As Product)

    Using db As New NorthwindClasses1DataContext(connectionString)
        db.Products.Attach(newProd, originalProd)

        Try
            ' Optional: Specify a ConflictMode value
            ' in call to SubmitChanges.
            db.SubmitChanges()

        Catch ex As Exception
            ' Handle potential change conflicgt in whatever way
            ' is appropriate for your application.
            ' For more information, see the MSDN article
            ' "How to: Manage Change Conflicts (LINQ to
            ' SQL)".
        End Try

    End Using
End Sub
```

```
public void UpdateProductInfo(Product newProd, Product originalProd)
{
     using (NorthwindClasses1DataContext db = new
        NorthwindClasses1DataContext(connectionString))
     {
         db.Products.Attach(newProd, originalProd);
         try
         {
               // Optional: Specify a ConflictMode value
               // in call to SubmitChanges.
               db.SubmitChanges();
         }
        catch (ChangeConflictException e)
        {
            // Handle potential change conflict in whatever way
            // is appropriate for your application.
            // For more information, see the MSDN article
            // How to: Manage Change Conflicts (LINQ to SQL)/
        }
    }
}
```

To update a collection, call AttachAll instead of `Attach`.

Expected Entity Members

As stated previously, only certain members of the entity object are required to be set before you call the `Attach` methods. Entity members that are required to be set must fulfill the following criteria:

- Be part of the entity's identity.
- Be expected to be modified.
- Be a timestamp or have its UpdateCheck attribute set to something besides `Never`.

If a table uses a timestamp or version number for an optimistic concurrency check, you must set those members before you call Attach. A member is dedicated for optimistic concurrency checking when the IsVersion property is set to true on that Column attribute. Any requested updates will be submitted only if the version number or timestamp values are the same on the database.

A member is also used in the optimistic concurrency check as long as the member does not have UpdateCheck set to `Never`. The default value is `Always` if no other value is specified.

If any one of these required members is missing, a ChangeConflictException is thrown during SubmitChanges ("Row not found or changed").

State

After an entity object is attached to the DataContext instance, the object is considered to be in the `PossiblyModified` state. There are three ways to force an attached object to be considered `Modified`.

1. Attach it as unmodified, and then directly modify the fields.

2. Attach it with the Attach overload that takes current and original object instances. This supplies the change tracker with old and new values so that it will automatically know which fields have changed.

3. Attach it with the Attach overload that takes a second Boolean parameter (set to true). This will tell the change tracker to consider the object modified without having to supply any original values. In this approach, the object must have a version/timestamp field.

For more information, see Object States and Change-Tracking.

If an entity object already occurs in the ID Cache with the same identity as the object being attached, a DuplicateKeyException is thrown.

When you attach with an `IEnumerable` set of objects, a DuplicateKeyException is thrown when an already existing key is present. Remaining objects are not attached.

See Also

N-Tier and Remote Applications with LINQ to SQL
Background Information

Object Identity

5/1/2017 • 3 min to read • Edit Online

Objects in the runtime have unique identities. Two variables that refer to the same object actually refer to the same instance of the object. Because of this fact, changes that you make by way of a path through one variable are immediately visible through the other.

Rows in a relational database table do not have unique identities. Because each row has a unique primary key, no two rows share the same key value. However, this fact constrains only the contents of the database table.

In reality, data is most often brought out of the database and into a different tier, where an application works with it. This is the model that LINQ to SQL supports. When data is brought out of the database as rows, you have no expectation that two rows that represent the same data actually correspond to the same row instances. If you query for a specific customer two times, you get two rows of data. Each row contains the same information.

With objects you expect something very different. You expect that if you ask the DataContext for the same information repeatedly, it will in fact give you the same object instance. You expect this behavior because objects have special meaning for your application and you expect them to behave like objects. You designed them as hierarchies or graphs. You expect to retrieve them as such and not to receive multitudes of replicated instances just because you asked for the same thing more than one time.

In LINQ to SQL, the DataContext manages object identity. Whenever you retrieve a new row from the database, the row is logged in an identity table by its primary key, and a new object is created. Whenever you retrieve that same row, the original object instance is handed back to the application. In this manner the DataContext translates the concept of identity as seen by the database (that is, primary keys) into the concept of identity seen by the language (that is, instances). The application only sees the object in the state that it was first retrieved. The new data, if different, is discarded. For more information, see Retrieving Objects from the Identity Cache.

LINQ to SQL uses this approach to manage the integrity of local objects in order to support optimistic updates. Because the only changes that occur after the object is at first created are those made by the application, the intent of the application is clear. If changes by an outside party have occurred in the interim, they are identified at the time `SubmitChanges()` is called.

NOTE

If the object requested by the query is easily identifiable as one already retrieved, no query is executed. The identity table acts as a cache of all previously retrieved objects.

Examples

Object Caching Example 1

In this example, if you execute the same query two times, you receive a reference to the same object in memory every time.

```
Customer cust1 =
    (from cust in db.Customers
     where cust.CustomerID == "BONAP"
     select cust).First();

Customer cust2 =
    (from cust in db.Customers
     where cust.CustomerID == "BONAP"
     select cust).First();
```

```
Dim cust1 As Customer = _
    (From cust In db.Customers _
    Where cust.CustomerID = "BONAP" _
    Select cust).First()

Dim cust2 As Customer = _
    (From cust In db.Customers _
    Where cust.CustomerID = "BONAP" _
    Select cust).First()
```

Object Caching Example 2

In this example, if you execute different queries that return the same row from the database, you receive a reference to the same object in memory every time.

```
Customer cust1 =
    (from cust in db.Customers
     where cust.CustomerID == "BONAP"
     select cust).First();

Customer cust2 =
    (from ord in db.Orders
     where ord.Customer.CustomerID == "BONAP"
     select ord).First().Customer;
```

```
Dim cust1 As Customer = _
    (From cust In db.Customers _
    Where cust.CustomerID = "BONAP" _
    Select cust).First()

Dim cust2 As Customer = _
    (From ord In db.Orders _
    Where ord.Customer.CustomerID = "BONAP" _
    Select ord).First().Customer
```

See Also

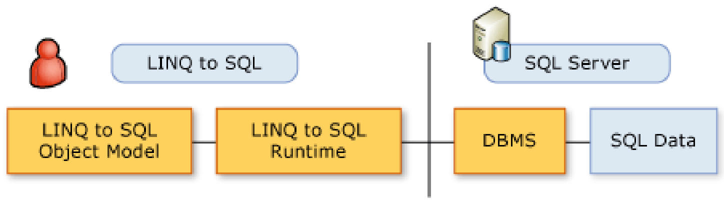
Background Information

The LINQ to SQL Object Model

5/1/2017 • 4 min to read • Edit Online

In LINQ to SQL, an object model expressed in the programming language of the developer is mapped to the data model of a relational database. Operations on the data are then conducted according to the object model.

In this scenario, you do not issue database commands (for example, `INSERT`) to the database. Instead, you change values and execute methods within your object model. When you want to query the database or send it changes, LINQ to SQL translates your requests into the correct SQL commands and sends those commands to the database.



The most fundamental elements in the LINQ to SQL object model and their relationship to elements in the relational data model are summarized in the following table:

LINQ TO SQL OBJECT MODEL	RELATIONAL DATA MODEL
Entity class	Table
Class member	Column
Association	Foreign-key relationship
Method	Stored Procedure or Function

NOTE

The following descriptions assume that you have a basic knowledge of the relational data model and rules.

LINQ to SQL Entity Classes and Database Tables

In LINQ to SQL, a database table is represented by an *entity class*. An entity class is like any other class you might create except that you annotate the class by using special information that associates the class with a database table. You make this annotation by adding a custom attribute (TableAttribute) to your class declaration, as in the following example:

Example

```
[Table(Name = "Customers")]
public class Customerzz
{
    public string CustomerID;
    // ...
    public string City;
}
```

```
<Table(Name:="Customers")> _
Public Class Customer
    Public CustomerID As String
    ' ...
    Public City As String
End Class
```

Only instances of classes declared as tables (that is, entity classes) can be saved to the database.

For more information, see the Table Attribute section of Attribute-Based Mapping.

LINQ to SQL Class Members and Database Columns

In addition to associating classes with tables, you designate fields or properties to represent database columns. For this purpose, LINQ to SQL defines the ColumnAttribute attribute, as in the following example:

Example

```
[Table(Name = "Customers")]
public class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID;
    [Column]
    public string City;
}
```

```
<Table(Name:="Customers")> _
Public Class Customer
    <Column(IsPrimaryKey:=True)> _
    Public CustomerID As String

    <Column()> _
    Public City As String
End Class
```

Only fields and properties mapped to columns are persisted to or retrieved from the database. Those not declared as columns are considered as transient parts of your application logic.

The ColumnAttribute attribute has a variety of properties that you can use to customize these members that represent columns (for example, designating a member as representing a primary key column). For more information, see the Column Attribute section of Attribute-Based Mapping.

LINQ to SQL Associations and Database Foreign-key Relationships

In LINQ to SQL, you represent database associations (such as foreign-key to primary-key relationships) by applying the AssociationAttribute attribute. In the following segment of code, the `Order` class contains a `Customer` property that has an AssociationAttribute attribute. This property and its attribute provide the `Order` class with a relationship to the `Customer` class.

The following code example shows the `Customer` property from the `Order` class.

Example

```
[Association(Name="FK_Orders_Customers", Storage="_Customer", ThisKey="CustomerID", IsForeignKey=true)]
public Customer Customer
{
 get
 {
  return this._Customer.Entity;
 }
 set
 {
  Customer previousValue = this._Customer.Entity;
  if (((previousValue != value)
     || (this._Customer.HasLoadedOrAssignedValue == false)))
  {
   this.SendPropertyChanging();
   if ((previousValue != null))
   {
    this._Customer.Entity = null;
    previousValue.Orders.Remove(this);
   }
   this._Customer.Entity = value;
   if ((value != null))
   {
    value.Orders.Add(this);
    this._CustomerID = value.CustomerID;
   }
   else
   {
    this._CustomerID = default(string);
   }
   this.SendPropertyChanged("Customer");
  }
 }
}
```

```
<Association(Name:="FK_Orders_Customers", Storage:="_Customer", ThisKey:="CustomerID",
IsForeignKey:=true)>  _
Public Property Customer() As Customer
 Get
  Return Me._Customer.Entity
 End Get
 Set
  Dim previousValue As Customer = Me._Customer.Entity
  If (((previousValue Is value)  _
     = false)  _
     OrElse (Me._Customer.HasLoadedOrAssignedValue = false)) Then
   Me.SendPropertyChanging
   If ((previousValue Is Nothing)  _
      = false) Then
    Me._Customer.Entity = Nothing
    previousValue.Orders.Remove(Me)
   End If
   Me._Customer.Entity = value
   If ((value Is Nothing)  _
      = false) Then
    value.Orders.Add(Me)
    Me._CustomerID = value.CustomerID
   Else
    Me._CustomerID = CType(Nothing, String)
   End If
   Me.SendPropertyChanged("Customer")
  End If
 End Set
   End Property
```

For more information, see the Association Attribute section of Attribute-Based Mapping.

LINQ to SQL Methods and Database Stored Procedures

LINQ to SQL supports stored procedures and user-defined functions. In LINQ to SQL, you map these database-defined abstractions to client objects so that you can access them in a strongly typed manner from client code. The method signatures resemble as closely as possible the signatures of the procedures and functions defined in the database. You can use IntelliSense to discover these methods.

A result set that is returned by a call to a mapped procedure is a strongly typed collection.

LINQ to SQL maps stored procedures and functions to methods by using the FunctionAttribute and ParameterAttribute attributes. Methods representing stored procedures are distinguished from those representing user-defined functions by the IsComposable property. If this property is set to `false` (the default), the method represents a stored procedure. If it is set to `true`, the method represents a database function.

NOTE

If you are using Visual Studio, you can use the Object Relational Designer to create methods mapped to stored procedures and user-defined functions.

Example

```
   // This is an example of a stored procedure in the Northwind
   // sample database. The IsComposable property defaults to false.
[Function(Name="dbo.CustOrderHist")]
public ISingleResult<CustOrderHistResult> CustOrderHist([Parameter(Name="CustomerID", DbType="NChar(5)")]
string customerID)
{
 IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())),
customerID);
 return ((ISingleResult<CustOrderHistResult>)(result.ReturnValue));
}
```

```
   ' This is an example of a stored procedure in the Northwind
   ' sample database. The IsComposable property defaults to false.
   <FunctionAttribute(Name:="dbo.CustOrderHist")> _
Public Function CustOrderHist(<Parameter(Name:="CustomerID", DbType:="NChar(5)")> ByVal customerID As
String) As ISingleResult(Of CustOrderHistResult)
       Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.GetCurrentMethod,
MethodInfo), customerID)
       Return CType(result.ReturnValue, ISingleResult(Of CustOrderHistResult))
   End Function
```

For more information, see the Function Attribute, Stored Procedure Attribute, and Parameter Attribute sections of Attribute-Based Mapping and Stored Procedures.

See Also

Attribute-Based Mapping
Background Information

Object States and Change-Tracking

5/1/2017 • 4 min to read • Edit Online

LINQ to SQL objects always participate in some *state*. For example, when LINQ to SQL creates a new object, the object is in `Unchanged` state. A new object that you yourself create is unknown to the DataContext and is in `Untracked` state. Following successful execution of SubmitChanges, all objects known to LINQ to SQL are in `Unchanged` state. (The single exception is represented by those that have been successfully deleted from the database, which are in `Deleted` state and unusable in that DataContext instance.)

Object States

The following table lists the possible states for LINQ to SQL objects.

STATE	DESCRIPTION
`Untracked`	An object not tracked by LINQ to SQL. Examples include the following: - An object not queried through the current DataContext (such as a newly created object). - An object created through deserialization - An object queried through a different DataContext.
`Unchanged`	An object retrieved by using the current DataContext and not known to have been modified since it was created.
`PossiblyModified`	An object which is *attached* to a DataContext. For more information, see Data Retrieval and CUD Operations in N-Tier Applications (LINQ to SQL).
`ToBeInserted`	An object not retrieved by using the current DataContext. This causes a database `INSERT` during SubmitChanges.
`ToBeUpdated`	An object known to have been modified since it was retrieved. This causes a database `UPDATE` during SubmitChanges.
`ToBeDeleted`	An object marked for deletion, causing a database `DELETE` during SubmitChanges.
`Deleted`	An object that has been deleted in the database. This state is final and does not allow for additional transitions.

Inserting Objects

You can explicitly request `Inserts` by using InsertOnSubmit. Alternatively, LINQ to SQL can infer `Inserts` by finding objects connected to one of the known objects that must be updated. For example, if you add an `Untracked` object to an EntitySet<TEntity> or set an EntityRef<TEntity> to an `Untracked` object, you make the `Untracked` object reachable by way of tracked objects in the graph. While processing SubmitChanges, LINQ to SQL traverses the tracked objects and discovers any reachable persistent objects that are not tracked. Such objects are candidates for insertion into the database.

For classes in an inheritance hierarchy, InsertOnSubmit(`o`) also sets the value of the member designated as the *discriminator* to match the type of the object `o`. In the case of a type matching the default discriminator value, this action causes the discriminator value to be overwritten with the default value. For more information, see Inheritance Support.

> **IMPORTANT**
>
> An object added to a `Table` is not in the identity cache. The identity cache reflects only what is retrieved from the database. After a call to InsertOnSubmit, the added entity does not appear in queries against the database until SubmitChanges is successfully completed.

Deleting Objects

You mark a tracked object `o` for deletion by calling DeleteOnSubmit(o) on the appropriate Table<TEntity>. LINQ to SQL considers the removal of an object from an EntitySet<TEntity> as an update operation, and the corresponding foreign key value is set to null. The target of the operation (`o`) is not deleted from its table. For example, `cust.Orders.DeleteOnSubmit(ord)` indicates an update where the relationship between `cust` and `ord` is severed by setting the foreign key `ord.CustomerID` to null. It does not cause the deletion of the row corresponding to `ord`.

LINQ to SQL performs the following processing when an object is deleted (DeleteOnSubmit) from its table:

- When SubmitChanges is called, a `DELETE` operation is performed for that object.
- The removal is not propagated to related objects regardless of whether they are loaded. Specifically, related objects are not loaded for updating the relationship property.
- After successful execution of SubmitChanges, the objects are set to the `Deleted` state. As a result, you cannot use the object or its `id` in that DataContext. The internal cache maintained by a DataContext instance does not eliminate objects that are retrieved or added as new, even after the objects have been deleted in the database.

You can call DeleteOnSubmit only on an object tracked by the DataContext. For an `Untracked` object, you must call Attach before you call DeleteOnSubmit. Calling DeleteOnSubmit on an `Untracked` object throws an exception.

> **NOTE**
>
> Removing an object from a table tells LINQ to SQL to generate a corresponding SQL `DELETE` command at the time of SubmitChanges. This action does not remove the object from the cache or propagate the deletion to related objects.
>
> To reclaim the `id` of a deleted object, use a new DataContext instance. For cleanup of related objects, you can use the *cascade delete* feature of the database, or else manually delete the related objects.
>
> The related objects do not have to be deleted in any special order (unlike in the database).

Updating Objects

You can detect `Updates` by observing notifications of changes. Notifications are provided through the PropertyChanging event in property setters. When LINQ to SQL is notified of the first change to an object, it creates a copy of the object and considers the object a candidate for generating an `Update` statement.

For objects that do not implement INotifyPropertyChanging, LINQ to SQL maintains a copy of the values that objects had when they were first materialized. When you call SubmitChanges, LINQ to SQL compares the current and original values to decide whether the object has been changed.

For updates to relationships, the reference from the child to the parent (that is, the reference corresponding to the foreign key) is considered the authority. The reference in the reverse direction (that is, from parent to child) is optional. Relationship classes (EntitySet<TEntity> and EntityRef<TEntity>) guarantee that the bidirectional references are consistent for one-to-many and one-to-one relationships. If the object model does not use EntitySet<TEntity> or EntityRef<TEntity>, and if the reverse reference is present, it is your responsibility to keep it consistent with the forward reference when the relationship is updated.

If you update both the required reference and the corresponding foreign key, you must make sure that they agree. An InvalidOperationException exception is thrown if the two are not synchronized at the time that you call SubmitChanges. Although foreign key value changes are sufficient for affecting an update of the underlying row, you should change the reference to maintain connectivity of the object graph and bidirectional consistency of relationships.

See Also

Background Information
Insert, Update, and Delete Operations

Optimistic Concurrency: Overview

5/31/2017 • 3 min to read • Edit Online

LINQ to SQL supports optimistic concurrency control. The following table describes terms that apply to optimistic concurrency in LINQ to SQL documentation:

TERMS	DESCRIPTION
concurrency	The situation in which two or more users at the same time try to update the same database row.
concurrency conflict	The situation in which two or more users at the same time try to submit conflicting values to one or more columns of a row.
concurrency control	The technique used to resolve concurrency conflicts.
optimistic concurrency control	The technique that first investigates whether other transactions have changed values in a row before permitting changes to be submitted. Contrast with *pessimistic concurrency control*, which locks the record to avoid concurrency conflicts. *Optimistic* control is so termed because it considers the chances of one transaction interfering with another to be unlikely.
conflict resolution	The process of refreshing a conflicting item by querying the database again and then reconciling differences. When an object is refreshed, the LINQ to SQL change tracker holds the following data: - The values originally taken from the database and used for the update check. - The new database values from the subsequent query. LINQ to SQL then determines whether the object is in conflict (that is, whether one or more of its member values has changed). If the object is in conflict, LINQ to SQL next determines which of its members are in conflict. Any member conflict that LINQ to SQL discovers is added to a conflict list.

In the LINQ to SQL object model, an *optimistic concurrency conflict* occurs when both of the following conditions are true:

- The client tries to submit changes to the database.
- One or more update-check values have been updated in the database since the client last read them.

Resolution of this conflict includes discovering which members of the object are in conflict, and then deciding what you want to do about it.

NOTE

Only members mapped as Always or WhenChanged participate in optimistic concurrency checks. No check is performed for members marked Never. For more information, see UpdateCheck.

Example

For example, in the following scenario, User1 starts to prepare an update by querying the database for a row. User1 receives a row with values of Alfreds, Maria, and Sales.

User1 wants to change the value of the Manager column to Alfred and the value of the Department column to Marketing. Before User1 can submit those changes, User2 has submitted changes to the database. So now the value of the Assistant column has been changed to Mary and the value of the Department column to Service.

When User1 now tries to submit changes, the submission fails and a ChangeConflictException exception is thrown. This result occurs because the database values for the Assistant column and the Department column are not those that were expected. Members representing the Assistant and Department columns are in conflict. The following table summarizes the situation.

	MANAGER	ASSISTANT	DEPARTMENT
Original state	Alfreds	Maria	Sales
User1	Alfred		Marketing
User2		Mary	Service

You can resolve conflicts such as this in different ways. For more information, see How to: Manage Change Conflicts.

Conflict Detection and Resolution Checklist

You can detect and resolve conflicts at any level of detail. At one extreme, you can resolve all conflicts in one of three ways (see RefreshMode) without additional consideration. At the other extreme, you can designate a specific action for each type of conflict on every member in conflict.

- Specify or revise UpdateCheck options in your object model.

  For more information, see How to: Specify Which Members are Tested for Concurrency Conflicts.

- In the try/catch block of your call to SubmitChanges, specify at what point you want exceptions to be thrown.

  For more information, see How to: Specify When Concurrency Exceptions are Thrown.

- Determine how much conflict detail you want to retrieve, and include code in your try/catch block accordingly.

  For more information, see How to: Retrieve Entity Conflict Information and How to: Retrieve Member Conflict Information.

- Include in your `try` / `catch` code how you want to resolve the various conflicts you discover.

  For more information, see How to: Resolve Conflicts by Retaining Database Values, How to: Resolve Conflicts by Overwriting Database Values, and How to: Resolve Conflicts by Merging with Database Values.

LINQ to SQL Types That Support Conflict Discovery and Resolution

Classes and features to support the resolution of conflicts in optimistic concurrency in LINQ to SQL include the following:

- System.Data.Linq.ObjectChangeConflict
- System.Data.Linq.MemberChangeConflict
- System.Data.Linq.ChangeConflictCollection
- System.Data.Linq.ChangeConflictException
- System.Data.Linq.DataContext.ChangeConflicts
- System.Data.Linq.DataContext.SubmitChanges
- System.Data.Linq.DataContext.Refresh
- System.Data.Linq.Mapping.ColumnAttribute.UpdateCheck
- System.Data.Linq.Mapping.UpdateCheck
- System.Data.Linq.RefreshMode

See Also

How to: Manage Change Conflicts

Query Concepts

5/1/2017 • 1 min to read • Edit Online

This section describes key concepts for designing LINQ queries in LINQ to SQL.

In This Section

LINQ to SQL Queries
Refers to general LINQ topics, and explains items specific to LINQ to SQL.

Querying Across Relationships
Explains how to use associations in the LINQ to SQL object model.

Remote vs. Local Execution
Explains how to specify where you want your query executed.

Deferred versus Immediate Loading
Describes how to specify when related objects are loaded.

Related Sections

Programming Guide
Contains links to topics that explain the LINQ to SQL technology.

Object Identity
Explains the concept of object identity in LINQ to SQL.

Introduction to LINQ Queries (C#)
Provides an introduction to query operations in LINQ.

LINQ to SQL Queries

5/1/2017 • 1 min to read • Edit Online

You define LINQ to SQL queries by using the same syntax as you would in LINQ. The only difference is that the objects referenced in your queries are mapped to elements in a database. For more information, see Introduction to LINQ Queries (C#).

LINQ to SQL translates the queries you write into equivalent SQL queries and sends them to the server for processing. More specifically, your application uses the LINQ to SQL API to request query execution. The LINQ to SQL provider then transforms the query into SQL text and delegates execution to the ADO provider. The ADO provider returns query results as a `DataReader`. The LINQ to SQL provider translates the ADO results to an IQueryable collection of user objects.

NOTE

Most methods and operators on .NET Framework built-in types have direct translations to SQL. Those that LINQ cannot translate generate run-time exceptions. For more information, see SQL-CLR Type Mapping.

The following table shows the similarities and differences between LINQ and LINQ to SQL query items.

ITEM	LINQ QUERY	LINQ TO SQL QUERY
Return type of the local variable that holds the query (for queries that return sequences)	Generic `IEnumerable`	Generic `IQueryable`
Specifying the data source	Uses the `From` (Visual Basic) or `from` (C#) clause	Same
Filtering	Uses the `Where` / `where` clause	Same
Grouping	Uses the `Group…By` / `groupby` clause	Same
Selecting (Projecting)	Uses the `Select` / `select` clause	Same
Deferred versus immediate execution	See Introduction to LINQ Queries (C#)	Same
Implementing joins	Uses the `Join` / `join` clause	Can use the `Join` / `join` clause, but more effectively uses the AssociationAttribute attribute. For more information, see Querying Across Relationships.
Remote versus local execution		For more information, see Remote vs. Local Execution.
Streaming versus cached querying	Not applicable in a local memory scenario	

See Also

Introduction to LINQ Queries (C#)

Basic LINQ Query Operations

Type Relationships in LINQ Query Operations

Query Concepts

Querying Across Relationships

5/1/2017 • 3 min to read • Edit Online

References to other objects or collections of other objects in your class definitions directly correspond to foreign-key relationships in the database. You can use these relationships when you query by using dot notation to access the relationship properties and navigate from one object to another. These access operations translate to more complex joins or correlated subqueries in the equivalent SQL.

For example, the following query navigates from orders to customers as a way to restrict the results to only those orders for customers located in London.

```
        Northwnd db = new Northwnd(@"northwnd.mdf");

        IQueryable<Order> londonOrderQuery =
from ord in db.Orders
where ord.Customer.City == "London"
select ord;
```

```
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim londonOrderQuery = _
From ord In db.Orders _
Where ord.Customer.City = "London" _
Select ord
```

If relationship properties did not exist you would have to write them manually as *joins*, just as you would do in a SQL query, as in the following code:

```
        Northwnd db = new Northwnd(@"northwnd.mdf");
        IQueryable<Order> londonOrderQuery =
from cust in db.Customers
join ord in db.Orders on cust.CustomerID equals ord.CustomerID
where cust.City == "London"
select ord;
```

```
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim londOrderQuery = _
From cust In db.Customers _
Join ord In db.Orders On cust.CustomerID Equals ord.CustomerID _
Select ord
```

You can use the *relationship* property to define this particular relationship one time. You can then use the more convenient dot syntax. But relationship properties exist more importantly because domain-specific object models are typically defined as hierarchies or graphs. The objects that you program against have references to other objects. It is only a happy coincidence that object-to-object relationships correspond to foreign-key-styled relationships in databases. Property access then provides a convenient way to write joins.

With regard to this, relationship properties are more important on the results side of a query than as part of the query itself. After the query has retrieved data about a particular customer, the class definition indicates that customers have orders. In other words, you expect the `Orders` property of a particular customer to be a collection that is populated with all the orders from that customer. That is in fact the contract you declared by defining the classes in this manner. You expect to see the orders there even if the query did not request orders. You expect your object model to maintain an illusion that it is an in-memory extension of the database with related objects immediately available.

Now that you have relationships, you can write queries by referring to the relationship properties defined in your classes. These relationship references correspond to foreign-key relationships in the database. Operations that use these relationships translate to more complex joins in the equivalent SQL. As long as you have defined a relationship (using the AssociationAttribute attribute), you do not have to code an explicit join in LINQ to SQL.

To help maintain this illusion, LINQ to SQL implements a technique called *deferred loading*. For more information, see Deferred versus Immediate Loading.

Consider the following SQL query to project a list of `CustomerID`-`OrderID` pairs:

```
SELECT t0.CustomerID, t1.OrderID
FROM   Customers AS t0 INNER JOIN
          Orders AS t1 ON t0.CustomerID = t1.CustomerID
WHERE  (t0.City = @p0)
```

To obtain the same results by using LINQ to SQL, you use the `Orders` property reference already existing in the `Customer` class. The `Orders` reference provides the necessary information to execute the query and project the `CustomerID`-`OrderID` pairs, as in the following code:

```
        Northwnd db = new Northwnd(@"northwnd.mdf");
        var idQuery =
from cust in db.Customers
from ord in cust.Orders
where cust.City == "London"
select new { cust.CustomerID, ord.OrderID };
```

```
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim idQuery = _
From cust In db.Customers, ord In cust.Orders _
Where cust.City = "London" _
Select cust.CustomerID, ord.OrderID
```

You can also do the reverse. That is, you can query `Orders` and use its `Customer` relationship reference to access information about the associated `Customer` object. The following code projects the same `CustomerID`-`OrderID` pairs as before, but this time by querying `Orders` instead of `Customers`.

```
        Northwnd db = new Northwnd(@"northwnd.mdf");
        var idQuery =
from ord in db.Orders
where ord.Customer.City == "London"
select new { ord.Customer.CustomerID, ord.OrderID };
```

```
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim idQuery = _
From ord In db.Orders _
Where ord.Customer.City = "London" _
Select ord.CustomerID, ord.OrderID
```

See Also

Query Concepts

Remote vs. Local Execution

5/1/2017 • 2 min to read • Edit Online

You can decide to execute your queries either remotely (that is, the database engine executes the query against the database) or locally (LINQ to SQL executes the query against a local cache).

Remote Execution

Consider the following query:

```
            Northwnd db = new Northwnd(@"northwnd.mdf");
            Customer c = db.Customers.Single(x => x.CustomerID == "19283");
foreach (Order ord in
    c.Orders.Where(o => o.ShippedDate.Value.Year == 1998))
{
    // Do something.
}
```

```
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim c As Customer = _
(From cust In db.Customers _
Where cust.CustomerID = 19283).First()
    Dim orders = From ord In c.Orders _
        Where ord.ShippedDate.Value.Year = 1998
    For Each nextOrder In orders
        ' Do something.
    Next
```

If your database has thousands of rows of orders, you do not want to retrieve them all to process a small subset. In LINQ to SQL, the EntitySet<TEntity> class implements the IQueryable interface. This approach makes sure that such queries can be executed remotely. Two major benefits flow from this technique:

- Unnecessary data is not retrieved.
- A query executed by the database engine is often more efficient because of the database indexes.

Local Execution

In other situations, you might want to have the complete set of related entities in the local cache. For this purpose, EntitySet<TEntity> provides the Load method to explicitly load all the members of the EntitySet<TEntity>.

If an EntitySet<TEntity> is already loaded, subsequent queries are executed locally. This approach helps in two ways:

- If the complete set must be used locally or multiple times, you can avoid remote queries and associated latencies.
- The entity can be serialized as a complete entity.

The following code fragment illustrates how local execution can be obtained:

```
            Northwnd db = new Northwnd(@"northwnd.mdf");
            Customer c = db.Customers.Single(x => x.CustomerID == "19283");
c.Orders.Load();

foreach (Order ord in
    c.Orders.Where(o => o.ShippedDate.Value.Year == 1998))
{
    // Do something.
}

        }
```

```
    Dim db As New Northwnd("c:\northwnd.mdf")
    Dim c As Customer = _
(From cust In db.Customers _
 Where cust.CustomerID = 19283).First
    c.Orders.Load()

    Dim orders = From ord In c.Orders _
        Where ord.ShippedDate.Value.Year = 1998

    For Each nextOrder In orders
        ' Do something.
    Next
```

Comparison

These two capabilities provide a powerful combination of options: remote execution for large collections and local execution for small collections or where the complete collection is needed. You implement remote execution through IQueryable, and local execution against an in-memory IEnumerable<T> collection. To force local execution (that is, IEnumerable<T>), see Convert a Type to a Generic IEnumerable.

Queries Against Unordered Sets

Note the important difference between a local collection that implements List<T> and a collection that provides remote queries executed against *unordered sets* in a relational database. List<T> methods such as those that use index values require list semantics, which typically cannot be obtained through a remote query against an unordered set. For this reason, such methods implicitly load the EntitySet<TEntity> to allow local execution.

See Also

Query Concepts

Deferred versus Immediate Loading

5/1/2017 • 2 min to read • Edit Online

When you query for an object, you actually retrieve only the object you requested. The *related* objects are not automatically fetched at the same time. (For more information, see Querying Across Relationships.) You cannot see the fact that the related objects are not already loaded, because an attempt to access them produces a request that retrieves them.

For example, you might want to query for a particular set of orders and then only occasionally send an e-mail notification to particular customers. You would not necessarily need initially to retrieve all customer data with every order. You can use deferred loading to defer retrieval of extra information until you absolutely have to. Consider the following example:

```
    Northwnd db = new Northwnd(@"northwnd.mdf");

    IQueryable<Order> notificationQuery =
    from ord in db.Orders
 where ord.ShipVia == 3
  select ord;

    foreach (Order ordObj in notificationQuery)
    {
        if (ordObj.Freight > 200)
            SendCustomerNotification(ordObj.Customer);
        ProcessOrder(ordObj);
    }

}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")
Dim notificationQuery = _
    From ord In db.Orders _
    Where ord.ShipVia = 3 _
    Select ord

For Each ordObj As Order In notificationQuery
    If ordObj.Freight > 200 Then
        SendCustomerNotification(ordObj.Customer)
        ProcessOrder(ordObj)
    End If

Next
```

The opposite might also be true. You might have an application that has to view customer and order data at the same time. You know you need both sets of data. You know your application needs order information for each customer as soon as you get the results. You would not want to submit individual queries for orders for every customer. What you really want is to retrieve the order data together with the customers.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

db.DeferredLoadingEnabled = false;

IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custQuery)
{
    foreach (Order ordObj in custObj.Orders)
    {
        ProcessCustomerOrder(ordObj);
    }
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

db.DeferredLoadingEnabled = False

Dim custQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Select cust

For Each custObj As Customer In custQuery
    For Each ordObj As Order In custObj.Orders
        ProcessCustomerOrder(ordObj)
    Next
Next
```

You can also join customers and orders in a query by forming the cross-product and retrieving all the relative bits of data as one large projection. But these results are not entities. (For more information, see The LINQ to SQL Object Model). Entities are objects that have identity and that you can modify, whereas these results would be projections that cannot be changed and persisted. Even worse, you would be retrieving lots of redundant data as each customer repeats for each order in the flattened join output.

What you really need is a way to retrieve a set of related objects at the same time. The set is a delineated section of a graph so that you would never be retrieving more or less than was necessary for your intended use. For this purpose, LINQ to SQL provides DataLoadOptions for immediate loading of a region of your object model. Methods include:

- The LoadWith method, to immediately load data related to the main target.
- The AssociateWith method, to filter objects retrieved for a particular relationship.

See Also

Query Concepts

Retrieving Objects from the Identity Cache

5/1/2017 • 3 min to read • Edit Online

This topic describes the types of LINQ to SQL queries that return an object from the identity cache that is managed by the DataContext.

In LINQ to SQL, one of the ways in which the DataContext manages objects is by logging object identities in an identity cache as queries are executed. In some cases, LINQ to SQL will attempt to retrieve an object from the identity cache before executing a query in the database.

In general, for a LINQ to SQL query to return an object from the identity cache, the query must be based on the primary key of an object and must return a single object. In particular, the query must be in one of the general forms shown below.

> **NOTE**
> Pre-compiled queries will not return objects from the identity cache. For more information about pre-compiled queries, see CompiledQuery and How to: Store and Reuse Queries.

A query must be in one of the following general forms to retrieve an object from the identity cache:

- Table<TEntity> `.Function1(` `predicate` `)`
- Table<TEntity> `.Function1(` `predicate` `).Function2()`

In these general forms, `Function1`, `Function2`, and `predicate` are defined as follows.

`Function1` can be any of the following:

- Where
- First
- FirstOrDefault
- Single
- SingleOrDefault

`Function2` can be any of the following:

- First
- FirstOrDefault
- Single
- SingleOrDefault

`predicate` must be an expression in which the object's primary key property is set to a constant value. If an object has a primary key defined by more than one property, each primary key property must be set to a constant value. The following are examples of the form `predicate` must take:

- `c => c.PK == constant_value`
- `c => c.PK1 == constant_value1 && c=> c.PK2 == constant_value2`

Example

The following code provides examples of the types of LINQ to SQL queries that retrieve an object from the identity cache.

```
NorthwindDataContext context = new NorthwindDataContext();

// This query does not retrieve an object from
// the query cache because it is the first query.
// There are no objects in the cache.
var a = context.Customers.First();
Console.WriteLine("First query gets customer {0}. ", a.CustomerID);

// This query returns an object from the query cache.
var b = context.Customers.Where(c => c.CustomerID == a.CustomerID);
foreach (var customer in b )
{
    Console.WriteLine(customer.CustomerID);
}

// This query returns an object from the identity cache.
// Note that calling FirstOrDefault(), Single(), or SingleOrDefault()
// instead of First() will also return an object from the cache.
var x = context.Customers.
    Where(c => c.CustomerID == a.CustomerID).
    First();
Console.WriteLine(x.CustomerID);

// This query returns an object from the identity cache.
// Note that calling FirstOrDefault(), Single(), or SingleOrDefault()
// instead of First() (each with the same predicate) will also
// return an object from the cache.
var y = context.Customers.First(c => c.CustomerID == a.CustomerID);
Console.WriteLine(y.CustomerID);
```

```
Dim context As New NorthwindDataContext()

' This query does not retrieve an object from
' the query cache because it is the first query.
' There are no objects in the cache.
Dim a = context.Customers.First()
Console.WriteLine("First query gets customer {0}. ", a.CustomerID)

' This query returns an object from the query cache.
Dim b = context.Customers.Where(Function(c) c.CustomerID = a.CustomerID)
For Each customer In b
    Console.WriteLine(customer.CustomerID)
Next

' This query returns an object from the identity cache.
' Note that calling FirstOrDefault(), Single(), or SingleOrDefault()
' instead of First() will also return an object from the cache.
Dim x = context.Customers. _
    Where(Function(c) c.CustomerID = a.CustomerID). _
    First()
Console.WriteLine(x.CustomerID)

' This query returns an object from the identity cache.
' Note that calling FirstOrDefault(), Single(), or SingleOrDefault()
' instead of First() (each with the same predicate) will also
' return an object from the cache.
Dim y = context.Customers.First(Function(c) c.CustomerID = a.CustomerID)
Console.WriteLine(y.CustomerID)
```

See Also

Query Concepts
Object Identity
Background Information
Object Identity

Security in LINQ to SQL

5/1/2017 • 1 min to read • Edit Online

Security risks are always present when you connect to a database. Although LINQ to SQL may include some new ways to work with data in SQL Server, it does not provide any additional security mechanisms.

Access Control and Authentication

LINQ to SQL does not have its own user model or authentication mechanisms. Use SQL Server Security to control access to the database, database tables, views, and stored procedures that are mapped to your object model. Grant the minimally required access to users and require strong passwords for user authentication.

Mapping and Schema Information

SQL-CLR type mapping and database schema information in your object model or external mapping file is available for all with access to those files in the file system. Assume that schema information will be available to all who can access the object model or external mapping file.To prevent more widespread access to schema information, use file security mechanisms to secure source files and mapping files.

Connection Strings

Using passwords in connection strings should be avoided whenever possible. Not only is a connection string a security risk in its own right, but the connection string may also be added in clear text to the object model or external mapping file when using the Object Relational Designer or SQLMetal command-line tool. Anyone with access to the object model or external mapping file via the file system could see the connection password (if it is included in the connection string).

To minimize such risks, use integrated security to make a trusted connection with SQL Server. By using this approach, you do not have to store a password in the connection string. For more information, see SQL Server Security.

In the absence of integrated security, a clear-text password will be needed in the connection string. The best way to help secure your connection string, in increasing order of risk, is as follows:

- Use integrated security.
- Secure connection strings with passwords and minimize passing around connection strings.
- Use a System.Data.SqlClient.SqlConnection class instead of a connection string since it limits the duration of exposure. The LINQ to SQL System.Data.Linq.DataContext class can be instantiated using a SqlConnection.
- Minimize lifetimes and touch points for all connection strings.

See Also

Background Information
Frequently Asked Questions

Serialization

5/1/2017 • 5 min to read • Edit Online

This topic describes LINQ to SQL serialization capabilities. The paragraphs that follow provide information about how to add serialization during code generation at design time and the run-time serialization behavior of LINQ to SQL classes.

You can add serialization code at design time by either of the following methods:

- In the Object Relational Designer, change the **Serialization Mode** property to **Unidirectional**.
- On the SQLMetal command line, add the **/serialization** option. For more information, see SqlMetal.exe (Code Generation Tool).

Overview

The code generated by LINQ to SQL provides deferred loading capabilities by default. Deferred loading is very convenient on the mid-tier for transparent loading of data on demand. However, it is problematic for serialization, because the serializer triggers deferred loading whether deferred loading is intended or not. In effect, when an object is serialized, its transitive closure under all outbound defer-loaded references is serialized.

The LINQ to SQL serialization feature addresses this problem, primarily through two mechanisms:

- A DataContext mode for turning off deferred loading (ObjectTrackingEnabled). For more information, see DataContext.
- A code-generation switch to generate System.Runtime.Serialization.DataContractAttribute and System.Runtime.Serialization.DataMemberAttribute attributes on generated entities. This aspect, including the behavior of defer-loading classes under serialization, is the major subject of this topic.

Definitions

- *DataContract serializer*: Default serializer used by the Windows Communication Framework (WCF) component of the .NET Framework 3.0 or later versions.
- *Unidirectional serialization*: The serialized version of a class that contains only a one-way association property (to avoid a cycle). By convention, the property on the parent side of a primary-foreign key relationship is marked for serialization. The other side in a bidirectional association is not serialized.

  Unidirectional serialization is the only type of serialization supported by LINQ to SQL.

Code Example

The following code uses the traditional `Customer` and `Order` classes from the Northwind sample database, and shows how these classes are decorated with serialization attributes.

```
// The class is decorated with the DataContract attribute.
[Table(Name="dbo.Customers")]
[DataContract()]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
```

```
' The class is decorated with the DataContract attribute.
<Table(Name:="dbo.Customers"),  _
 DataContract()>  _
Partial Public Class Customer
 Implements System.ComponentModel.INotifyPropertyChanging, System.ComponentModel.INotifyPropertyChanged
```

```
   // Private fields are not decorated with any attributes, and are
   // elided.
   private string _CustomerID;

   // Public properties are decorated with the DataMember
   // attribute and the Order property specifying the serial
   // number. See the Order class later in this topic for
   // exceptions.
public Customer()
{
 this.Initialize();
}

[Column(Storage="_CustomerID", DbType="NChar(5) NOT NULL", CanBeNull=false, IsPrimaryKey=true)]
[DataMember(Order=1)]
public string CustomerID
{
 get
 {
  return this._CustomerID;
 }
 set
 {
  if ((this._CustomerID != value))
  {
   this.OnCustomerIDChanging(value);
   this.SendPropertyChanging();
   this._CustomerID = value;
   this.SendPropertyChanged("CustomerID");
   this.OnCustomerIDChanged();
  }
 }
}
```

```
   ' Private fields are not decorated with any attributes,
   ' and are elided.
   Private _CustomerID As String

   ' Public properties are decorated with the DataMember
   ' attribute and the Order property specifying the
   ' serial number. See the Order class later in this topic
   ' for exceptions
<Column(Storage:="_CustomerID", DbType:="NChar(5) NOT NULL", CanBeNull:=false, IsPrimaryKey:=true),  _
 DataMember(Order:=1)>  _
Public Property CustomerID() As String
 Get
  Return Me._CustomerID
 End Get
 Set
  If ((Me._CustomerID = value)  _
     = false) Then
   Me.OnCustomerIDChanging(value)
   Me.SendPropertyChanging
   Me._CustomerID = value
   Me.SendPropertyChanged("CustomerID")
   Me.OnCustomerIDChanged
  End If
 End Set
End Property
```

```
   // The following Association property is decorated with
   // DataMember because it is the parent side of the
   // relationship. The reverse property in the Order class
   // does not have a DataMember attribute. This factor
   // prevents a 'cycle.'
[Association(Name="FK_Orders_Customers", Storage="_Orders", OtherKey="CustomerID", DeleteRule="NO ACTION")]
[DataMember(Order=13)]
public EntitySet<Order> Orders
{
 get
 {
  return this._Orders;
 }
 set
 {
  this._Orders.Assign(value);
 }
}
```

```
   ' The following Association property is decorated with
   ' DataMember because it is the parent side of the
   ' relationship. The reverse property in the Order
   ' class does not have a DataMember attribute. This
   ' factor prevents a 'cycle.'
   <Association(Name:="FK_Orders_Customers", Storage:="_Orders", OtherKey:="CustomerID", DeleteRule:="NO
ACTION"), _
 DataMember(Order:=13)> _
Public Property Orders() As EntitySet(Of [Order])
       Get
           Return Me._Orders
       End Get
       Set(ByVal value As EntitySet(Of [Order]))
           Me._Orders.Assign(Value)
       End Set
   End Property
```

For the `Order` class in the following example, only the reverse association property corresponding to the

`Customer` class is shown for brevity. It does not have a `DataMember` attribute to avoid a cycle.

```
// The class for the Orders table is also decorated with the
// DataContract attribute.
[Table(Name="dbo.Orders")]
[DataContract()]
public partial class Order : INotifyPropertyChanging, INotifyPropertyChanged
```

```
' The class for the Orders table is also decorated with the
' DataContract attribute.
<Table(Name:="dbo.Orders"),  _
 DataContract()>  _
Partial Public Class [Order]
 Implements System.ComponentModel.INotifyPropertyChanging, System.ComponentModel.INotifyPropertyChanged
```

```
   // Private fields for the Orders table are not decorated with
   // any attributes, and are elided.
private int _OrderID;

   // Public properties are decorated with the DataMember
   // attribute.
   // The reverse Association property on the side of the
   // foreign key does not have the DataMember attribute.
   [Association(Name = "FK_Orders_Customers", Storage = "_Customer", ThisKey = "CustomerID", IsForeignKey =
true)]
   public Customer Customer
```

```
   ' Private fields for the Orders table are not decorated with
   ' any attributes, and are elided.
   Private _CustomerID As String

   ' Public properties are decorated with the DataMember
   ' attribute.
   ' The reverse Association property on the side of the
   ' foreign key does not have the DataMember attribute.
<Association(Name:="FK_Orders_Customers", Storage:="_Customer", ThisKey:="CustomerID", IsForeignKey:=true)>  _
Public Property Customer() As Customer
```

How to Serialize the Entities

You can serialize the entities in the codes shown in the previous section as follows;

```
Northwnd db = new Northwnd(@"c\northwnd.mdf");

Customer cust = db.Customers.Where(c => c.CustomerID ==
    "ALFKI").Single();

DataContractSerializer dcs =
    new DataContractSerializer(typeof(Customer));
StringBuilder sb = new StringBuilder();
XmlWriter writer = XmlWriter.Create(sb);
dcs.WriteObject(writer, cust);
writer.Close();
string xml = sb.ToString();
```

```
Dim db As New Northwnd("...")

Dim cust = (From c In db.Customers _
           Where c.CustomerID = "ALFKI").Single

Dim dcs As New DataContractSerializer(GetType(Customer))

Dim sb As StringBuilder = New StringBuilder()
Dim writer As XmlWriter = XmlWriter.Create(sb)
dcs.WriteObject(writer, cust)
writer.Close()
Dim xml As String = sb.ToString()
```

Self-Recursive Relationships

Self-recursive relationships follow the same pattern. The association property corresponding to the foreign key does not have a `DataMember` attribute, whereas the parent property does.

Consider the following class that has two self-recursive relationships: Employee.Manager/Reports and Employee.Mentor/Mentees.

```
// No DataMember attribute.
public Employee Manager;
[DataMember(Order = 3)]
public EntitySet<Employee> Reports;

// No DataMember
public Employee Mentor;
[DataMember(Order = 5)]
public EntitySet<Employee> Mentees;
```

```
' No DataMember attribute
Public Manager As Employee
<DataMember(Order:=3)> _
Public Reports As EntitySet(Of Employee)

' No DataMember attribute
Public Mentor As Employee
<DataMember(Order:=5)> _
Public Mentees As EntitySet(Of Employee)
```

See Also

Background Information
SqlMetal.exe (Code Generation Tool)
How to: Make Entities Serializable

Stored Procedures

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL uses methods in your object model to represent stored procedures in the database. You designate methods as stored procedures by applying the FunctionAttribute attribute and, where required, the ParameterAttribute attribute. For more information, see The LINQ to SQL Object Model.

Developers using Visual Studio would typically use the Object Relational Designer to map stored procedures. The topics in this section show how to form and call these methods in your application if you write the code yourself.

In This Section

How to: Return Rowsets
Describes how to return rows of data and shows how to use an input parameter.

How to: Use Stored Procedures that Take Parameters
Describes how to use input and output parameters.

How to: Use Stored Procedures Mapped for Multiple Result Shapes
Describes how to provide for returns of multiple shapes in the same stored procedure.

How to: Use Stored Procedures Mapped for Sequential Result Shapes
Describes how to provide for multiple shapes where the return sequence is known.

Customizing Operations By Using Stored Procedures
Describes how to use stored procedures to implement insert, update, and delete operations.

Customizing Operations by Using Stored Procedures Exclusively
Describes how to use nothing but stored procedures to implement insert, update, and delete operations.

Related Sections

Programming Guide
Provides information about how to create and use your LINQ to SQL object model.

Walkthrough: Using Only Stored Procedures (Visual Basic)
Includes procedures that illustrate how to use stored procedures in Visual Basic.

Walkthrough: Using Only Stored Procedures (C#)
Includes procedures that illustrate how to use stored procedures in C#.

How to: Return Rowsets

5/1/2017 • 1 min to read • Edit Online

This example returns a rowset from the database, and includes an input parameter to filter the result.

When you execute a stored procedure that returns a rowset, you use a *result* class that stores the returns from the stored procedure. For more information, see Analyzing LINQ to SQL Source Code.

Example

The following example represents a stored procedure that returns rows of customers and uses an input parameter to return only those rows that list "London" as the customer city. The example assumes an enumerable `CustomersByCityResult` class.

```
CREATE PROCEDURE [dbo].[Customers By City]
    (@param1 NVARCHAR(20))
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    SELECT CustomerID, ContactName, CompanyName, City from Customers
        as c where c.City=@param1
END
```

```
[Function(Name="dbo.Customers By City")]
public ISingleResult<CustomersByCityResult> CustomersByCity([Parameter(DbType="NVarChar(20)")] string param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this,         ((MethodInfo)(MethodInfo.GetCurrentMethod())),
param1);
    return ((ISingleResult<CustomersByCityResult>)(result.ReturnValue));
}

// Call the stored procedure.
void ReturnRowset()
{
    Northwnd db = new Northwnd(@"c:\northwnd.mdf");

    ISingleResult<CustomersByCityResult> result =
        db.CustomersByCity("London");

    foreach (CustomersByCityResult cust in result)
    {
        Console.WriteLine("CustID={0}; City={1}", cust.CustomerID,
            cust.City);
    }
}
```

```
    <FunctionAttribute(Name:="dbo.Customers By City")> _
        Public Function CustomersByCity(<Parameter(DbType:="NVarChar(20)")> ByVal param1 As String) As
ISingleResult(Of CustomersByCityResult)

        Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.GetCurrentMethod, MethodInfo),
param1)
        Return CType(result.ReturnValue, ISingleResult(Of CustomersByCityResult))
    End Function

Sub ReturnRowset()
    ' Call the stored procedure.
    Dim db As New Northwnd("c:\northwnd.mdf")

    Dim result As IEnumerable(Of CustomersByCityResult) = _
        db.CustomersByCity("London")

    For Each cust As CustomersByCityResult In result
        Console.WriteLine("CustID={0}; City={1}", _
            cust.CustomerID, cust.City)
    Next

End Sub
```

See Also

Stored Procedures
Downloading Sample Databases

How to: Use Stored Procedures that Take Parameters

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL maps output parameters to reference parameters, and for value types declares the parameter as nullable.

For an example of how to use an input parameter in a query that returns a rowset, see How to: Return Rowsets.

Example

The following example takes a single input parameter (the customer ID) and returns an out parameter (the total sales for that customer).

```
CREATE PROCEDURE [dbo].[CustOrderTotal]
@CustomerID nchar(5),
@TotalSales money OUTPUT
AS
SELECT @TotalSales = SUM(OD.UNITPRICE*(1-OD.DISCOUNT) * OD.QUANTITY)
FROM ORDERS O, "ORDER DETAILS" OD
where O.CUSTOMERID = @CustomerID AND O.ORDERID = OD.ORDERID
```

```
   [Function(Name="dbo.CustOrderTotal")]
[return: Parameter(DbType="Int")]
public int CustOrderTotal([Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID,
[Parameter(Name="TotalSales", DbType="Money")] ref System.Nullable<decimal> totalSales)
{
 IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())),
customerID, totalSales);
 totalSales = ((System.Nullable<decimal>)(result.GetParameterValue(1)));
 return ((int)(result.ReturnValue));
}
```

```
<FunctionAttribute(Name:="dbo.CustOrderTotal")> _
 Public Function CustOrderTotal(<Parameter(Name:="CustomerID", DbType:="NChar(5)")> ByVal customerID As String,
<Parameter(Name:="TotalSales", DbType:="Money")> ByRef totalSales As System.Nullable(Of Decimal)) As
<Parameter(DbType:="Int")> Integer
    Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.GetCurrentMethod, MethodInfo),
customerID, totalSales)
    totalSales = CType(result.GetParameterValue(1), System.Nullable(Of Decimal))
    Return CType(result.ReturnValue, Integer)
End Function
```

Example

You would call this stored procedure as follows:

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");
decimal? totalSales = 0;
db.CustOrderTotal("alfki", ref totalSales);

Console.WriteLine(totalSales);
```

```
Dim db As New Northwnd("C:\...\northwnd.mdf")
Dim totalSales As Decimal? = 0
db.CustOrderTotal("alfki", totalSales)

Console.WriteLine(totalSales)
```

See Also

Stored Procedures
Downloading Sample Databases
Using Nullable Types
Nullable Value Types

How to: Use Stored Procedures Mapped for Multiple Result Shapes

5/1/2017 • 2 min to read • Edit Online

When a stored procedure can return multiple result shapes, the return type cannot be strongly typed to a single projection shape. Although LINQ to SQL can generate all possible projection types, it cannot know the order in which they will be returned.

Contrast this scenario with stored procedures that produce multiple result shapes sequentially. For more information, see How to: Use Stored Procedures Mapped for Sequential Result Shapes.

The ResultTypeAttribute attribute is applied to stored procedures that return multiple result types to specify the set of types the procedure can return.

Example

In the following SQL code example, the result shape depends on the input (`shape =1` or `shape = 2`). You do not know which projection will be returned first.

```
CREATE PROCEDURE VariableResultShapes(@shape int)
AS
if(@shape = 1)
    select CustomerID, ContactTitle, CompanyName from customers
else if(@shape = 2)
    select OrderID, ShipName from orders
```

```
[Function(Name="dbo.VariableResultShapes")]
[ResultType(typeof(VariableResultShapesResult1))]
[ResultType(typeof(VariableResultShapesResult2))]
public IMultipleResults VariableResultShapes([Parameter(DbType="Int")] System.Nullable<int> shape)
{
 IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())), shape);
 return ((IMultipleResults)(result.ReturnValue));
}
```

```
<FunctionAttribute(Name:="dbo.VariableResultShapes"), _
ResultType(GetType(VariableResultShapesResult1)), _
ResultType(GetType(VariableResultShapesResult2))> _
Public Function VariableResultShapes(<Parameter(DbType:="Int")> ByVal shape As System.Nullable(Of Integer)) As
IMultipleResults
    Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.GetCurrentMethod, MethodInfo),
shape)
    Return CType(result.ReturnValue, IMultipleResults)
End Function
```

Example

You would use code similar to the following to execute this stored procedure.

NOTE

You must use the GetResult pattern to obtain an enumerator of the correct type, based on your knowledge of the stored procedure.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Assign the results of the procedure with an argument
// of (1) to local variable 'result'.
IMultipleResults result = db.VariableResultShapes(1);

// Iterate through the list and write results (the company names)
// to the console.
foreach(VariableResultShapesResult1 compName in
    result.GetResult<VariableResultShapesResult1>())
{
    Console.WriteLine(compName.CompanyName);
}

// Pause to view company names; press Enter to continue.
Console.ReadLine();

// Assign the results of the procedure with an argument
// of (2) to local variable 'result'.
IMultipleResults result2 = db.VariableResultShapes(2);

// Iterate through the list and write results (the order IDs)
// to the console.
foreach (VariableResultShapesResult2 ord in
    result2.GetResult<VariableResultShapesResult2>())
{
    Console.WriteLine(ord.OrderID);
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

' Assign the results of the procedure with an argument
' of (1) to local variable 'result'.
Dim result As IMultipleResults = db.VariableResultShapes(1)

' Iterate through the list and write results (the company name)
' to the console.
For Each compName As VariableResultShapesResult1 _
    In result.GetResult(Of VariableResultShapesResult1)()
    Console.WriteLine(compName.CompanyName)
Next

' Pause to view company names; press Enter to continue.
Console.ReadLine()

' Assign the results of the procedure with an argument
' of (2) to local variable 'result.'
Dim result2 As IMultipleResults = db.VariableResultShapes(2)

' Iterate through the list and write results (the order IDs)
' to the console.
For Each ord As VariableResultShapesResult2 _
    In result2.GetResult(Of VariableResultShapesResult2)()
    Console.WriteLine(ord.OrderID)
Next
```

See Also

Stored Procedures

How to: Use Stored Procedures Mapped for Sequential Result Shapes

5/1/2017 • 1 min to read • Edit Online

This kind of stored procedure can generate more than one result shape, but you know in what order the results are returned. Contrast this scenario with the scenario where you do not know the sequence of the returns. For more information, see How to: Use Stored Procedures Mapped for Multiple Result Shapes.

Example

Here is the T-SQL of a stored procedure that returns multiple result shapes sequentially:

```
CREATE PROCEDURE MultipleResultTypesSequentially
AS
select * from products
select * from customers
```

```
   [Function(Name="dbo.MultipleResultTypesSequentially")]
[ResultType(typeof(MultipleResultTypesSequentiallyResult1))]
[ResultType(typeof(MultipleResultTypesSequentiallyResult2))]
public IMultipleResults MultipleResultTypesSequentially()
{
 IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.GetCurrentMethod())));
 return ((IMultipleResults)(result.ReturnValue));
}
```

```
<FunctionAttribute(Name:="dbo.MultipleResultTypesSequentially"), _
ResultType(GetType(MultipleResultTypesSequentiallyResult1)), _
ResultType(GetType(MultipleResultTypesSequentiallyResult2))> _
Public Function MultipleResultTypesSequentially() As IMultipleResults
    Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.GetCurrentMethod, MethodInfo))
    Return CType(result.ReturnValue, IMultipleResults)
End Function
```

Example

You would use code similar to the following to execute this stored procedure.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

IMultipleResults sprocResults =
    db.MultipleResultTypesSequentially();

// First read products.
foreach (Product prod in sprocResults.GetResult<Product>())
{
    Console.WriteLine(prod.ProductID);
}

// Next read customers.
foreach (Customer cust in sprocResults.GetResult<Customer>())
{
    Console.WriteLine(cust.CustomerID);
}
```

```
Dim db As New Northwnd("c:\northwnd.mdf")

Dim sprocResults As IMultipleResults = _
    db.MultipleResultTypesSequentially

' First read products.
For Each prod As Product In sprocResults.GetResult(Of Product)()
    Console.WriteLine(prod.ProductID)
Next

' Next read customers.
For Each cust As Customer In sprocResults.GetResult(Of Customer)()
    Console.WriteLine(cust.CustomerID)
Next
```

See Also

Stored Procedures

Customizing Operations By Using Stored Procedures

5/1/2017 • 3 min to read • Edit Online

Stored procedures represent a common approach to overriding default behavior. The examples in this topic show how you can use generated method wrappers for stored procedures, and how you can call stored procedures directly.

If you are using Visual Studio, you can use the Object Relational Designer to assign stored procedures to perform inserts, updates, and deletes.

NOTE

To read back database-generated values, use output parameters in your stored procedures. If you cannot use output parameters, write a partial method implementation instead of relying on overrides generated by the Object Relational Designer. Members mapped to database-generated values must be set to appropriate values after `INSERT` or `UPDATE` operations have successfully completed. For more information, see Responsibilities of the Developer In Overriding Default Behavior.

Example

Description

In the following example, assume that the `Northwind` class contains two methods to call stored procedures that are being used for overrides in a derived class.

Code

```
[Function()]
public IEnumerable<Order> CustomerOrders(
    [Parameter(Name = "CustomerID", DbType = "NChar(5)")]
    string customerID)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())),
        customerID);
    return ((IEnumerable<Order>)(result.ReturnValue));
}

[Function()]
public IEnumerable<Customer> CustomerById(
    [Parameter(Name = "CustomerID", DbType = "NChar(5)")]
    string customerID)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())),
        customerID);
    return (IEnumerable<Customer>)(result.ReturnValue);
}
```

```
<[Function]()> _
Public Function CustomerOrders( _
    <Parameter(Name:="CustomerID", DbType:="NChar(5)")> ByVal _
    customerID As String) As IEnumerable(Of Order)

    Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, _
        (CType(MethodInfo.GetCurrentMethod(), MethodInfo)), _
        customerID)
    Return CType(result.ReturnValue, IEnumerable(Of Order))
End Function

<[Function]()> _
Public Function CustomerById( _
    <Parameter(Name:="CustomerID", DbType:="NChar(5)")> ByVal _
        customerID As String) As IEnumerable(Of Customer)

    Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, _
        CType(MethodInfo.GetCurrentMethod(), MethodInfo), _
        customerID)

    Return CType(result.ReturnValue, IEnumerable(Of Customer))
End Function
```

Example

Description

The following class uses these methods for the override.

Code

```
public class NorthwindThroughSprocs : Northwnd
{

    public NorthwindThroughSprocs(string connection) :
        base(connection)
    {
    }

    // Override loading of Customer.Orders by using method wrapper.
    private IEnumerable<Order> LoadOrders(Customer customer)
    {
        return this.CustomerOrders(customer.CustomerID);
    }
    // Override loading of Order.Customer by using method wrapper.
    private Customer LoadCustomer(Order order)
    {
        return this.CustomerById(order.CustomerID).Single();
    }
    // Override INSERT operation on Customer by calling the
    // stored procedure directly.
    private void InsertCustomer(Customer customer)
    {
        // Call the INSERT stored procedure directly.
        this.ExecuteCommand("exec sp_insert_customer …");
    }
    // The UPDATE override works similarly, that is, by
    // calling the stored procedure directly.
    private void UpdateCustomer(Customer original, Customer current)
    {
        // Call the UPDATE stored procedure by using current
        // and original values.
        this.ExecuteCommand("exec sp_update_customer …");
    }
    // The DELETE override works similarly.
    private void DeleteCustomer(Customer customer)
    {
        // Call the DELETE stored procedure directly.
        this.ExecuteCommand("exec sp_delete_customer …");
    }
}
```

```
Public Class NorthwindThroughSprocs : Inherits Northwnd
    Sub New()
        MyBase.New("")
    End Sub
    ' Override loading of Customer.Orders by using method wrapper.
    Private Function LoadOrders(ByVal customer As Customer) As  _
        IEnumerable(Of Order)
        Return Me.CustomerOrders(customer.CustomerID)
    End Function

    ' Override loading of Order.Customer by using method wrapper.
    Private Function LoadCustomer(ByVal order As Order) As Customer
        Return Me.CustomerById(order.CustomerID).Single()
    End Function

    ' Override INSERT operation on Customer by calling the
    ' stored procedure directly.
    Private Sub InsertCustomer(ByVal customer As Customer)
        ' Call the INSERT stored procedure directly.
        Me.ExecuteCommand("exec sp_insert_customer …")
    End Sub

    ' The UPDATE override works similarly, that is, by
    ' calling the stored procedure directly.
    Private Sub UpdateCustomer(ByVal original As Customer, ByVal _
        current As Customer)
        ' Call the UPDATE stored procedure by using current
        ' and original values.
        Me.ExecuteCommand("exec sp_update_customer …")
    End Sub

    ' The DELETE override works similarly.
    Private Sub DeleteCustomer(ByVal customer As Customer)
        ' Call the DELETE stored procedure directly.
        Me.ExecuteCommand("exec sp_delete_customer …")
    End Sub
End Class
```

Example

Description

You can use `NorthwindThroughSprocs` exactly as you would use `Northwnd`.

Code

```
NorthwindThroughSprocs db = new NorthwindThroughSprocs("");
var custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;

foreach (Customer custObj in custQuery)
    // deferred loading of cust.Orders uses the override LoadOrders.
    foreach (Order ord in custObj.Orders)
        // ...
        // Make some changes to customers/orders.
        // Overrides for Customer are called during the execution of the
        // following:
        db.SubmitChanges();
```

```
Dim db As New NorthwindThroughSprocs()
Dim custQuery = From cust In db.Customers _
                Where cust.City = "London" _
                Select cust

For Each custObj In custQuery
    ' Deferred loading of cust.Orders uses the override LoadOrders.
    For Each ord In custObj.Orders
        ' ...
        ' Make some changes to customers/orders.
        ' Overrides for Customer are called during the execution
        ' of the following:
        db.SubmitChanges()
    Next
Next
```

See Also

Responsibilities of the Developer In Overriding Default Behavior

Customizing Operations by Using Stored Procedures Exclusively

5/1/2017 • 1 min to read • Edit Online

Access to data by using only stored procedures is a common scenario.

Example

Description

You can modify the example provided in Customizing Operations By Using Stored Procedures by replacing even the first query (which causes dynamic SQL execution) with a method call that wraps a stored procedure.

Assume `CustomersByCity` is the method, as in the following example.

Code

```
[Function()]
public IEnumerable<Customer> CustomersByCity(
    [Parameter(Name = "City", DbType = "NVarChar(15)")]
    string city)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())),
        city);
    return ((IEnumerable<Customer>)(result.ReturnValue));
}
```

```
<[Function]()> _
Public Function CustomersByCity( _
    <Parameter(Name:="City", DbType:="NVarChar(15)")> ByVal _
        city As String) As IEnumerable(Of Customer)

    Dim result = Me.ExecuteMethodCall(Me, _
        (CType(MethodInfo.GetCurrentMethod(), IEnumerable(Of  _
        Customer))), city)
    Return CType(result.ReturnValue, IEnumerable(Of Customer))
End Function
```

The following code executes without any dynamic SQL.

```
NorthwindThroughSprocs db = new NorthwindThroughSprocs("...");
// Use a method call (stored procedure wrapper) instead of
// a LINQ query against the database.
var custQuery =
    db.CustomersByCity("London");

foreach (Customer custObj in custQuery)
{
    // Deferred loading of custObj.Orders uses the override
    // LoadOrders. There is no dynamic SQL.
    foreach (Order ord in custObj.Orders)
    {
        // Make some changes to customers/orders.
        // Overrides for Customer are called during the execution
        // of the following.
    }
}
db.SubmitChanges();
```

```
Dim db As New Northwnd("...")
' Use a method call (stored procedure wrapper) instead of
' a LINQ query against the database.
Dim custQuery = db.CustomersByCity("London")

For Each custObj In custQuery
    ' Deferred loading of custObj.Orders uses the override
    ' LoadOrders. There is no dynamic SQL.

    For Each ord In custObj.Orders
        ' Make some changes to customers/orders.
        ' Overrides for Customer are called during the execution
        ' of the following:
        db.SubmitChanges()
    Next
Next
```

See Also

Responsibilities of the Developer In Overriding Default Behavior

Transaction Support

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL supports three distinct transaction models. The following lists these models in the order of checks performed.

Explicit Local Transaction

When SubmitChanges is called, if the Transaction property is set to a (`IDbTransaction`) transaction, the SubmitChanges call is executed in the context of the same transaction.

It is your responsibility to commit or rollback the transaction after successful execution of the transaction. The connection corresponding to the transaction must match the connection used for constructing the DataContext. An exception is thrown if a different connection is used.

Explicit Distributable Transaction

You can call LINQ to SQL APIs (including but not limited to SubmitChanges) in the scope of an active Transaction. LINQ to SQL detects that the call is in the scope of a transaction and does not create a new transaction. LINQ to SQL also avoids closing the connection in this case. You can perform query and SubmitChanges executions in the context of such a transaction.

Implicit Transaction

When you call SubmitChanges, LINQ to SQL checks to see whether the call is in the scope of a Transaction or if the `Transaction` property (`IDbTransaction`) is set to a user-started local transaction. If it finds neither transaction, LINQ to SQL starts a local transaction (`IDbTransaction`) and uses it to execute the generated SQL commands. When all SQL commands have been successfully completed, LINQ to SQL commits the local transaction and returns.

See Also

Background Information
How to: Bracket Data Submissions by Using Transactions

SQL-CLR Type Mismatches

5/1/2017 • 14 min to read • Edit Online

LINQ to SQL automates much of the translation between the object model and SQL Server. Nevertheless, some situations prevent exact translation. These key mismatches between the common language runtime (CLR) types and the SQL Server database types are summarized in the following sections. You can find more details about specific type mappings and function translation at SQL-CLR Type Mapping and Data Types and Functions.

Data Types

Translation between the CLR and SQL Server occurs when a query is being sent to the database, and when the results are sent back to your object model. For example, the following Transact-SQL query requires two value conversions:

```
Select DateOfBirth From Customer Where CustomerId = @id
```

Before the query can be executed on SQL Server, the value for the Transact-SQL parameter must be specified. In this example, the `id` parameter value must first be translated from a CLR System.Int32 type to a SQL Server `INT` type so that the database can understand what the value is. Then to retrieve the results, the SQL Server `DateOfBirth` column must be translated from a SQL Server `DATETIME` type to a CLR System.DateTime type for use in the object model. In this example, the types in the CLR object model and SQL Server database have natural mappings. But, this is not always the case.

Missing Counterparts

The following types do not have reasonable counterparts.

- Mismatches in the CLR System namespace:
  - **Unsigned integers**. These types are typically mapped to their signed counterparts of larger size to avoid overflow. Literals can be converted to a signed numeric of the same or smaller size, based on value.
  - **Boolean**. These types can be mapped to a bit or larger numeric or string. A literal can be mapped to an expression that evaluates to the same value (for example, `1=1` in SQL for `True` in CLS).
  - **TimeSpan**. This type represents the difference between two `DateTime` values and does not correspond to the `timestamp` of SQL Server. The CLR System.TimeSpan may also map to the SQL Server `TIME` type in some cases. The SQL Server `TIME` type was only intended to represent positive values less than 24 hours. The CLR TimeSpan has a much larger range.

  NOTE

  SQL Server-specific .NET Framework types in System.Data.SqlTypes are not included in this comparison.

- Mismatches in SQL Server:
  - **Fixed length character types**. Transact-SQL distinguishes between Unicode and non-Unicode categories and has three distinct types in each category: fixed length `nchar`/`char`, variable length `nvarchar`/`varchar`, and larger-sized `ntext`/`text`. The fixed length character types could be mapped to the CLR System.Char type for retrieving characters, but they do not really correspond to

the same type in conversions and behavior.

- **Bit**. Although the `bit` domain has the same number of values as `Nullable<Boolean>`, the two are different types. `Bit` takes values `1` and `0` instead of `true` / `false`, and cannot be used as an equivalent to Boolean expressions.

- **Timestamp**. Unlike the CLR System.TimeSpan type, the SQL Server `TIMESTAMP` type represents an 8-byte number generated by the database that is unique for each update and is not based on the difference between DateTime values.

- **Money** and **SmallMoney**. These types can be mapped to Decimal but are basically different types and are treated as such by server-based functions and conversions.

Multiple Mappings

There are many SQL Server data types that you can map to one or more CLR data types. There are also many CLR types that you can map to one or more SQL Server types. Although a mapping may be supported by LINQ to SQL, it does not mean that the two types mapped between the CLR and SQL Server are a perfect match in precision, range, and semantics. Some mappings may include differences in any or all of these dimensions. You can find details about these potential differences for the various mapping possibilities at SQL-CLR Type Mapping.

User-defined Types

User-defined CLR types are designed to help bridge the type system gap. Nevertheless they surface interesting issues about type versioning. A change in the version on the client might not be matched by a change in the type stored on the database server. Any such change causes another type mismatch where the type semantics might not match and the version gap is likely to become visible. Further complications occur as inheritance hierarchies are refactored in successive versions.

Expression Semantics

In addition to the pairwise mismatch between CLR and database types, expressions add complexity to the mismatch. Mismatches in operator semantics, function semantics, implicit type conversion, and precedence rules must be considered.

The following subsections illustrate the mismatch between apparently similar expressions. It might be possible to generate SQL expressions that are semantically equivalent to a given CLR expression. However, it is not clear whether the semantic differences between apparently similar expressions are evident to a CLR user, and therefore whether the changes that are required for semantic equivalence are intended or not. This is an especially critical issue when an expression is evaluated for a set of values. The visibility of the difference might depend on data- and be hard to identify during coding and debugging.

Null Semantics

SQL expressions provide three-valued logic for Boolean expressions. The result can be true, false, or null. By contrast, CLR specifies two-valued Boolean result for comparisons involving null values. Consider the following code:

```
Nullable<int> i = null;
Nullable<int> j = null;
if (i == j)
{
    // This branch is executed.
}
```

```
Dim i? As Integer = Nothing
Dim j? As Integer = Nothing
If i = j Then
    '  This branch is executed.
End If
```

```
-- Assume col1 and col2 are integer columns with null values.
-- Assume that ANSI null behavior has not been explicitly
--    turned off.
Select …
From …
Where col1 = col2
-- Evaluates to null, not true and the corresponding row is not
--     selected.
-- To obtain matching behavior (i -> col1, j -> col2) change
--     the query to the following:
Select …
From …
Where
    col1 = col2
or (col1 is null and col2 is null)
-- (Visual Basic 'Nothing'.)
```

A similar problem occurs with the assumption about two-valued results.

```
if ((i == j) || (i != j)) // Redundant condition.
{
    // ...
}
```

```
If (i = j) Or (i <> j) Then ' Redundant condition.
    ' ...
End If
```

```
-- Assume col1 and col2 are nullable columns.
-- Assume that ANSI null behavior has not been explicitly
--     turned off.
Select …
From …
Where
    col1 = col2
or col1 != col2
-- Visual Basic: col1 <> col2.

-- Excludes the case where the boolean expression evaluates
--     to null. Therefore the where clause does not always
--     evaluate to true.
```

In the previous case, you can get equivalent behavior in generating SQL, but the translation might not accurately reflect your intention.

LINQ to SQL does not impose C# `null` or Visual Basic `nothing` comparison semantics on SQL. Comparison operators are syntactically translated to their SQL equivalents. The semantics reflect SQL semantics as defined by server or connection settings. Two null values are considered unequal under default SQL Server settings (although you can change the settings to change the semantics). Regardless, LINQ to SQL does not consider server settings in query translation.

A comparison with the literal `null` (`nothing`) is translated to the appropriate SQL version (`is null` or `is not null`).

The value of `null` (`nothing`) in collation is defined by SQL Server; LINQ to SQL does not change the collation.

Type Conversion and Promotion

SQL supports a rich set of implicit conversions in expressions. Similar expressions in C# would require an explicit cast. For example:

- `Nvarchar` and `DateTime` types can be compared in SQL without any explicit casts; C# requires explicit conversion.
- `Decimal` is implicitly converted to `DateTime` in SQL. C# does not allow for an implicit conversion.

Likewise, type precedence in Transact-SQL differs from type precedence in C# because the underlying set of types is different. In fact, there is no clear subset/superset relationship between the precedence lists. For example, comparing an `nvarchar` with a `varchar` causes the implicit conversion of the `varchar` expression to `nvarchar`. The CLR provides no equivalent promotion.

In simple cases, these differences cause CLR expressions with casts to be redundant for a corresponding SQL expression. More importantly, the intermediate results of a SQL expression might be implicitly promoted to a type that has no accurate counterpart in C#, and vice versa. Overall, the testing, debugging, and validation of such expressions adds significant burden on the user.

Collation

Transact-SQL supports explicit collations as annotations to character string types. These collations determine the validity of certain comparisons. For example, comparing two columns with different explicit collations is an error. The use of much simplified CTS string type does not cause such errors. Consider the following example:

```
create table T2 (
    Col1 nvarchar(10),
    Col2      nvarchar(10) collate Latin_general_ci_as
)
```

```
class C
{
string s1;       // Map to T2.Col1.
string s2;       // Map to T2.Col2.

    void Compare()
    {
        if (s1 == s2) // This is correct.
        {
            // ...
        }
    }
}
```

```
Class C
    Dim s1 As String    ' Map to T2.Col1.
    Dim s2 As String    ' Map to T2.Col2.
    Sub Compare()
        If s1 = s2 Then ' This is correct.
            ' ...
        End If
    End Sub
End Class
```

```
Select …
From …
Where Col1 = Col2
-- Error, collation conflict.
```

In effect, the collation subclause creates a *restricted type* that is not substitutable.

Similarly, the sort order can be significantly different across the type systems. This difference affects the sorting of results. Guid is sorted on all 16 bytes by lexicographic order (`IComparable()`), whereas T-SQL compares GUIDs in the following order: node(10-15), clock-seq(8-9), time-high(6-7), time-mid(4-5), time-low(0-3). This ordering was done in SQL 7.0 when NT-generated GUIDs had such an octet order. The approach ensured that GUIDs generated at the same node cluster came together in sequential order according to timestamp. The approach was also useful for building indexes (inserts become appends instead of random IOs). The order was scrambled later in Windows because of privacy concerns, but SQL must maintain compatibility. A workaround is to use SqlGuid instead of Guid.

Operator and Function Differences

Operators and functions that are essentially comparable have subtly different semantics. For example:

- C# specifies short circuit semantics based on lexical order of operands for logical operators `&&` and `||`. SQL on the other hand is targeted for set-based queries and therefore provides more freedom for the optimizer to decide the order of execution. Some of the implications include the following:
    - Semantically equivalent translation would require " `CASE` … `WHEN` … `THEN` " construct in SQL to avoid reordering of operand execution.
    - A loose translation to `AND` / `OR` operators could cause unexpected errors if the C# expression relies on evaluation the second operand being based on the result of the evaluation of the first operand.
- `Round()` function has different semantics in .NET Framework and in T-SQL.
- Starting index for strings is 0 in the CLR but 1 in SQL. Therefore, any function that has index needs index translation.
- The CLR supports modulus ('%') operator for floating point numbers but SQL does not.
- The `Like` operator effectively acquires automatic overloads based on implicit conversions. Although the `Like` operator is defined to operate on character string types, implicit conversion from numeric types or `DateTime` types allows for those non-string types to be used with `Like` just as well. In CTS, comparable implicit conversions do not exist. Therefore, additional overloads are needed.

  > **NOTE**
  >
  > This `Like` operator behavior applies to C# only; the Visual Basic `Like` keyword is unchanged.

- Overflow is always checked in SQL but it has to be explicitly specified in C# (not in Visual Basic) to avoid wraparound. Given integer columns C1, C2 and C3, if C1+C2 is stored in C3 (Update T Set C3 = C1 + C2).

  ```
  create table T3 (
      Col1      integer,
      Col2      integer
  )
  insert into T3 (col1, col2) values (2147483647, 5)
  -- Valid values: max integer value and 5.
  select * from T3 where col1 + col2 < 0
  -- Produces arithmetic overflow error.
  ```

```
// C# overflow in absence of explicit checks.
int i = Int32.MaxValue;
int j = 5;
if (i+j < 0) Console.WriteLine("Overflow!");
// This code prints the overflow message.
```

```
' Does not apply.
' Visual Basic overflow in absence of implicit check
' (turn off overflow checks in compiler options)
Dim I As Integer = Int32.MaxValue
Dim j As Integer = 5
If I + j < 0 Then
    ' This code prints the overflow message.
    Console.WriteLine("Overflow!")
End If
```

- SQL performs symmetric arithmetic rounding while .NET Framework uses banker's rounding. See Knowledgebase article 196652 for additional details.

- By default, for common locales, character-string comparisons are case-insensitive in SQL. In Visual Basic and in C#, they are case-sensitive. For example, `s == "Food"` (`s = "Food"` in Visual Basic) and `s == "Food"` can yield different results if `s` is `food`.

  ```
  -- Assume default US-English locale (case insensitive).
  create table T4 (
      Col1      nvarchar (256)
  )
  insert into T4 values (‘Food’)
  insert into T4 values (‘FOOD’)
  select * from T4 where Col1 = ‘food’
  -- Both the rows are returned because of case-insensitive matching.
  ```

```
// C# equivalent on collections of Strings in place of nvarchars.
String[] strings = { "food", "FOOD" };
foreach (String s in strings)
{
    if (s == "food")
    {
        Console.WriteLine(s);
    }
}
// Only "food" is returned.
```

```
' Visual Basic equivalent on collections of Strings in place of
' nvarchars.
Dim strings() As String = {"food", "FOOD"}
For Each s As String In strings
    If s = "food" Then
        Console.WriteLine(s)
    End If
Next
' Only "food" is returned.
```

- Operators/ functions applied to fixed length character type arguments in SQL have significantly different semantics than the same operators/functions applied to the CLR System.String. This could also be viewed as an extension of the missing counterpart problem discussed in the section about types.

```
create table T4 (
    Col1      nchar(4)
)
Insert into T5(Col1) values ('21');
Insert into T5(Col1) values ('1021');
Select * from T5 where Col1 like '%1'
-- Only the second row with Col1 = '1021' is returned.
-- Not the first row!
```

```
// Assume Like(String, String) method.
string s = ""; // map to T4.Col1
if (System.Data.Linq.SqlClient.SqlMethods.Like(s, "%1"))
{
    Console.WriteLine(s);
}
// Expected to return true for both "21" and "1021"
```

```
' Assume Like(String, String) method.
Dim s As String    ' Map to T4.Col1.
If s Like (System.Data.Linq.SqlClient.SqlMethods.Like(s, "%1")) Then
    Console.WriteLine(s)
End If
' Expected to return true for both "21" and "1021".
```

A similar problem occurs with string concatenation.

```
create table T6 (
    Col1      nchar(4)
    Col2       nchar(4)
)
Insert into T6 values ('a', 'b');
Select Col1+Col2 from T6
-- Returns concatenation of padded strings "a   b   " and not "ab".
```

In summary, a convoluted translation might be required for CLR expressions and additional operators/functions may be necessary to expose SQL functionality.

Type Casting

In C# and in SQL, users can override the default semantics of expressions by using explicit type casts (`Cast` and `Convert`). However, exposing this capability across the type system boundary poses a dilemma. A SQL cast that provides the desired semantics cannot be easily translated to a corresponding C# cast. On the other hand, a C# cast cannot be directly translated into an equivalent SQL cast because of type mismatches, missing counterparts, and different type precedence hierarchies. There is a trade-off between exposing the type system mismatch and losing significant power of expression.

In other cases, type casting might not be needed in either domain for validation of an expression but might be required to make sure that a non-default mapping is correctly applied to the expression.

```
-- Example from "Non-default Mapping" section extended
create table T5 (
    Col1      nvarchar(10),
    Col2      nvarchar(10)
)
Insert into T5(col1, col2) values (‘3’, ‘2’);
```

```
class C
{
    int x;        // Map to T5.Col1.
    int y;        // Map to T5.Col2.

    void Casting()
    {
        // Intended predicate.
        if (x + y > 4)
        {
            // valid for the data above
        }
    }
}
```

```
Class C
    Dim x As Integer        ' Map to T5.Col1.
    Dim y As Integer        ' Map to T5.Col2.

    Sub Casting()
        ' Intended predicate.
        If (x + y) > 4 Then
            ' Valid for the data above.
        End If
    End Sub
End Class
```

```
Select *
From T5
Where Col1 + Col2 > 4
-- "Col1 + Col2" expr evaluates to '32'
```

Performance Issues

Accounting for some SQL Server-CLR type differences may resut in a decrease in performance when crossing between the CLR and SQL Server type systems. Examples of scenarios impacting performance include the following:

- Forced order of evaluation for logical and/or operators
- Generating SQL to enforce order of predicate evaluation restricts the SQL optimizer's ability.
- Type conversions, whether introduced by a CLR compiler or by an Object-Relational query implementation, may curtail index usage.

  For example,

  ```
  -- Table DDL
  create table T5 (
      Col1      varchar(100)
  )
  ```

  ```
  class C5
  {
      string s;        // Map to T5.Col1.
  }
  ```

```
Class C5
    Dim s As String ' Map to T5.Col1.
End Class
```

Consider the translation of expression `(s = SOME_STRING_CONSTANT)`.

```
-- Corresponding part of SQL where clause
Where …
Col1 = SOME_STRING_CONSTANT
-- This expression is of the form <varchar> = <nvarchar>.
-- Hence SQL introduces a conversion from varchar to nvarchar,
--     resulting in
Where …
Convert(nvarchar(100), Col1) = SOME_STRING_CONSTANT
-- Cannot use the index for column Col1 for some implementations.
```

In addition to semantic differences, it is important to consider impacts to performance when crossing between the SQL Server and CLR type systems. For large data sets, such performance issues can determine whether an application is deployable.

See Also

Background Information

SQL-CLR Custom Type Mappings

5/10/2017 • 2 min to read • Edit Online

Type mapping between SQL Server and the common language runtime (CLR) is automatically specified when you use the SQLMetal command-line tool, Object Relational Designer (O/R Designer).

When no customized mapping is performed, these tools assign default type mappings as described in SQL-CLR Type Mapping. If you want to type mappings differently from these defaults, you need to do some customization of the type mappings.

When customizing type mappings, the recommended approach is to make the changes in an intermediary DBML file. Then, your customized DBML file should be used when you create you code and mapping files with SQLMetal or O/R Designer.

Once you instantiate the DataContext object from the code and mapping files, the System.Data.Linq.DataContext.CreateDatabase method creates a database based on the type mappings that are specified. If there are no CLR `type` attributes specified in the mappings, the default type mappings will be used.

Customization with SQLMetal or O/R Designer

With SQLMetal and O/R Designer, you can automatically create an object model that includes the type mapping information inside or outside the code file. Because these files are overwritten by SQLMetal or O/R Designer, each time you recreate your mappings, the recommended approach to specifying custom type mappings is to customize a DBML file.

To customize type mappings with SQLMetal or O/R Designer, first generate a DBML file. Then, before generating the code file or mapping file, modify the DBML file to identify the desired type mappings. With SQLMetal, you have to manually change the `Type` and `DbType` attributes in the DBML file to make your type mapping customizations. With O/R Designer, you can make your changes within the Designer. For more information about using the O/R Designer, see LINQ to SQL Tools in Visual Studio.

NOTE
Some type mappings may result in overflow or data loss exceptions while translating to or from the database. Carefully review the Type Mapping Run-time Behavior Matrix in SQL-CLR Type Mapping before making any customizations.

In order for your type mapping customizations to be recognized by SQLMetal or O/R Designer, you need to make sure that these tools are supplied with the path to your custom DBML file when you generate your code file or external mapping file. Although not required for type mapping customization, it is recommended that you always separate your type mapping information from your code file and generate the additional external type mapping file. Doing so will leave some flexibility by not requiring that the code file be recompiled.

Incorporating Database Changes

When your database changes, you will need to update your DBML file to reflect those changes. One way to do this is to automatically create a new DBML file and then re-do your type mapping customizations. Alternatively, you could compare the differences between your new DBML file and your customized DBML file and update your custom DBML file manually to reflect the database change.

See Also

SQL-CLR Type Mapping

Code Generation in LINQ to SQL

User-Defined Functions

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL uses methods in your object model to represent user-defined functions. You designate methods as functions by applying the FunctionAttribute attribute and, where required, the ParameterAttribute attribute. For more information, see The LINQ to SQL Object Model.

To avoid an InvalidOperationException, user-defined functions in LINQ to SQL must be in one of the following forms:

- A function wrapped as a method call having the correct mapping attributes. For more information, see Attribute-Based Mapping.
- A static SQL method specific to LINQ to SQL.
- A function supported by a .NET Framework method.

The topics in this section show how to form and call these methods in your application if you write the code yourself. Developers using Visual Studio would typically use the Object Relational Designer to map user-defined functions.

In This Section

How to: Use Scalar-Valued User-Defined Functions
Describes how to implement a function that returns scalar values.

How to: Use Table-Valued User-Defined Functions
Describes how to implement a function that returns table values.

How to: Call User-Defined Functions Inline
Describes how to make inline calls to functions and the differences in execution when the call is made inline.

How to: Use Scalar-Valued User-Defined Functions

5/1/2017 • 1 min to read • Edit Online

You can map a client method defined on a class to a user-defined function by using the FunctionAttribute attribute. Note that the body of the method constructs an expression that captures the intent of the method call, and passes that expression to the DataContext for translation and execution.

NOTE

Direct execution occurs only if the function is called outside a query. For more information, see How to: Call User-Defined Functions Inline.

Example

The following SQL code presents a scalar-valued user-defined function `ReverseCustName()`.

```
CREATE FUNCTION ReverseCustName(@string varchar(100))
RETURNS varchar(100)
AS
BEGIN
    DECLARE @custName varchar(100)
    -- Implementation left as exercise for users.
    RETURN @custName
END
```

You would map a client method such as the following for this code:

```
[Function(Name = "dbo.ReverseCustName", IsComposable = true)]
[return: Parameter(DbType = "VarChar(100)")]
public string ReverseCustName([Parameter(Name = "string",
    DbType = "VarChar(100)")] string @string)
{
    return ((string)(this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())),
        @string).ReturnValue));
}
```

```
<FunctionAttribute(Name:="dbo.ReverseCustName", _
IsComposable:=True)> _
Public Function ReverseCustName(<Parameter(Name:="string", _
DbType:="VarChar(100)")> ByVal [string] As String) As _
<Parameter(DbType:="VarChar(100)")> String
    Return CType(Me.ExecuteMethodCall(Me, _
        CType(MethodInfo.GetCurrentMethod, MethodInfo), _
        [string]).ReturnValue, String)
End Function
```

See Also

User-Defined Functions

How to: Use Table-Valued User-Defined Functions

5/1/2017 • 1 min to read • Edit Online

A table-valued function returns a single rowset (unlike stored procedures, which can return multiple result shapes). Because the return type of a table-valued function is `Table`, you can use a table-valued function anywhere in SQL that you can use a table. You can also treat the table-valued function just as you would a table.

Example

The following SQL function explicitly states that it returns a `TABLE`. Therefore, the returned rowset structure is implicitly defined.

```
CREATE FUNCTION ProductsCostingMoreThan(@cost money)
RETURNS TABLE
AS
RETURN
    SELECT ProductID, UnitPrice
    FROM Products
    WHERE UnitPrice > @cost
```

LINQ to SQL maps the function as follows:

```
   [Function(Name="dbo.ProductsCostingMoreThan", IsComposable=true)]
public IQueryable<ProductsCostingMoreThanResult> ProductsCostingMoreThan([Parameter(DbType="Money")]
System.Nullable<decimal> cost)
{
 return this.CreateMethodCallQuery<ProductsCostingMoreThanResult>(this, ((MethodInfo)
(MethodInfo.GetCurrentMethod())), cost);
}
```

```
   <FunctionAttribute(Name:="dbo.ProductsCostingMoreThan", IsComposable:=True)> _
Public Function ProductsCostingMoreThan(<Parameter(DbType:="Money")> ByVal cost As System.Nullable(Of Decimal))
As IQueryable(Of ProductsCostingMoreThanResult)
       Return Me.CreateMethodCallQuery(Of ProductsCostingMoreThanResult)(Me, CType(MethodInfo.GetCurrentMethod,
MethodInfo), cost)
   End Function
```

Example

The following SQL code shows that you can join to the table that the function returns and otherwise treat it as you would any other table:

```
SELECT p2.ProductName, p1.UnitPrice
FROM dbo.ProductsCostingMoreThan(80.50)
AS p1 INNER JOIN Products AS p2 ON p1.ProductID = p2.ProductID
```

In LINQ to SQL, the query would be rendered as follows:

```
        var q =
from p in db.ProductsCostingMoreThan(80.50m)
join s in db.Products on p.ProductID equals s.ProductID
select new { p.ProductID, s.UnitPrice };
```

```
    Dim q = _
From p In db.ProductsCostingMoreThan(80.5), p1 In db.Products _
Where p.ProductID = p1.ProductID _
Select p.ProductID, p1.UnitPrice
```

See Also

User-Defined Functions

How to: Call User-Defined Functions Inline

5/1/2017 • 1 min to read • Edit Online

Although you can call user-defined functions inline, functions that are included in a query whose execution is deferred are not executed until the query is executed. For more information, see Introduction to LINQ Queries (C#).

When you call the same function outside a query, LINQ to SQL creates a simple query from the method call expression. The following is the SQL syntax (the parameter `@p0` is bound to the constant passed in):

```
SELECT dbo.ReverseCustName(@p0)
```

LINQ to SQL creates the following:

```
string str = db.ReverseCustName("LINQ to SQL");
```

```
Dim str As String = db.ReverseCustName("LINQ to SQL")
```

Example

In the following LINQ to SQL query, you can see an inline call to the generated user-defined function method `ReverseCustName`. The function is not executed immediately because query execution is deferred. The SQL built for this query translates to a call to the user-defined function in the database (see the SQL code following the query).

```
var custQuery =
    from cust in db.Customers
    select new {cust.ContactName, Title =
        db.ReverseCustName(cust.ContactTitle)};
```

```
Dim custQuery = _
    From cust In db.Customers _
    Select cust.ContactName, Title = _
    db.ReverseCustName(cust.ContactTitle)
```

```
SELECT [t0].[ContactName],
    dbo.ReverseCustName([t0].[ContactTitle]) AS [Title]
FROM [Customers] AS [t0]
```

See Also

User-Defined Functions

Reference

5/1/2017 • 1 min to read • Edit Online

This section provides reference information for LINQ to SQL developers.

You are also encouraged to search the MSDN Library for specific issues, and especially to participate in the LINQ Forum, where you can discuss more complex topics in detail with experts. In addition, you can study a white paper detailing LINQ to SQL technology, complete with Visual Basic and C# code examples. For more information, see LINQ to SQL: .NET Language-Integrated Query for Relational Data.

In This Section

Data Types and Functions
Describes how common language runtime (CLR) constructs have corresponding expressions in SQL only where LINQ to SQL has explicitly provided a conversion in the translation engine.

Attribute-Based Mapping
Describes the LINQ to SQL attribute-based approach to mapping a LINQ to SQL object model to a SQL Server database.

Code Generation in LINQ to SQL
Describes how LINQ to SQL obtains meta information from a database and then generates code files.

External Mapping
Describes the LINQ to SQL external-mapping approach to mapping a LINQ to SQL object model to a SQL Server database. Provides the XSD schema definition for mapping files.

Frequently Asked Questions
Provides answers to common questions regarding LINQ to SQL.

SQL Server Compact and LINQ to SQL
Describes how SQL Server Compact 3.5 differs from SQL Server in LINQ to SQL applications.

Standard Query Operator Translation
Describes how LINQ to SQL translates Standard Query Operators to SQL commands.

Related Sections

LINQ to SQL
Provides a portal for LINQ to SQL topics.

LINQ (Language-Integrated Query)
Provides a portal for LINQ topics.

LinqDataSource Technology Overview
Describes how the LinqDataSource control exposes LINQ to Web developers through the ASP.NET data-source control architecture.

Data Types and Functions

5/1/2017 • 1 min to read • Edit Online

The topics listed in the following table describe LINQ to SQL support for members, constructs, and casts of the common language runtime (CLR). Supported members and constructs are available to use in your LINQ to SQL queries.

An unsupported item in the table means that LINQ to SQL cannot translate the CLR member, construct, or cast for execution on the SQL Server. You may still be able to use them in your code, but they must be evaluated before the query is translated to Transact-SQL or after the results have been retrieved from the database.

TOPIC	DESCRIPTION
SQL-CLR Type Mapping	Provides a detailed matrix of mappings between CLR types and SQL Server types.
Basic Data Types	Summarizes differences in behavior from the .NET Framework.
Boolean Data Types	Summarizes differences in behavior from the .NET Framework.
Null Semantics	Provides links to LINQ to SQL topics that discuss null and nullable issues.
Numeric and Comparison Operators	Summarizes differences in behavior from the .NET Framework.
Sequence Operators	Summarizes differences in behavior from the .NET Framework.
System.Convert Methods	Summarizes differences in behavior from the .NET Framework.
System.DateTime Methods	Describes LINQ to SQL support for members of the System.DateTime structure.
System.DateTimeOffset Methods	Describes LINQ to SQL support for members of the System.DateTimeOffset structure.
System.Math Methods	Summarizes differences in behavior from the .NET Framework.
System.Object Methods	Summarizes differences in behavior from the .NET Framework.
System.String Methods	Summarizes differences in behavior from the .NET Framework.
System.TimeSpan Methods	Describes LINQ to SQL support for members of the System.TimeSpan structure.

TOPIC	DESCRIPTION
Unsupported Functionality	Describes functionality that is not supported in LINQ to SQL.

See Also

SQL-CLR Type Mismatches
Reference
.NET Framework Class Library in Visual Studio

SQL-CLR Type Mapping

5/31/2017 • 12 min to read • Edit Online

In LINQ to SQL, the data model of a relational database maps to an object model that is expressed in the programming language of your choice. When the application runs, LINQ to SQL translates the language-integrated queries in the object model into SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates the results back to objects that you can work with in your own programming language.

In order to translate data between the object model and the database, a *type mapping* must be defined. LINQ to SQL uses a type mapping to match each common language runtime (CLR) type with a particular SQL Server type. You can define type mappings and other mapping information, such as database structure and table relationships, inside the object model with attribute-based mapping. Alternatively, you can specify the mapping information outside the object model with an external mapping file. For more information, see Attribute-Based Mapping and External Mapping.

This topic discusses the following points:

- Default Type Mapping
- Type Mapping Run-time Behavior Matrix
- Behavior Differences Between CLR and SQL Execution
- Enum Mapping
- Numeric Mapping
- Text and XML Mapping
- Date and Time Mapping
- Binary Mapping
- Miscellaneous Mapping

Default Type Mapping

You can create the object model or external mapping file automatically with the Object Relational Designer (O/R Designer) or the SQLMetal command-line tool. The default type mappings for these tools define which CLR types are chosen to map to columns inside the SQL Server database. For more information about using these tools, see Creating the Object Model.

You can also use the CreateDatabase method to create a SQL Server database based on the mapping information in the object model or external mapping file. The default type mappings for the CreateDatabase method define which type of SQL Server columns are created to map to the CLR types in the object model. For more information, see How to: Dynamically Create a Database.

Type Mapping Run-time Behavior Matrix

The following diagram shows the expected run-time behavior of specific type mappings when data is retrieved from or saved to the database. With the exception of serialization, LINQ to SQL does not support mapping between any CLR or SQL Server data types that are not specified in this matrix. For more information on serialization support, see Binary Serialization.

NOTE

Some type mappings may result in overflow or data loss exceptions while translating to or from the database.

Custom Type Mapping

With LINQ to SQL, you are not limited to the default type mappings used by the O/R Designer, SQLMetal, and the CreateDatabase method. You can create custom type mappings by explicitly specifying them in a DBML file. Then you can use that DBML file to create the object model code and mapping file. For more information, see SQL-CLR Custom Type Mappings.

Behavior Differences Between CLR and SQL Execution

Because of differences in precision and execution between the CLR and SQL Server, you may receive different results or experience different behavior depending on where you perform your calculations. Calculations performed in LINQ to SQL queries are actually translated to Transact-SQL and then executed on the SQL Server database. Calculations performed outside LINQ to SQL queries are executed within the context of the CLR.

For example, the following are some differences in behavior between the CLR and SQL Server:

- SQL Server orders some data types differently than data of equivalent type in the CLR. For example, SQL Server data of type `UNIQUEIDENTIFIER` is ordered differently than CLR data of type System.Guid.
- SQL Server handles some string comparison operations differently than the CLR. In SQL Server, string comparison behavior depends on the collation settings on the server. For more information, see Working with Collations in the Microsoft SQL Server Books Online.
- SQL Server may return different values for some mapped functions than the CLR. For example, equality functions will differ because SQL Server considers two strings to be equal if they only differ in trailing white space; whereas the CLR considers them to be not equal.

Enum Mapping

LINQ to SQL supports mapping the CLR System.Enum type to SQL Server types in two ways:

- Mapping to SQL numeric types (`TINYINT` , `SMALLINT` , `INT` , `BIGINT`)

  When you map a CLR System.Enum type to a SQL numeric type, you map the underlying integer value of the CLR System.Enum to the value of the SQL Server database column. For example, if a System.Enum named `DaysOfWeek` contains a member named `Tue` with an underlying integer value of 3, that member maps to a database value of 3.

- Mapping to SQL text types (`CHAR` , `NCHAR` , `VARCHAR` , `NVARCHAR`)

  When you map a CLR System.Enum type to a SQL text type, the SQL database value is mapped to the names of the CLR System.Enum members. For example, if a System.Enum named `DaysOfWeek` contains a member named `Tue` with an underlying integer value of 3, that member maps to a database value of `Tue` .

NOTE

When mapping SQL text types to a CLR System.Enum, include only the names of the Enum members in the mapped SQL column. Other values are not supported in the Enum-mapped SQL column.

The O/R Designer and SQLMetal command-line tool cannot automatically map a SQL type to a CLR Enum class. You must explicitly configure this mapping by customizing a DBML file for use by the O/R Designer and

SQLMetal. For more information about custom type mapping, see SQL-CLR Custom Type Mappings.

Because a SQL column intended for enumeration will be of the same type as other numeric and text columns; these tools will not recognize your intent and default to mapping as described in the following Numeric Mapping and Text and XML Mapping sections. For more information about generating code with the DBML file, see Code Generation in LINQ to SQL.

The System.Data.Linq.DataContext.CreateDatabase method creates a SQL column of numeric type to map a CLR System.Enum type.

Numeric Mapping

LINQ to SQL lets you map many CLR and SQL Server numeric types. The following table shows the CLR types that O/R Designer and SQLMetal select when building an object model or external mapping file based on your database.

SQL SERVER TYPE	DEFAULT CLR TYPE MAPPING USED BY O/R DESIGNER AND SQLMETAL
`BIT`	System.Boolean
`TINYINT`	System.Int16
`INT`	System.Int32
`BIGINT`	System.Int64
`SMALLMONEY`	System.Decimal
`MONEY`	System.Decimal
`DECIMAL`	System.Decimal
`NUMERIC`	System.Decimal
`REAL/FLOAT(24)`	System.Single
`FLOAT/FLOAT(53)`	System.Double

The next table shows the default type mappings used by the System.Data.Linq.DataContext.CreateDatabase method to define which type of SQL columns are created to map to the CLR types defined in your object model or external mapping file.

CLR TYPE	DEFAULT SQL SERVER TYPE USED BY SYSTEM.DATA.LINQ.DATACONTEXT.CREATEDATABASE
System.Boolean	`BIT`
System.Byte	`TINYINT`
System.Int16	`SMALLINT`
System.Int32	`INT`

CLR TYPE	DEFAULT SQL SERVER TYPE USED BY SYSTEM.DATA.LINQ.DATACONTEXT.CREATEDATABASE
System.Int64	`BIGINT`
System.SByte	`SMALLINT`
System.UInt16	`INT`
System.UInt32	`BIGINT`
System.UInt64	`DECIMAL(20)`
System.Decimal	`DECIMAL(29,4)`
System.Single	`REAL`
System.Double	`FLOAT`

There are many other numeric mappings you can choose, but some may result in overflow or data loss exceptions while translating to or from the database. For more information, see the Type Mapping Run Time Behavior Matrix.

Decimal and Money Types

The default precision of SQL Server `DECIMAL` type (18 decimal digits to the left and right of the decimal point) is much smaller than the precision of the CLR `Decimal` type that it is paired with by default. This can result in precision loss when you save data to the database. However, just the opposite can happen if the SQL Server `DECIMAL` type is configured with greater than 29 digits of precision. When a SQL Server `DECIMAL` type has been configured with a greater precision than the CLR System.Decimal, precision loss can occur when retrieving data from the database.

The SQL Server `MONEY` and `SMALLMONEY` types, which are also paired with the CLR System.Decimal type by default, have a much smaller precision, which can result in overflow or data loss exceptions when saving data to the database.

Text and XML Mapping

There are also many text-based and XML types that you can map with LINQ to SQL. The following table shows the CLR types that O/R Designer and SQLMetal select when building an object model or external mapping file based on your database.

SQL SERVER TYPE	DEFAULT CLR TYPE MAPPING USED BY O/R DESIGNER AND SQLMETAL
`CHAR`	System.String
`NCHAR`	System.String
`VARCHAR`	System.String
`NVARCHAR`	System.String

SQL SERVER TYPE	DEFAULT CLR TYPE MAPPING USED BY O/R DESIGNER AND SQLMETAL
`TEXT`	System.String
`NTEXT`	System.String
`XML`	System.Xml.Linq.XElement

The next table shows the default type mappings used by the System.Data.Linq.DataContext.CreateDatabase method to define which type of SQL columns are created to map to the CLR types defined in your object model or external mapping file.

CLR TYPE	DEFAULT SQL SERVER TYPE USED BY SYSTEM.DATA.LINQ.DATACONTEXT.CREATEDATABASE
System.Char	`NCHAR(1)`
System.String	`NVARCHAR(4000)`
System.Char[]	`NVARCHAR(4000)`
Custom type implementing `Parse()` and `ToString()`	`NVARCHAR(MAX)`

There are many other text-based and XML mappings you can choose, but some may result in overflow or data loss exceptions while translating to or from the database. For more information, see the Type Mapping Run Time Behavior Matrix.

XML Types

The SQL Server `XML` data type is available starting in Microsoft SQL Server 2005. You can map the SQL Server `XML` data type to XElement, XDocument, or String. If the column stores XML fragments that cannot be read into XElement, the column must be mapped to String to avoid run-time errors. XML fragments that must be mapped to String include the following:

- A sequence of XML elements
- Attributes
- Public Identifiers (PI)
- Comments

Although you can map XElement and XDocument to SQL Server as shown in the Type Mapping Run Time Behavior Matrix, the System.Data.Linq.DataContext.CreateDatabase method has no default SQL Server type mapping for these types.

Custom Types

If a class implements `Parse()` and `ToString()`, you can map the object to any SQL text type (`CHAR`, `NCHAR`, `VARCHAR`, `NVARCHAR`, `TEXT`, `NTEXT`, `XML`). The object is stored in the database by sending the value returned by `ToString()` to the mapped database column. The object is reconstructed by invoking `Parse()` on the string returned by the database.

NOTE

LINQ to SQL does not support serialization by using System.Xml.Serialization.IXmlSerializable.

Date and Time Mapping

With LINQ to SQL, you can map many SQL Server date and time types. The following table shows the CLR types that O/R Designer and SQLMetal select when building an object model or external mapping file based on your database.

SQL SERVER TYPE	DEFAULT CLR TYPE MAPPING USED BY O/R DESIGNER AND SQLMETAL
`SMALLDATETIME`	System.DateTime
`DATETIME`	System.DateTime
`DATETIME2`	System.DateTime
`DATETIMEOFFSET`	System.DateTimeOffset
`DATE`	System.DateTime
`TIME`	System.TimeSpan

The next table shows the default type mappings used by the System.Data.Linq.DataContext.CreateDatabase method to define which type of SQL columns are created to map to the CLR types defined in your object model or external mapping file.

CLR TYPE	DEFAULT SQL SERVER TYPE USED BY SYSTEM.DATA.LINQ.DATACONTEXT.CREATEDATABASE
System.DateTime	`DATETIME`
System.DateTimeOffset	`DATETIMEOFFSET`
System.TimeSpan	`TIME`

There are many other date and time mappings you can choose, but some may result in overflow or data loss exceptions while translating to or from the database. For more information, see the Type Mapping Run Time Behavior Matrix.

NOTE

The SQL Server types `DATETIME2`, `DATETIMEOFFSET`, `DATE`, and `TIME` are available starting with Microsoft SQL Server 2008. LINQ to SQL supports mapping to these new types starting with the .NET Framework version 3.5 SP1.

System.Datetime

The range and precision of the CLR System.DateTime type is greater than the range and precision of the SQL Server `DATETIME` type, which is the default type mapping for the System.Data.Linq.DataContext.CreateDatabase method. To help avoid exceptions related to dates outside the range of `DATETIME`, use `DATETIME2`, which is available starting with Microsoft SQL Server 2008. `DATETIME2` can match the range and precision of the CLR

System.DateTime.

SQL Server dates have no concept of TimeZone, a feature that is richly supported in the CLR. TimeZone values are saved as is to the database without TimeZone conversion, regardless of the original DateTimeKind information. When DateTime values are retrieved from the database, their value is loaded as is into a DateTime with a DateTimeKind of Unspecified. For more information about supported System.DateTime methods, see System.DateTime Methods.

System.TimeSpan

Microsoft SQL Server 2008 and the .NET Framework 3.5 SP1 let you map the CLR System.TimeSpan type to the SQL Server `TIME` type. However, there is a large difference between the range that the CLR System.TimeSpan supports and what the SQL Server `TIME` type supports. Mapping values less than 0 or greater than 23:59:59.9999999 hours to the SQL `TIME` will result in overflow exceptions. For more information, see System.TimeSpan Methods.

In Microsoft SQL Server 2000 and SQL Server 2005, you cannot map database fields to TimeSpan. However, operations on TimeSpan are supported because TimeSpan values can be returned from DateTime subtraction or introduced into an expression as a literal or bound variable.

Binary Mapping

There are many SQL Server types that can map to the CLR type System.Data.Linq.Binary. The following table shows the SQL Server types that cause O/R Designer and SQLMetal to define a CLR System.Data.Linq.Binary type when building an object model or external mapping file based on your database.

SQL SERVER TYPE	DEFAULT CLR TYPE MAPPING USED BY O/R DESIGNER AND SQLMETAL
`BINARY(50)`	System.Data.Linq.Binary
`VARBINARY(50)`	System.Data.Linq.Binary
`VARBINARY(MAX)`	System.Data.Linq.Binary
`VARBINARY(MAX)` with the `FILESTREAM` attribute	System.Data.Linq.Binary
`IMAGE`	System.Data.Linq.Binary
`TIMESTAMP`	System.Data.Linq.Binary

The next table shows the default type mappings used by the System.Data.Linq.DataContext.CreateDatabase method to define which type of SQL columns are created to map to the CLR types defined in your object model or external mapping file.

CLR TYPE	DEFAULT SQL SERVER TYPE USED BY SYSTEM.DATA.LINQ.DATACONTEXT.CREATEDATABASE
System.Data.Linq.Binary	`VARBINARY(MAX)`
System.Byte	`VARBINARY(MAX)`
System.Runtime.Serialization.ISerializable	`VARBINARY(MAX)`

There are many other binary mappings you can choose, but some may result in overflow or data loss exceptions while translating to or from the database. For more information, see the Type Mapping Run Time Behavior Matrix.

SQL Server FILESTREAM

The `FILESTREAM` attribute for `VARBINARY(MAX)` columns is available starting with Microsoft SQL Server 2008; you can map to it with LINQ to SQL starting with the .NET Framework version 3.5 SP1.

Although you can map `VARBINARY(MAX)` columns with the `FILESTREAM` attribute to Binary objects, the System.Data.Linq.DataContext.CreateDatabase method is unable to automatically create columns with the `FILESTREAM` attribute. For more information about `FILESTREAM`, see FILESTREAM Overview on Microsoft SQL Server Books Online.

Binary Serialization

If a class implements the ISerializable interface, you can serialize an object to any SQL binary field (`BINARY`, `VARBINARY`, `IMAGE`). The object is serialized and deserialized according to how the ISerializable interface is implemented. For more information, see Binary Serialization.

Miscellaneous Mapping

The following table shows the default type mappings for some miscellaneous types that have not yet been mentioned. The following table shows the CLR types that O/R Designer and SQLMetal select when building an object model or external mapping file based on your database.

SQL SERVER TYPE	DEFAULT CLR TYPE MAPPING USED BY O/R DESIGNER AND SQLMETAL
`UNIQUEIDENTIFIER`	System.Guid
`SQL_VARIANT`	System.Object

The next table shows the default type mappings used by the System.Data.Linq.DataContext.CreateDatabase method to define which type of SQL columns are created to map to the CLR types defined in your object model or external mapping file.

CLR TYPE	DEFAULT SQL SERVER TYPE USED BY SYSTEM.DATA.LINQ.DATACONTEXT.CREATEDATABASE
System.Guid	`UNIQUEIDENTIFIER`
System.Object	`SQL_VARIANT`

LINQ to SQL does not support any other type mappings for these miscellaneous types. For more information, see the Type Mapping Run Time Behavior Matrix.

See Also

Attribute-Based Mapping
External Mapping
Data Types and Functions
SQL-CLR Type Mismatches

Basic Data Types

5/1/2017 • 1 min to read • Edit Online

Because LINQ to SQL queries translate to Transact-SQL before they are executed on the Microsoft SQL Server. LINQ to SQL supports much of the same built-in functionality that SQL Server does for basic data types.

Casting

Implicit or explicit casts are enabled from a source CLR type to a target CLR type if there is a similar valid conversion within SQL Server. For more information about CLR casting, see CType Function (Visual Basic) and as. After conversion, casts change the behavior of operations performed on a CLR expression to match the behavior of other CLR expressions that naturally map to the destination type. Casts are also translatable in the context of inheritance mapping. Objects can be cast to more specific entity subtypes so that their subtype-specific data can be accessed.

Equality Operators

LINQ to SQL supports the following equality operators on basic data types inside LINQ to SQL queries:

- Equal and Inequality Operator: Equality and inequality operators are supported for numeric Boolean, DateTime, and TimeSpan types. For more about Visual Basic operators `=` and `<>`, see Comparison Operators. For more information about C# comparison operators `==` and `!=`, see == Operator and != Operator, respectively
- Is operator: The `IS` operator has a supported translation when inheritance mapping is being used. It can be used instead of directly testing the discriminator column to determine whether an object is of a specific entity type, and is translated to a check on the discriminator column. For more information about the Visual Basic and C# Is operators, see Is Operator and is.

See Also

SQL-CLR Type Mapping
Data Types and Functions

Boolean Data Types

5/1/2017 • 1 min to read • Edit Online

Boolean operators work as expected in the common language runtime (CLR), except that short-circuiting behavior is not translated. For example, the Visual Basic `AndAlso` operator behaves like the `And` operator. The C# `&&` operator behaves like the `&` operator.

LINQ to SQL supports the following operators.

VISUAL BASIC	C#
And Operator	& Operator
AndAlso Operator	&& Operator
Or Operator	| Operator
OrElse Operator	|| Operator
Xor Operator	^ Operator
Not Operator	! Operator

See Also

Data Types and Functions

Null Semantics

5/1/2017 • 1 min to read • Edit Online

The following table provides links to various parts of the LINQ to SQL documentation where `null` (`Nothing` in Visual Basic) issues are discussed.

TOPIC	DESCRIPTION
SQL-CLR Type Mismatches	The "Null Semantics" section of this topic includes discussion of the three-state SQL Boolean versus the two-state common language runtime (CLR) Boolean, the literal `Nothing` (Visual Basic) and `null` (C#), and other similar issues.
Standard Query Operator Translation	The "Null Semantics" section of this topic describes null comparison semantics in LINQ to SQL.
System.String Methods	The "Differences from .NET" section of this topic describes how a return of 0 from LastIndexOf might mean either that the string is null or that the found position is 0.
Compute the Sum of Values in a Numeric Sequence	Describes how the Sum operator evaluates to `null` (`Nothing` in Visual Basic) instead of 0 for a sequence that contains only nulls or for an empty sequence.

See Also

Data Types and Functions

Numeric and Comparison Operators

5/10/2017 • 1 min to read • Edit Online

Arithmetic and comparison operators work as expected in the common language runtime (CLR) except as follows:

- SQL does not support the modulus operator on floating-point numbers.
- SQL does not support unchecked arithmetic.
- Increment and decrement operators cause side-effects when you use them in expressions that cannot be replicated in SQL and are, therefore, not supported.

Supported Operators

LINQ to SQL supports the following operators.

- Basic arithmetic operators:
  - `+`
  - `-` (subtraction)
  - `*`
  - `/`
  - Visual Basic integer division (`\`)
  - `%` (Visual Basic `Mod`)
  - `<<`
  - `>>`
  - `-` (unary negation)
- Basic comparison operators:
  - Visual Basic `=` and C# `==`
  - Visual Basic `<>` and C# `!=`
  - Visual Basic `Is/IsNot`
  - `<`
  - `<=`
  - `>`
  - `>=`

See Also

Data Types and Functions
C# Operators
Operators

Sequence Operators

5/1/2017 • 1 min to read • Edit Online

Generally speaking, LINQ to SQL does not support sequence operators that have one or more of the following qualities:

- Take a lambda with an index parameter.
- Rely on the properties of sequential rows, such as TakeWhile.
- Rely on an arbitrary CLR implementation, such as IComparer<T>.

EXAMPLES OF UNSUPPORTED
System.Linq.Enumerable.Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)
System.Linq.Enumerable.Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)
System.Linq.Enumerable.Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)
System.Linq.Enumerable.TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)
System.Linq.Enumerable.TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)
System.Linq.Enumerable.SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)
System.Linq.Enumerable.SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)
System.Linq.Enumerable.GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)
System.Linq.Enumerable.GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>, IEqualityComparer<TKey>)
System.Linq.Enumerable.Reverse<TSource>(IEnumerable<TSource>)
System.Linq.Enumerable.DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)
System.Linq.Enumerable.ElementAt<TSource>(IEnumerable<TSource>, Int32)
System.Linq.Enumerable.ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)
System.Linq.Enumerable.Range(Int32, Int32)
System.Linq.Enumerable.Repeat<TResult>(TResult, Int32)
System.Linq.Enumerable.Empty<TResult>()
System.Linq.Enumerable.Contains<TSource>(IEnumerable<TSource>, TSource)

EXAMPLES OF UNSUPPORTED
System.Linq.Enumerable.Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)
System.Linq.Enumerable.Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)
System.Linq.Enumerable.Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)
System.Linq.Enumerable.SequenceEqual

Differences from .NET

All supported sequence operators work as expected in the common language runtime (CLR) except for `Average`. `Average` returns a value of the same type as the type being averaged, whereas in the CLR `Average` always returns either a Double or a Decimal. If the source argument is explicitly cast to double / decimal or the selector casts to double / decimal, the resulting SQL will also have such a conversion and the result will be as expected.

See Also

Data Types and Functions

System.Convert Methods

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL does not support the following Convert methods.

- Versions with an IFormatProvider parameter.
- Methods that involve char arrays or byte arrays:
  - FromBase64CharArray
  - ToBase64CharArray
  - FromBase64String
  - ToBase64String
- The following methods:
  - `public static <Type2> To<Type2>(<Type1> value);` where

    `Type1` and `Type2` are each one of `sbyte`, `uint`, `ulong`, or `ushort`.
  - C#:

    `int To<int type>(string value, int fromBase),`

    `ToString(... value, int toBase)`
  - Visual Basic:

    `Function To(Of [Numeric])(value as String, fromBase As Integer)`

    `As [Numeric], ToString( value As …, toBase As Integer)`
  - IsDBNull
  - GetTypeCode
  - ChangeType

See Also

Data Types and Functions

System.DateTime Methods

5/1/2017 • 1 min to read • Edit Online

The following LINQ to SQL-supported methods, operators, and properties are available to use in LINQ to SQL queries. When a method, operator or property is unsupported, LINQ to SQL cannot translate the member for execution on the SQL Server. You may use these members in your code, however, they must be evaluated before the query is translated to Transact-SQL or after the results have been retrieved from the database.

Supported System.DateTime Members

Once mapped in the object model or external mapping file, LINQ to SQL allows you to call the following System.DateTime members inside LINQ to SQL queries.

SUPPORTED DATETIME METHODS	SUPPORTED DATETIME OPERATORS	SUPPORTED DATETIME PROPERTIES
Add	Addition	Date
AddDays	Equality	Day
AddHours	GreaterThan	DayOfWeek
AddMilliseconds	GreaterThanOrEqual	DayOfYear
AddMinutes	Inequality	Hour
AddMonths	LessThan	Millisecond
AddSeconds	LessThanOrEqual	Minute
AddTicks	Subtraction	Month
AddYears		Now
Compare		Second
CompareTo(DateTime)		TimeOfDay
Equals(DateTime)		Today
		Year

Members Not Supported by LINQ to SQL

The following members are not supported inside LINQ to SQL queries.

IsDaylightSavingTime	IsLeapYear

DaysInMonth	ToBinary
ToFileTime	ToFileTimeUtc
ToLongDateString	ToLongTimeString
ToOADate	ToShortDateString
ToShortTimeString	ToUniversalTime
FromBinary	UtcNow
FromFileTime	FromFileTimeUtc
FromOADate	GetDateTimeFormats

Method Translation Example

All methods supported by LINQ to SQL are translated to Transact-SQL before they are sent to SQL Server. For example, consider the following pattern.

```
(dateTime1 – dateTime2).{Days, Hours, Milliseconds, Minutes, Months, Seconds, Years}
```

When it is recognized, it is translated into a direct call to the SQL Server `DATEDIFF` function, as follows:

```
DATEDIFF({DatePart}, @dateTime1, @dateTime2)
```

SQLMethods Date and Time Methods

In addition to the methods offered by the DateTime structure, LINQ to SQL offers the methods listed in the following table from the System.Data.Linq.SqlClient.SqlMethods class for working with date and time.

DateDiffDay	DateDiffMillisecond	DateDiffNanosecond
DateDiffHour	DateDiffMinute	DateDiffSecond
DateDiffMicrosecond	DateDiffMonth	DateDiffYear

See Also

Query Concepts
Creating the Object Model
SQL-CLR Type Mapping
Data Types and Functions

System.Math Methods

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL does not support the following Math methods.

- System.Math.DivRem(Int32, Int32, Int32)
- System.Math.DivRem(Int64, Int64, Int64)
- System.Math.IEEERemainder(Double, Double)

Differences from .NET

The .NET Framework has different rounding semantics from SQL Server. The Round method in the .NET Framework performs *Banker's rounding*, where numbers that ends in .5 round to the nearest even digit instead of to the next higher digit. For example, 2.5 rounds to 2, while 3.5 rounds to 4. (This technique helps avoid systematic bias toward higher values in large data transactions.)

In SQL, the `ROUND` function instead always rounds away from 0. Therefore 2.5 rounds to 3, contrasted with its rounding to 2 in the .NET Framework.

LINQ to SQL passes through to the SQL `ROUND` semantics and does not try to implement Banker's rounding.

See Also

Data Types and Functions

System.Object Methods

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL supports the following Object methods.

System.Object.Equals(Object)	System.Object.Equals(Object, Object)
System.Object.ToString()	

LINQ to SQL does not support the following Object methods.

System.Object.GetHashCode()	System.Object.ReferenceEquals(Object, Object)
System.Object.MemberwiseClone()	System.Object.GetType()
System.Object.ToString() for binary types such as `BINARY`, `VARBINARY`, `IMAGE`, and `TIMESTAMP`.	

Differences from .NET

The output of System.Object.ToString() for double uses SQL `CONVERT`(NVARCHAR(30), @x, 2) on SQL. SQL always uses 16 digits and scientific notation in this case (for example, "0.000000000000000e+000" for 0). As a result, System.Object.ToString() conversion does not produce the same string as System.Convert.ToString in the .NET Framework.

See Also

Data Types and Functions

System.String Methods

5/1/2017 • 1 min to read • Edit Online

LINQ to SQL does not support the following String methods.

Unsupported System.String Methods in General

Unsupported String methods in general:

- Culture-aware overloads (methods that take a `CultureInfo` / `StringComparison` / `IFormatProvider`).
- Methods that take or produce a `char` array.

Unsupported System.String Static Methods

UNSUPPORTED SYSTEM.STRING STATIC METHODS
System.String.Copy(String)
System.String.Compare(String, String, Boolean)
System.String.Compare(String, String, Boolean, CultureInfo)
System.String.Compare(String, Int32, String, Int32, Int32)
System.String.Compare(String, Int32, String, Int32, Int32, Boolean)
System.String.Compare(String, Int32, String, Int32, Int32, Boolean, CultureInfo)
System.String.CompareOrdinal(String, String)
System.String.CompareOrdinal(String, Int32, String, Int32, Int32)
System.String.Format
System.String.Join

Unsupported System.String Non-static Methods

UNSUPPORTED SYSTEM.STRING NON-STATIC METHODS
System.String.IndexOfAny(Char[])
System.String.Split
System.String.ToCharArray()
System.String.ToUpper(CultureInfo)

UNSUPPORTED SYSTEM.STRING NON-STATIC METHODS
System.String.TrimEnd(Char[])
System.String.TrimStart(Char[])

Differences from .NET

- Queries do not account for SQL Server collations that might be in effect on the server, and therefore will provide culture-sensitive, case-insensitive comparisons by default. This behavior differs from the default, case-sensitive semantics of the .NET Framework.
- When `LastIndexOf` returns 0, either the string is `NULL` or the found position is 0.
- Unexpected results might be returned from concatenation or other operations on fixed-length strings (`CHAR` , `NCHAR`), because these types automatically have padding applied in the database.
- Because many methods, such as `Replace` , `ToLower` , `ToUpper` , and the character indexer, have no valid translation for `TEXT` or `NTEXT` columns and XML, `SqlExceptions` occur if translated normally. This behavior is considered acceptable for these types. However, all string operations must match common language runtime (CLR) semantics for `VARCHAR` , `NVARCHAR` , `VARCHAR(max)` , and `NVARCHAR(max)` .

See Also

Data Types and Functions

System.TimeSpan Methods

5/1/2017 • 1 min to read • Edit Online

Member support for System.TimeSpan greatly depends on the versions of the .NET Framework and Microsoft SQL Server that you are using.

When a method, operator, or property is unsupported; it means that LINQ to SQL cannot translate the member for execution on the SQL Server. You may still be able to use these members in your code. However, they must be evaluated before the query is translated to Transact-SQL or after the results have been retrieved from the database.

Previous Limitations

When using LINQ to SQL with versions of the .NET Framework prior to .NET Framework 3.5 SP1, you cannot map SQL Server database fields to System.TimeSpan. However, operations on TimeSpan are supported because TimeSpan values can be returned from DateTime subtraction or introduced into an expression as a literal or bound variable.

Supported System.TimeSpan Method Support

The following LINQ to SQL-supported methods, operators, and properties are available for you to use in your LINQ to SQL queries. Once mapped in the object model or external mapping file, LINQ to SQL allows you to call many of the System.TimeSpan members inside your LINQ to SQL queries.

SUPPORTED TIMESPAN METHODS	SUPPORTED TIMESPAN OPERATORS	SUPPORTED TIMESPAN PROPERTIES
Compare	Equality	Days
CompareTo(TimeSpan)	GreaterThan	Hours
Duration	GreaterThanOrEqual	MaxValue
Equals(TimeSpan, TimeSpan)	Inequality	Milliseconds
Equals(TimeSpan)	LessThan	Minutes
	LessThanOrEqual	

NOTE

The ability to map System.TimeSpan to a SQL `TIME` column with LINQ to SQL requires the .NET Framework 3.5 SP1 and beyond. The SQL `TIME` data type is only available in Microsoft SQL Server 2008 and beyond.

Addition and Subtraction

Although the CLR System.TimeSpan type does support addition and subtraction, the SQL `TIME` type does not. Because of this, your LINQ to SQL queries will generate errors if they attempt addition and subtraction when they are mapped to the SQL `TIME` type. You can find other considerations for working with SQL date and time types in SQL-CLR Type Mapping.

See Also

Query Concepts

Creating the Object Model

SQL-CLR Type Mapping

Data Types and Functions

Unsupported Functionality

5/1/2017 • 1 min to read • Edit Online

In LINQ to SQL, the following SQL functionality cannot be exposed through translation of existing common language runtime (CLR) and .NET Framework constructs:

- `STDDEV`
- `LIKE`

  Although `LIKE` is not supported through direct translation, similar functionality exists in the SqlMethods class. For more information, see System.Data.Linq.SqlClient.SqlMethods.Like.
- `DATEDIFF`

  LINQ to SQL has limited support for `DATEDIFF`. Similar functionality exists in the SqlMethods class.
- `ROUND`

  LINQ to SQL has limited support for `ROUND`. For more information, see System.Math Methods.

See Also

Data Types and Functions

System.DateTimeOffset Methods

5/1/2017 • 1 min to read • Edit Online

Once mapped in the object model or external mapping file, LINQ to SQL allows you to call most of the System.DateTimeOffset methods, operators, and properties from within your LINQ to SQL queries.

The only methods not supported are those inherited from System.Object that do not make sense in the context of LINQ to SQL queries, such as: `Finalize`, `GetHashCode`, `GetType`, and `MemberwiseClone`. These methods are not supported because LINQ to SQL cannot translate them for execution on the SQL Server.

NOTE

The common language runtime (CLR) System.DateTimeOffset structure, and the ability to map it to a SQL `DATETIMEOFFSET` column with LINQ to SQL, requires the .NET Framework 3.5 SP1 or beyond. The SQL `DATETIMEOFFSET` column is only available in Microsoft SQL Server 2008 and beyond.

SQLMethods Date and Time Methods

In addition to the methods offered by the DateTimeOffset structure, LINQ to SQL offers the methods listed in the following table from the System.Data.Linq.SqlClient.SqlMethods class for working with date and time.

DateDiffDay	DateDiffMillisecond	DateDiffNanosecond
DateDiffHour	DateDiffMinute	DateDiffSecond
DateDiffMicrosecond	DateDiffMonth	DateDiffYear

See Also

Query Concepts
Creating the Object Model
SQL-CLR Type Mapping

Attribute-Based Mapping

5/1/2017 • 5 min to read • Edit Online

LINQ to SQL maps a SQL Server database to a LINQ to SQL object model by either applying attributes or by using an external mapping file. This topic outlines the attribute-based approach.

In its most elementary form, LINQ to SQL maps a database to a DataContext, a table to a class, and columns and relationships to properties on those classes. You can also use attributes to map an inheritance hierarchy in your object model. For more information, see How to: Generate the Object Model in Visual Basic or C#.

Developers using Visual Studio typically perform attribute-based mapping by using the Object Relational Designer. You can also use the SQLMetal command-line tool, or you can hand-code the attributes yourself. For more information, see How to: Generate the Object Model in Visual Basic or C#.

NOTE

You can also map by using an external XML file. For more information, see External Mapping.

The following sections describe attribute-based mapping in more detail. For more information, see the System.Data.Linq.Mapping namespace.

DatabaseAttribute Attribute

Use this attribute to specify the default name of the database when a name is not supplied by the connection. This attribute is optional, but if you use it, you must apply the Name property, as described in the following table.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
Name	String	See Name	Used with its Name property, specifies the name of the database.

For more information, see DatabaseAttribute.

TableAttribute Attribute

Use this attribute to designate a class as an entity class that is associated with a database table or view. LINQ to SQL treats classes that have this attribute as persistent classes. The following table describes the Name property.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
Name	String	Same string as class name	Designates a class as an entity class associated with a database table.

For more information, see TableAttribute.

ColumnAttribute Attribute

Use this attribute to designate a member of an entity class to represent a column in a database table. You can apply this attribute to any field or property.

Only those members you identify as columns are retrieved and persisted when LINQ to SQL saves changes to the database. Members without this attribute are assumed to be non-persistent and are not submitted for inserts or updates.

The following table describes properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
AutoSync	AutoSync	Never	Instructs the common language runtime (CLR) to retrieve the value after an insert or update operation. Options: Always, Never, OnUpdate, OnInsert.
CanBeNull	Boolean	`true`	Indicates that a column can contain null values.
DbType	String	Inferred database column type	Uses database types and modifiers to specify the type of the database column.
Expression	String	Empty	Defines a computed column in a database.
IsDbGenerated	Boolean	`false`	Indicates that a column contains values that the database auto-generates.
IsDiscriminator	Boolean	`false`	Indicates that the column contains a discriminator value for a LINQ to SQL inheritance hierarchy.
IsPrimaryKey	Boolean	`false`	Specifies that this class member represents a column that is or is part of the primary keys of the table.
IsVersion	Boolean	`false`	Identifies the column type of the member as a database timestamp or version number.
UpdateCheck	UpdateCheck	`Always`, unless IsVersion is `true` for a member	Specifies how LINQ to SQL approaches the detection of optimistic concurrency conflicts.

For more information, see ColumnAttribute.

NOTE

AssociationAttribute and ColumnAttribute Storage property values are case sensitive. For example, ensure that values used in the attribute for the AssociationAttribute.Storage property match the case for the corresponding property names used elsewhere in the code. This applies to all .NET programming languages, even those which are not typically case sensitive, including Visual Basic. For more information about the Storage property, see System.Data.Linq.Mapping.DataAttribute.Storage.

AssociationAttribute Attribute

Use this attribute to designate a property to represent an association in the database, such as a foreign key to primary key relationship. For more information about relationships, see How to: Map Database Relationships.

The following table describes properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
DeleteOnNull	Boolean	`false`	When placed on an association whose foreign key members are all non-nullable, deletes the object when the association is set to null.
DeleteRule	String	None	Adds delete behavior to an association.
IsForeignKey	Boolean	`false`	If true, designates the member as the foreign key in an association representing a database relationship.
IsUnique	Boolean	`false`	If true, indicates a uniqueness constraint on the foreign key.
OtherKey	String	ID of the related class	Designates one or more members of the target entity class as key values on the other side of the association.
ThisKey	String	ID of the containing class	Designates members of this entity class to represent the key values on this side of the association.

For more information, see AssociationAttribute.

NOTE
AssociationAttribute and ColumnAttribute Storage property values are case sensitive. For example, ensure that values used in the attribute for the AssociationAttribute.Storage property match the case for the corresponding property names used elsewhere in the code. This applies to all .NET programming languages, even those which are not typically case sensitive, including Visual Basic. For more information about the Storage property, see System.Data.Linq.Mapping.DataAttribute.Storage.

InheritanceMappingAttribute Attribute

Use this attribute to map an inheritance hierarchy.

The following table describes properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
Code	String	None. Value must be supplied.	Specifies the code value of the discriminator.
IsDefault	Boolean	`false`	If true, instantiates an object of this type when no discriminator value in the store matches any one of the specified values.
Type	Type	None. Value must be supplied.	Specifies the type of the class in the hierarchy.

For more information, see InheritanceMappingAttribute.

FunctionAttribute Attribute

Use this attribute to designate a method as representing a stored procedure or user-defined function in the database.

The following table describes the properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
IsComposable	Boolean	`false`	If false, indicates mapping to a stored procedure. If true, indicates mapping to a user-defined function.
Name	String	Same string as name in the database	Specifies the name of the stored procedure or user-defined function.

For more information, see FunctionAttribute.

ParameterAttribute Attribute

Use this attribute to map input parameters on stored procedure methods.

The following table describes properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
DbType	String	None	Specifies database type.
Name	String	Same string as parameter name in database	Specifies a name for the parameter.

For more information, see ParameterAttribute.

ResultTypeAttribute Attribute

Use this attribute to specify a result type.

The following table describes properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
Type	Type	(None)	Used on methods mapped to stored procedures that return IMultipleResults. Declares the valid or expected type mappings for the stored procedure.

For more information, see ResultTypeAttribute.

DataAttribute Attribute

Use this attribute to specify names and private storage fields.

The following table describes properties of this attribute.

PROPERTY	TYPE	DEFAULT	DESCRIPTION
Name	String	Same as name in database	Specifies the name of the table, column, and so on.
Storage	String	Public accessors	Specifies the name of the underlying storage field.

For more information, see DataAttribute.

See Also

Reference

Code Generation in LINQ to SQL

6/2/2017 • 5 min to read • Edit Online

You can generate code to represent a database by using either the Object Relational Designer or the SQLMetal command-line tool. In either case, end-to-end code generation occurs in three stages:

1. The *DBML Extractor* extracts schema information from the database and reassembles the information into an XML-formatted DBML file.

2. The DBML file is scanned by the *DBML Validator* for errors.

3. If no validation errors appear, the file is passed to the Code Generator.

For more information, see SqlMetal.exe (Code Generation Tool). Developers using Visual Studio can also use the Object Relational Designer to generate code. See LINQ to SQL Tools in Visual Studio.

DBML Extractor

The DBML Extractor is a LINQ to SQL component that takes database metadata as input and produces a DBML file as output.

Code Generator

The Code Generator is a LINQ to SQL component that translates DBML files to Visual Basic, C#, or XML mapping files.

XML Schema Definition File

The DBML file must be valid against the following schema definition as an XSD file.

Distinguish this schema definition file from the schema definition file that is used to validate an external mapping file. For more information, see External Mapping).

NOTE

Visual Studio users will also find this XSD file in the XML Schemas dialog box as "DbmlSchema.xsd". To use the XSD file correctly for validating a DBML file, see How to: Validate DBML and External Mapping Files.

```
 ?<?xml version="1.0" encoding="utf-16"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://schemas.microsoft.com/linqtosql/dbml/2007"
xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007"
elementFormDefault="qualified" >
  <xs:element name="Database" type="Database" />
  <xs:complexType name="Database">
    <xs:sequence>
      <xs:element name="Connection" type="Connection" minOccurs="0" maxOccurs="1" />
      <xs:element name="Table" type="Table" minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="Function" type="Function" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="EntityNamespace" type="xs:string" use="optional" />
    <xs:attribute name="ContextNamespace" type="xs:string" use="optional" />
    <xs:attribute name="Class" type="xs:string" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="Modifier" type="ClassModifier" use="optional" />
    <xs:attribute name="BaseType" type="xs:string" use="optional" />
    <xs:attribute name="Provider" type="xs:string" use="optional" />
    <xs:attribute name="ExternalMapping" type="xs:boolean" use="optional" />
    <xs:attribute name="Serialization" type="SerializationMode" use="optional" />
    <xs:attribute name="EntityBase" type="xs:string" use="optional" />
  </xs:complexType>
  <xs:complexType name="Table">
    <xs:all>
      <xs:element name="Type" type="Type" minOccurs="1" maxOccurs="1" />
      <xs:element name="InsertFunction" type="TableFunction" minOccurs="0" maxOccurs="1" />
      <xs:element name="UpdateFunction" type="TableFunction" minOccurs="0" maxOccurs="1" />
      <xs:element name="DeleteFunction" type="TableFunction" minOccurs="0" maxOccurs="1" />
    </xs:all>
    <xs:attribute name="Name" type="xs:string" use="required" />
    <xs:attribute name="Member" type="xs:string" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="Modifier" type="MemberModifier" use="optional" />
  </xs:complexType>
  <xs:complexType name="Type">
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Column" type="Column" minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="Association" type="Association" minOccurs="0" maxOccurs="unbounded" />
      </xs:choice>
      <xs:element name="Type" type="Type" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="IdRef" type="xs:IDREF" use="optional" />
    <xs:attribute name="Id" type="xs:ID" use="optional" />
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="InheritanceCode" type="xs:string" use="optional" />
    <xs:attribute name="IsInheritanceDefault" type="xs:boolean" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="Modifier" type="ClassModifier" use="optional" />
  </xs:complexType>
  <xs:complexType name="Column">
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Member" type="xs:string" use="optional" />
    <xs:attribute name="Storage" type="xs:string" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="Modifier" type="MemberModifier" use="optional" />
    <xs:attribute name="Type" type="xs:string" use="required" />
    <xs:attribute name="DbType" type="xs:string" use="optional" />
    <xs:attribute name="IsReadOnly" type="xs:boolean" use="optional" />
    <xs:attribute name="IsPrimaryKey" type="xs:boolean" use="optional" />
    <xs:attribute name="IsDbGenerated" type="xs:boolean" use="optional" />
    <xs:attribute name="CanBeNull" type="xs:boolean" use="optional" />
    <xs:attribute name="UpdateCheck" type="UpdateCheck" use="optional" />
    <xs:attribute name="IsDiscriminator" type="xs:boolean" use="optional" />
    <xs:attribute name="Expression" type="xs:string" use="optional" />
    <xs:attribute name="IsVersion" type="xs:boolean" use="optional" />
    <xs:attribute name="IsDelayLoaded" type="xs:boolean" use="optional" />
    <xs:attribute name="AutoSync" type="AutoSync" use="optional" />
  </xs:complexType>
  <xs:complexType name="Association">
    <xs:attribute name="Name" type="xs:string" use="required" />
    <xs:attribute name="Member" type="xs:string" use="required" />
    <xs:attribute name="Storage" type="xs:string" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="Modifier" type="MemberModifier" use="optional" />
    <xs:attribute name="Type" type="xs:string" use="required" />
    <xs:attribute name="ThisKey" type="xs:string" use="optional" />
    <xs:attribute name="OtherKey" type="xs:string" use="optional" />
    <xs:attribute name="IsForeignKey" type="xs:boolean" use="optional" />
    <xs:attribute name="Cardinality" type="Cardinality" use="optional" />
    <xs:attribute name="DeleteRule" type="xs:string" use="optional" />
    <xs:attribute name="DeleteOnNull" type="xs:boolean" use="optional" />
  </xs:complexType>
  <xs:complexType name="Function">
```

```xml
  <xs:complexType name="Function">
    <xs:sequence>
      <xs:element name="Parameter" type="Parameter" minOccurs="0" maxOccurs="unbounded" />
      <xs:choice>
        <xs:element name="ElementType" type="Type" minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="Return" type="Return" minOccurs="0" maxOccurs="1" />
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required" />
    <xs:attribute name="Id" type="xs:ID" use="optional" />
    <xs:attribute name="Method" type="xs:string" use="optional" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
    <xs:attribute name="Modifier" type="MemberModifier" use="optional" />
    <xs:attribute name="HasMultipleResults" type="xs:boolean" use="optional" />
    <xs:attribute name="IsComposable" type="xs:boolean" use="optional" />
  </xs:complexType>
  <xs:complexType name="TableFunction">
    <xs:sequence>
      <xs:element name="Argument" type="TableFunctionParameter" minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="Return" type="TableFunctionReturn" minOccurs="0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="FunctionId" type="xs:IDREF" use="required" />
    <xs:attribute name="AccessModifier" type="AccessModifier" use="optional" />
  </xs:complexType>
  <xs:complexType name="Parameter">
    <xs:attribute name="Name" type="xs:string" use="required" />
    <xs:attribute name="Parameter" type="xs:string" use="optional" />
    <xs:attribute name="Type" type="xs:string" use="required" />
    <xs:attribute name="DbType" type="xs:string" use="optional" />
    <xs:attribute name="Direction" type="ParameterDirection" use="optional" />
  </xs:complexType>
  <xs:complexType name="Return">
    <xs:attribute name="Type" type="xs:string" use="required" />
    <xs:attribute name="DbType" type="xs:string" use="optional" />
  </xs:complexType>
  <xs:complexType name="TableFunctionParameter">
    <xs:attribute name="Parameter" type="xs:string" use="required" />
    <xs:attribute name="Member" type="xs:string" use="required" />
    <xs:attribute name="Version" type="Version" use="optional" />
  </xs:complexType>
  <xs:complexType name="TableFunctionReturn">
    <xs:attribute name="Member" type="xs:string" use="required" />
  </xs:complexType>
  <xs:complexType name="Connection">
    <xs:attribute name="Provider" type="xs:string" use="required" />
    <xs:attribute name="Mode" type="ConnectionMode" use="optional" />
    <xs:attribute name="ConnectionString" type="xs:string" use="optional" />
    <xs:attribute name="SettingsObjectName" type="xs:string" use="optional" />
    <xs:attribute name="SettingsPropertyName" type="xs:string" use="optional" />
  </xs:complexType>
  <xs:simpleType name="ConnectionMode">
    <xs:restriction base="xs:string">
      <xs:enumeration value="ConnectionString" />
      <xs:enumeration value="AppSettings" />
      <xs:enumeration value="WebSettings" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="AccessModifier">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Public" />
      <xs:enumeration value="Internal" />
      <xs:enumeration value="Protected" />
      <xs:enumeration value="ProtectedInternal" />
      <xs:enumeration value="Private" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="UpdateCheck">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Always" />
```

```
        <xs:enumeration value="Never" />
        <xs:enumeration value="WhenChanged" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="SerializationMode">
      <xs:restriction base="xs:string">
        <xs:enumeration value="None" />
        <xs:enumeration value="Unidirectional" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ParameterDirection">
      <xs:restriction base="xs:string">
        <xs:enumeration value="In" />
        <xs:enumeration value="Out" />
        <xs:enumeration value="InOut" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="Version">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Current" />
        <xs:enumeration value="Original" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="AutoSync">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Never" />
        <xs:enumeration value="OnInsert" />
        <xs:enumeration value="OnUpdate" />
        <xs:enumeration value="Always" />
        <xs:enumeration value="Default" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ClassModifier">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Sealed" />
        <xs:enumeration value="Abstract" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="MemberModifier">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Virtual" />
        <xs:enumeration value="Override" />
        <xs:enumeration value="New" />
        <xs:enumeration value="NewVirtual" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="Cardinality">
      <xs:restriction base="xs:string">
        <xs:enumeration value="One" />
        <xs:enumeration value="Many" />
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

Sample DBML File

The following code is an excerpt from the DBML file created from the Northwind sample database. You can generate the whole file by using SQLMetal with the **/xml** option. For more information, see SqlMetal.exe (Code Generation Tool).

```
<?xml version="1.0" encoding="utf-16"?>
<Database Name="northwnd" Class="Northwnd" xmlns="http://schemas.microsoft.com/dsltools/DLinqML">

  <Table Name="Customers">
    <Type Name="Customer">
      <Column Name="CustomerID" Type="System.String" DbType="NChar(5) NOT NULL" IsPrimaryKey="True"
CanBeNull="False" />
      <Column Name="CompanyName" Type="System.String" DbType="NVarChar(40) NOT NULL" CanBeNull="False" />
      <Column Name="ContactName" Type="System.String" DbType="NVarChar(30)" CanBeNull="True" />
      <Column Name="ContactTitle" Type="System.String" DbType="NVarChar(30)" CanBeNull="True" />
      <Column Name="Address" Type="System.String" DbType="NVarChar(60)" CanBeNull="True" />
      <Column Name="City" Type="System.String" DbType="NVarChar(15)" CanBeNull="True" />
      <Column Name="Region" Type="System.String" DbType="NVarChar(15)" CanBeNull="True" />
      <Column Name="PostalCode" Type="System.String" DbType="NVarChar(10)" CanBeNull="True" />
      <Column Name="Country" Type="System.String" DbType="NVarChar(15)" CanBeNull="True" />
      <Column Name="Phone" Type="System.String" DbType="NVarChar(24)" CanBeNull="True" />
      <Column Name="Fax" Type="System.String" DbType="NVarChar(24)" CanBeNull="True" />
      <Association Name="FK_CustomerCustomerDemo_Customers" Member="CustomerCustomerDemos"
ThisKey="CustomerID" OtherKey="CustomerID" OtherTable="CustomerCustomerDemo" DeleteRule="NO ACTION" />
      <Association Name="FK_Orders_Customers" Member="Orders" ThisKey="CustomerID" OtherKey="CustomerID"
OtherTable="Orders" DeleteRule="NO ACTION" />
    </Type>
  </Table>
</Database>
```

See Also

Background Information

External Mapping

How to: Generate the Object Model as an External File

Downloading Sample Databases

Reference

External Mapping

5/1/2017 • 3 min to read • Edit Online

LINQ to SQL supports *external mapping*, a process by which you use a separate XML file to specify mapping between the data model of the database and your object model. Advantages of using an external mapping file include the following:

- You can keep your mapping code out of your application code. This approach reduces clutter in your application code.
- You can treat an external mapping file something like a configuration file. For example, you can update how your application behaves after shipping the binaries by just swapping out the external mapping file.

Requirements

The mapping file must be an XML file, and the file must validate against a LINQ to SQL schema definition (.xsd) file.

The following rules apply:

- The mapping file must be an XML file.
- The XML mapping file must be valid against the XML schema definition file. For more information, see How to: Validate DBML and External Mapping Files.
- External mapping overrides attribute-based mapping. In other words, when you use an external mapping source to create a DataContext, the DataContext ignores all mapping attributes you have created on classes. This behavior is true whether the class is included in the external mapping file.
- LINQ to SQL does not support the hybrid use of the two mapping approaches (attribute-based and external).

XML Schema Definition File

External mapping in LINQ to SQL must be valid against the following XML schema definition.

Distinguish this schema definition file from the schema definition file that is used to validate a DBML file. For more information, see Code Generation in LINQ to SQL).

NOTE

Visual Studio users will also find this XSD file in the XML Schemas dialog box as "LinqToSqlMapping.xsd". To use this file correctly for validating an external mapping file, see How to: Validate DBML and External Mapping Files.

```
?<?xml version="1.0" encoding="utf-16"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://schemas.microsoft.com/linqtosql/mapping/2007"
xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007"
elementFormDefault="qualified" >
  <xs:element name="Database" type="Database" />
  <xs:complexType name="Database">
    <xs:sequence>
      <xs:element name="Table" type="Table" minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="Function" type="Function" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Provider" type="xs:string" use="optional" />
  </xs:complexType>
  <xs:complexType name="Table">
    <xs:sequence>
      <xs:element name="Type" type="Type" minOccurs="1" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Member" type="xs:string" use="optional" />
  </xs:complexType>
  <xs:complexType name="Type">
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Column" type="Column" minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="Association" type="Association" minOccurs="0" maxOccurs="unbounded" />
      </xs:choice>
      <xs:element name="Type" type="Type" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required" />
    <xs:attribute name="InheritanceCode" type="xs:string" use="optional" />
    <xs:attribute name="IsInheritanceDefault" type="xs:boolean" use="optional" />
  </xs:complexType>
  <xs:complexType name="Column">
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Member" type="xs:string" use="required" />
    <xs:attribute name="Storage" type="xs:string" use="optional" />
    <xs:attribute name="DbType" type="xs:string" use="optional" />
    <xs:attribute name="IsPrimaryKey" type="xs:boolean" use="optional" />
    <xs:attribute name="IsDbGenerated" type="xs:boolean" use="optional" />
    <xs:attribute name="CanBeNull" type="xs:boolean" use="optional" />
    <xs:attribute name="UpdateCheck" type="UpdateCheck" use="optional" />
    <xs:attribute name="IsDiscriminator" type="xs:boolean" use="optional" />
    <xs:attribute name="Expression" type="xs:string" use="optional" />
    <xs:attribute name="IsVersion" type="xs:boolean" use="optional" />
    <xs:attribute name="AutoSync" type="AutoSync" use="optional" />
  </xs:complexType>
  <xs:complexType name="Association">
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Member" type="xs:string" use="required" />
    <xs:attribute name="Storage" type="xs:string" use="optional" />
    <xs:attribute name="ThisKey" type="xs:string" use="optional" />
    <xs:attribute name="OtherKey" type="xs:string" use="optional" />
    <xs:attribute name="IsForeignKey" type="xs:boolean" use="optional" />
    <xs:attribute name="IsUnique" type="xs:boolean" use="optional" />
    <xs:attribute name="DeleteRule" type="xs:string" use="optional" />
    <xs:attribute name="DeleteOnNull" type="xs:boolean" use="optional" />
  </xs:complexType>
  <xs:complexType name="Function">
    <xs:sequence>
      <xs:element name="Parameter" type="Parameter" minOccurs="0" maxOccurs="unbounded" />
      <xs:choice>
        <xs:element name="ElementType" type="Type" minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="Return" type="Return" minOccurs="0" maxOccurs="1" />
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Method" type="xs:string" use="required" />
    <xs:attribute name="IsComposable" type="xs:boolean" use="optional" />
  </xs:complexType>
  <xs:complexType name="Parameter">
    <xs:attribute name="Name" type="xs:string" use="optional" />
    <xs:attribute name="Parameter" type="xs:string" use="required" />
    <xs:attribute name="DbType" type="xs:string" use="optional" />
    <xs:attribute name="Direction" type="ParameterDirection" use="optional" />
  </xs:complexType>
  <xs:complexType name="Return">
    <xs:attribute name="DbType" type="xs:string" use="optional" />
  </xs:complexType>
```

```
    <xs:simpleType name="UpdateCheck">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Always" />
        <xs:enumeration value="Never" />
        <xs:enumeration value="WhenChanged" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ParameterDirection">
      <xs:restriction base="xs:string">
        <xs:enumeration value="In" />
        <xs:enumeration value="Out" />
        <xs:enumeration value="InOut" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="AutoSync">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Never" />
        <xs:enumeration value="OnInsert" />
        <xs:enumeration value="OnUpdate" />
        <xs:enumeration value="Always" />
        <xs:enumeration value="Default" />
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

See Also

Code Generation in LINQ to SQL
Reference
How to: Generate the Object Model as an External File

Frequently Asked Questions

5/31/2017 • 7 min to read • Edit Online

The following sections answer some common issues that you might encounter when you implement LINQ.

Additional issues are addressed in Troubleshooting.

Cannot Connect

Q. I cannot connect to my database.

A. Make sure your connection string is correct and that your SQL Server instance is running. Note also that LINQ to SQL requires the Named Pipes protocol to be enabled. For more information, see Learning by Walkthroughs.

Changes to Database Lost

Q. I made a change to data in the database, but when I reran my application, the change was no longer there.

A. Make sure that you call SubmitChanges to save results to the database.

Database Connection: Open How Long?

Q. How long does my database connection remain open?

A. A connection typically remains open until you consume the query results. If you expect to take time to process all the results and are not opposed to caching the results, apply ToList to the query. In common scenarios where each object is processed only one time, the streaming model is superior in both `DataReader` and LINQ to SQL.

The exact details of connection usage depend on the following:

- Connection status if the DataContext is constructed with a connection object.
- Connection string settings (for example, enabling Multiple Active Result Sets (MARS). For more information, see Multiple Active Result Sets (MARS).

Updating Without Querying

Q. Can I update table data without first querying the database?

A. Although LINQ to SQL does not have set-based update commands, you can use either of the following techniques to update without first querying:

- Use ExecuteCommand to send SQL code.
- Create a new instance of the object and initialize all the current values (fields) that affect the update. Then attach the object to the DataContext by using Attach and modify the field you want to change.

Unexpected Query Results

Q. My query is returning unexpected results. How can I inspect what is occurring?

A. LINQ to SQL provides several tools for inspecting the SQL code it generates. One of the most important is Log. For more information, see Debugging Support.

Unexpected Stored Procedure Results

Q. I have a stored procedure whose return value is calculated by `MAX()`. When I drag the stored procedure to the O/R Designer surface, the return value is not correct.

A. LINQ to SQL provides two ways to return database-generated values by way of stored procedures:

- By naming the output result.
- By explicitly specifying an output parameter.

The following is an example of incorrect output. Because LINQ to SQL cannot map the results, it always returns 0:

```
create procedure proc2

as

begin

select max(i) from t where name like 'hello'

end
```

The following is an example of correct output by using an output parameter:

```
create procedure proc2

@result int OUTPUT

as

select @result = MAX(i) from t where name like 'hello'

go
```

The following is an example of correct output by naming the output result:

```
create procedure proc2

as

begin

select nax(i) AS MaxResult from t where name like 'hello'

end
```

For more information, see Customizing Operations By Using Stored Procedures.

Serialization Errors

Q. When I try to serialize, I get the following error: "Type 'System.Data.Linq.ChangeTracker+StandardChangeTracker' ... is not marked as serializable."

A. Code generation in LINQ to SQL supports DataContractSerializer serialization. It does not support XmlSerializer or BinaryFormatter. For more information, see Serialization.

Multiple DBML Files

Q. When I have multiple DBML files that share some tables in common, I get a compiler error.

A. Set the **Context Namespace** and **Entity Namespace** properties from the Object Relational Designer to a distinct value for each DBML file. This approach eliminates the name/namespace collision.

Avoiding Explicit Setting of Database-Generated Values on Insert or Update

Q. I have a database table with a `DateCreated` column that defaults to SQL `Getdate()`. When I try to insert a new record by using LINQ to SQL, the value gets set to `NULL`. I would expect it to be set to the database default.

A. LINQ to SQL handles this situation automatically for identity (auto-increment) and rowguidcol (database-generated GUID) and timestamp columns. In other cases, you should manually set IsDbGenerated=`true` and AutoSync=Always/OnInsert/OnUpdate properties.

Multiple DataLoadOptions

Q. Can I specify additional load options without overwriting the first?

A. Yes. The first is not overwritten, as in the following example:

```
Dim dlo As New DataLoadOptions()
dlo.LoadWith(Of Order)(Function(o As Order) o.Customer)
dlo.LoadWith(Of Order)(Function(o As Order) o.OrderDetails)
```

```
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Order>(o => o.Customer);
dlo.LoadWith<Order>(o => o.OrderDetails);
```

Errors Using SQL Compact 3.5

Q. I get an error when I drag tables out of a SQL Server Compact 3.5 database.

A. The Object Relational Designer does not support SQL Server Compact 3.5, although the LINQ to SQL runtime does. In this situation, you must create your own entity classes and add the appropriate attributes.

Errors in Inheritance Relationships

Q. I used the toolbox inheritance shape in the Object Relational Designer to connect two entities, but I get errors.

A. Creating the relationship is not enough. You must provide information such as the discriminator column, base class discriminator value, and derived class discriminator value.

Provider Model

Q. Is a public provider model available?

A. No public provider model is available. At this time, LINQ to SQL supports SQL Server and SQL Server Compact 3.5 only.

SQL-Injection Attacks

Q. How is LINQ to SQL protected from SQL-injection attacks?

A. SQL injection has been a significant risk for traditional SQL queries formed by concatenating user input. LINQ to SQL avoids such injection by using SqlParameter in queries. User input is turned into parameter values. This approach prevents malicious commands from being used from customer input.

Changing Read-only Flag in DBML Files

Q. How do I eliminate setters from some properties when I create an object model from a DBML file?

A. Take the following steps for this advanced scenario:

1. In the .dbml file, modify the property by changing the IsReadOnly flag to `True`.
2. Add a partial class. Create a constructor with parameters for the read-only members.
3. Review the default UpdateCheck value (Never) to determine whether that is the correct value for your application.

   Caution
   If you are using the Object Relational Designer in Visual Studio, your changes might be overwritten.

APTCA

Q. Is System.Data.Linq marked for use by partially trusted code?

A. Yes, the System.Data.Linq.dll assembly is among those .NET Framework assemblies marked with the AllowPartiallyTrustedCallersAttribute attribute. Without this marking, assemblies in the .NET Framework are intended for use only by fully trusted code.

The principal scenario in LINQ to SQL for allowing partially trusted callers is to enable the LINQ to SQL assembly to be accessed from Web applications, where the *trust* configuration is Medium.

Mapping Data from Multiple Tables

Q. The data in my entity comes from multiple tables. How do I map it?

A. You can create a view in a database and map the entity to the view. LINQ to SQL generates the same SQL for views as it does for tables.

> **NOTE**
> The use of views in this scenario has limitations. This approach works most safely when the operations performed on Table<TEntity> are supported by the underlying view. Only you know which operations are intended. For example, most applications are read-only, and another sizeable number perform `Create`/`Update`/`Delete` operations only by using stored procedures against views.

Connection Pooling

Q. Is there a construct that can help with DataContext pooling?

A. Do not try to reuse instances of DataContext. Each DataContext maintains state (including an identity cache) for one particular edit/query session. To obtain new instances based on the current state of the database, use a new DataContext.

You can still use underlying ADO.NET connection pooling. For more information, see SQL Server Connection Pooling (ADO.NET).

Second DataContext Is Not Updated

Q. I used one instance of DataContext to store values in the database. However, a second DataContext on the same database does not reflect the updated values. The second DataContext instance seems to return cached values.

A. This behavior is by design. LINQ to SQL continues to return the same instances/values that you saw in the first instance. When you make updates, you use optimistic concurrency. The original data is used to check against the current database state to assert that it is in fact still unchanged. If it has changed, a conflict occurs and your application must resolve it. One option of your application is to reset the original state to the current database state and to try the update again. For more information, see How to: Manage Change Conflicts.

You can also set ObjectTrackingEnabled to false, which turns off caching and change tracking. You can then retrieve the latest values every time that you query.

Cannot Call SubmitChanges in Read-only Mode

Q. When I try to call SubmitChanges in read-only mode, I get an error.

A. Read-only mode turns off the ability of the context to track changes.

See Also

Reference
Troubleshooting
Security in LINQ to SQL

SQL Server Compact and LINQ to SQL

5/1/2017 • 1 min to read • Edit Online

SQL Server Compact is the default database installed with Visual Studio. For more information, see PAVE OVER Using SQL Server Compact (Visual Studio).

This topic outlines the key differences in usage, configuration, feature sets, and scope of LINQ to SQL support.

Characteristics of SQL Server Compact in Relation to LINQ to SQL

By default, SQL Server Compact is installed for all Visual Studio editions, and is therefore available on the development computer for use with LINQ to SQL. But deployment of an application that uses SQL Server Compact and LINQ to SQL differs from that for a SQL Server application. SQL Server Compact is not a part of the .NET Framework, and therefore must be packaged with the application or downloaded separately from the Microsoft site.

Note the following characteristics:

- SQL Server Compact is packaged as a DLL that can be used against database files (.sdf extension) directly.
- SQL Server Compact runs in the same process as the client application. The efficiency of communication with SQL Server Compact can therefore be significantly higher than communicating with SQL Server. On the other hand, SQL Server Compact does require interoperability between managed and unmanaged code with its attendant costs.
- The size of the SQL Server Compact DLL is small. This feature reduces the overall application size.
- The LINQ to SQL runtime and the SQLMetal command-line tool support SQL Server Compact.
- The Object Relational Designer does not support SQL Server Compact.

Feature Set

The SQL Server Compact feature set is much simpler than the feature set of SQL Server in the following ways that can affect LINQ to SQL applications :

- SQL Server Compact does not support stored procedures or views.
- SQL Server Compact supports only a subset of data types and SQL functions.
- SQL Server Compact supports only a subset of SQL constructs.
- SQL Server Compact provides only a minimal optimizer. It is possible that some queries might time out.
- SQL Server Compact does not support partial trust.

See Also

Reference

Standard Query Operator Translation

5/1/2017 • 8 min to read • Edit Online

LINQ to SQL translates Standard Query Operators to SQL commands. The query processor of the database determines the execution semantics of SQL translation.

Standard Query Operators are defined against *sequences*. A sequence is *ordered* and relies on reference identity for each element of the sequence. For more information, see Standard Query Operators Overview.

SQL deals primarily with *unordered sets of values*. Ordering is typically an explicitly stated, post-processing operation that is applied to the final result of a query rather than to intermediate results. Identity is defined by values. For this reason, SQL queries are understood to deal with multisets (*bags*) instead of *sets*.

The following paragraphs describe the differences between the Standard Query Operators and their SQL translation for the SQL Server provider for LINQ to SQL.

Operator Support

Concat

The Concat method is defined for ordered multisets where the order of the receiver and the order of the argument are the same. Concat works as `UNION ALL` over the multisets followed by the common order.

The final step is ordering in SQL before results are produced. Concat does not preserve the order of its arguments. To ensure appropriate ordering, you must explicitly order the results of Concat.

Intersect, Except, Union

The Intersect and Except methods are well defined only on sets. The semantics for multisets is undefined.

The Union method is defined for multisets as the unordered concatenation of the multisets (effectively the result of the UNION ALL clause in SQL).

Take, Skip

Take and Skip methods are well defined only against *ordered sets*. The semantics for unordered sets or multisets are undefined.

NOTE

Take and Skip have certain limitations when they are used in queries against SQL Server 2000. For more information, see the "Skip and Take Exceptions in SQL Server 2000" entry in Troubleshooting.

Because of limitations on ordering in SQL, LINQ to SQL tries to move the ordering of the argument of these methods to the result of the method. For example, consider the following LINQ to SQL query:

```
var custQuery =
    (from cust in db.Customers
    where cust.City == "London"
    orderby cust.CustomerID
    select cust).Skip(1).Take(1);
```

```
Dim custQuery = _
    From cust In db.Customers _
    Where cust.City = "London" _
    Order By cust.CustomerID _
    Select cust Skip 1 Take 1
```

The generated SQL for this code moves the ordering to the end, as follows:

```
SELECT TOP 1 [t0].[CustomerID], [t0].[CompanyName],
FROM [Customers] AS [t0]
WHERE (NOT (EXISTS(
    SELECT NULL AS [EMPTY]
    FROM (
        SELECT TOP 1 [t1].[CustomerID]
        FROM [Customers] AS [t1]
        WHERE [t1].[City] = @p0
        ORDER BY [t1].[CustomerID]
        ) AS [t2]
    WHERE [t0].[CustomerID] = [t2].[CustomerID]
    ))) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
```

It becomes obvious that all the specified ordering must be consistent when Take and Skip are chained together. Otherwise, the results are undefined.

Both Take and Skip are well-defined for non-negative, constant integral arguments based on the Standard Query Operator specification.

Operators with No Translation

The following methods are not translated by LINQ to SQL. The most common reason is the difference between unordered multisets and sequences.

OPERATORS	RATIONALE
TakeWhile, SkipWhile	SQL queries operate on multisets, not on sequences. `ORDER BY` must be the last clause applied to the results. For this reason, there is no general-purpose translation for these two methods.
Reverse	Translation of this method is possible for an ordered set but is not currently translated by LINQ to SQL.
Last, LastOrDefault	Translation of these methods is possible for an ordered set but is not currently translated by LINQ to SQL.
ElementAt, ElementAtOrDefault	SQL queries operate on multisets, not on indexable sequences.
DefaultIfEmpty (overload with default arg)	In general, a default value cannot be specified for an arbitrary tuple. Null values for tuples are possible in some cases through outer joins.

Expression Translation

Null semantics

LINQ to SQL does not impose null comparison semantics on SQL. Comparison operators are syntactically translated to their SQL equivalents. For this reason, the semantics reflect SQL semantics that are defined by server or connection settings. For example, two null values are considered unequal under default SQL Server settings, but you can change the settings to change the semantics. LINQ to SQL does not consider server settings when it translates queries.

A comparison with the literal null is translated to the appropriate SQL version (`is null` or `is not null`).

The value of `null` in collation is defined by SQL Server. LINQ to SQL does not change the collation.

Aggregates

The Standard Query Operator aggregate method Sum evaluates to zero for an empty sequence or for a sequence that contains only nulls. In LINQ to SQL, the semantics of SQL are left unchanged, and Sum evaluates to `null` instead of zero for an empty sequence or for a sequence that contains only nulls.

SQL limitations on intermediate results apply to aggregates in LINQ to SQL. The Sum of 32-bit integer quantities is not computed by using 64-bit results. Overflow might occur for a LINQ to SQL translation of Sum, even if the Standard Query Operator implementation does not cause an overflow for the corresponding in-memory sequence.

Likewise, the LINQ to SQL translation of Average of integer values is computed as an `integer`, not as a `double`.

Entity Arguments

LINQ to SQL enables entity types to be used in the GroupBy and OrderBy methods. In the translation of these operators, the use of an argument of a type is considered to be the equivalent to specifying all members of that type. For example, the following code is equivalent:

```
db.Customers.GroupBy(c => c);
db.Customers.GroupBy(c => new { c.CustomerID, c.ContactName });
```

```
db.Customers.GroupBy(Function(c) c)
db.Customers.GroupBy(Function(c) New With {c.CustomerID, _
    c.ContactName})
```

Equatable / Comparable Arguments

Equality of arguments is required in the implementation of the following methods:

Contains

Skip

Union

Intersect

Except

LINQ to SQL supports equality and comparison for *flat* arguments, but not for arguments that are or contain sequences. A flat argument is a type that can be mapped to a SQL row. A projection of one or more entity types that can be statically determined not to contain a sequence is considered a flat argument.

Following are examples of flat arguments:

```
db.Customers.Select(c => c);
db.Customers.Select(c => new { c.CustomerID, c.City });
db.Orders.Select(o => new { o.OrderID, o.Customer.City });
db.Orders.Select(o => new { o.OrderID, o.Customer });
db.Customers.Select(Function(c) c)
db.Customers.Select(Function(c) New With {c.CustomerID, c.City})
db.Orders.Select(Function(o) New With {o.OrderID, o.Customer.City})
db.Orders.Select(Function(o) New With {o.OrderID, o.Customer})
```

The following are examples of non-flat (hierarchical) arguments.

```
// In the following line, c.Orders is a sequence.
db.Customers.Select(c => new { c.CustomerID, c.Orders });
// In the following line, the result has a sequence.
db.Customers.GroupBy(c => c.City);
```

```
' In the following line, c.Orders is a sequence.
db.Customers.Select(Function(c) New With {c.CustomerID, c.Orders})
' In the following line, the result has a sequence.
db.Customers.GroupBy(Function(c) c.City)
```

Visual Basic Function Translation

The following helper functions that are used by the Visual Basic compiler are translated to corresponding SQL operators and functions:

`CompareString`

`DateTime.Compare`

`Decimal.Compare`

`IIf (in Microsoft.VisualBasic.Interaction)`

Conversion methods:

ToBoolean	ToSByte	ToByte	ToChar
ToCharArrayRankOne	ToDate	ToDecimal	ToDouble
ToInteger	ToUInteger	ToLong	ToULong
ToShort	ToUShort	ToSingle	ToString

Inheritance Support

Inheritance Mapping Restrictions

For more information, see How to: Map Inheritance Hierarchies.

Inheritance in Queries

C# casts are supported only in projection. Casts that are used elsewhere are not translated and are ignored. Aside from SQL function names, SQL really only performs the equivalent of the common language runtime (CLR) Convert. That is, SQL can change the value of one type to another. There is no equivalent of CLR cast because there is no concept of reinterpreting the same bits as those of another type. That is why a C# cast works only locally. It is not remoted.

The operators, `is` and `as`, and the `GetType` method are not restricted to the `Select` operator. They can be used in other query operators also.

SQL Server 2008 Support

Starting with the .NET Framework 3.5 SP1, LINQ to SQL supports mapping to new date and time types introduced with SQL Server 2008. But, there are some limitations to the LINQ to SQL query operators that you can use when operating against values mapped to these new types.

Unsupported Query Operators

The following query operators are not supported on values mapped to the new SQL Server date and time types: `DATETIME2`, `DATE`, `TIME`, and `DATETIMEOFFSET`.

- `Aggregate`
- `Average`
- `LastOrDefault`
- `OfType`
- `Sum`

For more information about mapping to these SQL Server date and time types, see SQL-CLR Type Mapping.

SQL Server 2005 Support

LINQ to SQL does not support the following SQL Server 2005 features:

- Stored procedures written for SQL CLR.
- User-defined type.
- XML query features.

SQL Server 2000 Support

The following SQL Server 2000 limitations (compared to Microsoft SQL Server 2005) affect LINQ to SQL support.

Cross Apply and Outer Apply Operators

These operators are not available in SQL Server 2000. LINQ to SQL tries a series of rewrites to replace them with appropriate joins.

`Cross Apply` and `Outer Apply` are generated for relationship navigations. The set of queries for which such rewrites are possible is not well defined. For this reason, the minimal set of queries that is supported for SQL Server 2000 is the set that does not involve relationship navigation.

text / ntext

Data types `text` / `ntext` cannot be used in certain query operations against `varchar(max)` / `nvarchar(max)`, which are supported by Microsoft SQL Server 2005.

No resolution is available for this limitation. Specifically, you cannot use `Distinct()` on any result that contains members that are mapped to `text` or `ntext` columns.

Behavior Triggered by Nested Queries

SQL Server 2000 (through SP4) binder has some idiosyncrasies that are triggered by nested queries. The set of SQL queries that triggers these idiosyncrasies is not well defined. For this reason, you cannot define the set of LINQ to SQL queries that might cause SQL Server exceptions.

Skip and Take Operators

Take and Skip have certain limitations when they are used in queries against SQL Server 2000. For more information, see the "Skip and Take Exceptions in SQL Server 2000" entry in Troubleshooting.

Object Materialization

Materialization creates CLR objects from rows that are returned by one or more SQL queries.

- The following calls are *executed locally* as a part of materialization:
  - Constructors
  - `ToString` methods in projections
  - Type casts in projections
- Methods that follow the AsEnumerable method are *executed locally*. This method does not cause immediate execution.
- You can use a `struct` as the return type of a query result or as a member of the result type. Entities are required to be classes. Anonymous types are materialized as class instances, but named structs (non-entities) can be used in projection.
- A member of the return type of a query result can be of type IQueryable<T>. It is materialized as a local collection.
- The following methods cause the *immediate materialization* of the sequence that the methods are applied to:
  - ToList
  - ToDictionary
  - ToArray

See Also

Reference
Return Or Skip Elements in a Sequence
Concatenate Two Sequences
Return the Set Difference Between Two Sequences
Return the Set Intersection of Two Sequences
Return the Set Union of Two Sequences

Samples

5/1/2017 • 1 min to read • Edit Online

This topic provides links to the Visual Basic and C# solutions that contain LINQ to SQL sample code.

In This Section

Visual Basic version of the SampleQueries solution
Sample Queries (Visual Basic)

C# version of the SampleQueries solution
LINQ C# Samples Solution

Follow these steps to find additional examples of LINQ to SQL code and applications:

- Search the MSDN Library for specific issues.
- Participate in the LINQ Forum, where you can discuss more complex topics in detail with experts.
- Study the white paper that details LINQ to SQL technology, complete with Visual Basic and C# code examples. For more information, see LINQ to SQL: .NET Language-Integrated Query for Relational Data.

See Also

LINQ to SQL
LINQ to SQL Walkthroughs