|  |
| --- |
|  |
| LINQ to SQL I |
| LINQ to Relational Data |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

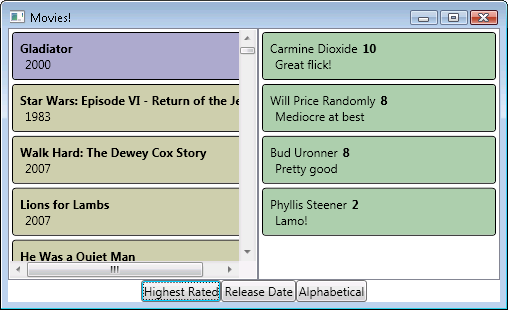
LINQ to SQL I

# Overview

After completing this lab, you should understand the following:

* How to create LINQ to SQL classes using Visual Studio 2008
* How to use a DataContext derived class to query a database using LINQ
* How to examine the T-SQL generated by LINQ to SQL
* How to take advantage of lazy loading in LINQ to SQL

In this lab, we are going to fill in the functionality for a WPF application to display movies, and their associated reviews. You should start this lab by opening the solution file in the LINQ\_SQL\_I\before directory. A completed version of the application exists in the LINQ\_SQL\_I\after directory. The completed application will look similar to the following.



# Part I – Setting Up The Database

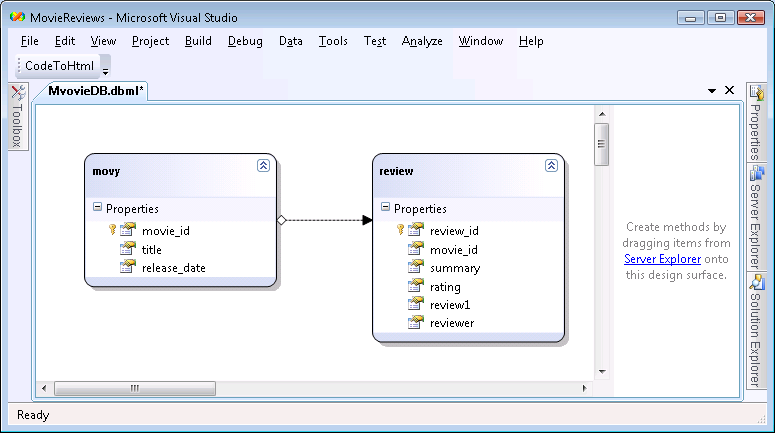
1. This lab requires the *moviereviews* database. If you’ve already installed the database you can move to the next section.
2. To install the database, execute the SQL commands in the Scripts\moviereviews.sql file where you’ve extracted the Pluralsight labs. You can execute the script file using SQL Management Studio, sqlcmd, or any other SQL tool. If you need help, the file Labs\DatabaseSetup\_Troubleshooting.doc file can walk you through the installation.

# Part II – Using the LINQ to SQL Designer

There are a number of options for creating LINQ to SQL classes. In this lab, we will focus on using the Visual Studio designer to generate code, however, for finer grained control you can use the command line tool sqlmetal.exe to generate code, or you can write all the code by hand.

After you’ve opened the solution in LINQ\_SQL\_I\before directory, run the project to ensure you are starting with a working solution. The application window should launch, but no data will display.

1. Right-click the MovieReviews project in the Solution Explorer window and select Add -> New Folder. Give the folder a name of *Model*.
2. Right-click the Model folder and select Add -> New Item. From the Data category, select LINQ to SQL classes. Name the file MovieDB.dbml.
3. You should now have a designer surface open. Click on the “Server Explorer” link in the designer to open the Server Explorer window.
4. Right-click the Data Connection node in Server Explorer and select Add Connection. In the following dialog, establish a connection to your SQL Server’s moviereviews database.
5. In the Server Explorer, expand the moviereviews node, and then expand the Tables node. Drag the movies table into the left-hand side of the designer, then drag the reviews table into the same area. Your designer should look like the following screen shot.



The shapes on the designer represent code-generated classes. The designer has tried to “un-pluralize” the class names for us, which can sometimes create strange results. We can change the names of the classes, and the names of the properties defined in the class from the Properties window.

1. Right-click the movy class on the designer and select Properties.
2. Change the Name property from *movy* to *Movie*, and save.

In the following steps we will examine the LINQ to SQL generated code.

1. In the Solution Explorer window, click to expand the MovieDB.dbml node. You should see a .layout and a .cs file. Open the .cs file.
2. The first class you should see here is the MovieDBDataContext class. This class derives from DataContext and serves as the gateway to the database.
3. Scroll down until you find the Movie class.

[Table(Name="dbo.movies")]

public partial class Movie : INotifyPropertyChanging,

INotifyPropertyChanged

1. Examine the attributes on the Movie class and it’s properties. These attributes map this class to the database..
2. Let’s try to use the generated classes. Open the MainWindow.xaml.cs file. Add a using statement at the beginning of the file for MovieReviews.Model – this is the namespace where our generated code resides.
3. Inside the MainWindow constructor, use MessageBox.Show to display the title of the movie with an ID of 5. You’ll need to instantiate a MovieDBDataContext class and query its Movies property.

public MainWindow()

{

InitializeComponent();

MovieDBDataContext dc = new MovieDBDataContext();

Movie movie = (from m in dc.Movies

where m.movie\_id == 5

select m).Single();

MessageBox.Show(movie.title);

}

1. Press F5 to run your project and ensure a message box appears with the text “Terminator 4”. Exit the application if everything is working, and **delete** all the code we’ve added to the MainWindow constructor.

# Part III – Working with DBML

There are still items we need to fix with our model. For example, all the properties in our model classes are lowercase, which is not what a CLR developer would expect. We could fix these properties in the LINQ to SQL designer by clicking on each property and using the Properties window, however, we are going to take this opportunity to explore the LINQ to SQL dbml file, which often gives us a more expeditious route for changes.

Right click on MovieDB.dbml and select Open With…

1. Select XML Editor and press OK. You should now see a screen full of XML – this is the contents of the dbml file.
2. Right click MovieDB.dbml again and select properties. Notice the Custom Tool property is set to MSLinqToSqlGenerator.

This tells us that a tool will generate code from this XML each time we make changes and save the dbml.

1. Go to the <Database> element. This Class attribute of this element controls the name of the generated DataContext derived class. Change the Class attribute’s value to MovieDB.

<Database Name="moviereviews" **Class="MovieDB"** xmlns="…">

1. On the next line is a <Connection> element. The connection string inside is used by the designer. The SettingsObjectName and SettingsPropertyName are used to generate code that will retrieve a connection string at runtime. We will not change any of these values.
2. The next line is a <Table> element. This element is responsible for mapping the movies table and its columns to a CLR type and its properties. The Type attribute is the CLR type that will represent this table. The Member attribute represents the name of the property that will be generated into the MovieDB class to expose the table. We don’t need to change these settings.
3. Now, move to the first <Column> element. The attributes here tell us the movie\_id column is represented by an int, and is an auto-generated, primary key value. By default, the name of the property that represents this column is move\_id, which looks wrong as a CLR property. Add a Member attribute to the <Column> to tell LINQ to SQL to use a property named MovieID instead.

<Column Name="movie\_id" **Member="MovieID"** Type="System.Int32"

DbType="Int NOT NULL IDENTITY" IsPrimaryKey="true"

IsDbGenerated="true" CanBeNull="false" />

1. Add Member attributes for the title and release\_date columns too, mapping the columns to Title and ReleaseDate properties.

<Column Name="title" **Member="Title"** … />

<Column Name="release\_date" **Member="ReleaseDate"** …/>

1. The last element inside the movie <Table> is an <Association> element that describes how this entity relates to other entities. Change the Member attribute to an uppercase Reviews, change the OtherKey value to MovieID, and change the Type to an uppercase Review. We also need to add a ThisKey attribute with a value of “MovieID”.

<Association **Name="Movie\_Review"** **Member="Reviews"**

**ThisKey="MovieID" OtherKey="MovieID"**

**Type="Review"** />

1. The next <Table> entry represents the reviews table. We want to change the mappings for reviews so that all property names are uppercase.
2. Change the Member attribute for the <Table> to Reviews.
3. Change the <Type> Name to Review
4. Add Member attributes for ReviewID, MovieID, Summary, and Rating in their respective columns.
5. The review column has an odd name (review1), but this is because a C# class cannot have a property with the same name as the class. We can change the Member attribute for the review column to Body.
6. Add a Member attribute for the reviewer column with the value Reviewer.
7. In the Association, make sure to change the ThisKey attribute to MovieID, and add an OtherKey attribute with a value of MovieID.

<Table Name="dbo.reviews" **Member="Reviews"**>

<Type **Name="Review"**>

<Column Name="review\_id" **Member="ReviewID"** …/>

<Column Name="movie\_id" **Member="MovieID"** … />

<Column Name="summary" **Member="Summary"** … />

<Column Name="rating" **Member="Rating"** … />

<Column Name="review" **Member="Body"** … />

<Column Name="reviewer" **Member="Reviewer"** … />

<Association **Name="Movie\_Review"** **Member="Movie"**

**ThisKey="MovieID" OtherKey="MovieID"**

**Type="Movie"** IsForeignKey="true" />

</Type>

</Table>

1. Save and then close the dbml file.
2. **Delete** the MovieDB.dbml.layout file.

This file describes how the dbml appears in the designer. One of the unfortunate side-effects of editing the dbml by hand is we can confuse the layout manager (if we made changes in the designer with the mouse instead, we would not have this problem). The designer will re-create this file in the next step.

1. Double click MovieDB.dbml to open the file in design view. We should see our changes reflected in the designer. Build the project to ensure there are no errors. .

# Part IV – Querying With LINQ to SQL

We are ready to put some data into our application. Open the MainWindow.xaml file.

1. Currently, we have some styles and templates defined, but no bindable controls. Scroll to the bottom of the file. You should find a comment indicating where we intend to place some controls.

<DockPanel >

<StackPanel HorizontalAlignment="Center" …

<!-- buttons -->

</StackPanel>

**<!-- lists -->**

</DockPanel>

1. Replace the *<!-- lists -->* comment with the following XAML.

<ListBox DockPanel.Dock="Left" Width="250"

ItemsSource="{Binding}"

ItemTemplate="{StaticResource \_movieTemplate}"

ItemContainerStyle="{StaticResource \_movieStyle}"

IsSynchronizedWithCurrentItem="True" >

</ListBox>

The above XAML tells the ListBox is will use data binding and sets some styles and data templates you can find in the Window.Resources section of the XAML. Now we just need to fetch the data.

1. Open MainWindowXaml.cs, and add a method to the MainWindow class named FetchMovies. The method should return an IQueryable<Movie>. Inside, use the MovieDB class to return all records from the movies table.

IQueryable<Movie> FetchMovies()

{

MovieDB db = new MovieDB();

return db.Movies;

}

1. Override the OnInitialized method of the MainWindow. After calling the base class OnInitialized method, invoke the FetchMovies method and assign the result to \_mainWindow.DataContext.

protected override void OnInitialized(EventArgs e)

{

base.OnInitialized(e);

\_mainWindow.DataContext = FetchMovies();

}

Don’t confuse the DataContext property in WPF controls with the DataContext class of LINQ to SQL. The DataContext property in WPF is one available mechanism for giving controls an object for data binding.

1. Press F5 to run the application. You should see all the movies from the database in the main window of the application.



1. Let’s add some ordering to our records. Replace the *<!-- buttons -->* comment in the XAML with the following markup.

<Button Click="ReleaseDateClick">Release Date</Button>

<Button Click="AlphabeticalClick">Alphabetical</Button>

1. Add the following event handlers for the button click events to the XAML’s code-behind file.

void AlphabeticalClick(object sender,

RoutedEventArgs e)

{

}

void ReleaseDateClick(object sender,

RoutedEventArgs e)

{

}

1. For the AlphabeticalClick event handler, use the FetchMovies method and apply a LINQ OrderBy operator to sort the movies by Title. Set the result to the \_mainWindow object’s DataContext property.

void AlphabeticalClick(object sender,

RoutedEventArgs e)

{

\_mainWindow.DataContext =

FetchMovies().OrderBy(m => m.Title);

}

1. For the ReleaseDateClick event handler, apply a LINQ OrderBy operator to sort the movies by ReleaseDate. Set the result to the \_mainWindow object’s DataContext property.

void ReleaseDateClick(object sender,

RoutedEventArgs e)

{

\_mainWindow.DataContext =

FetchMovies().OrderBy(m => m.ReleaseDate);

}

1. Run the application again to verify the new buttons are correctly sorting the results.
2. Before we move to the next section, think about the answer to the following question: when you click the button, are the movies being sorted inside the database server by a SQL query, or in-memory as CLR objects? We’ll find the answer in the next section.

# Part V – Logging LINQ Queries

Let’s rearrange the code in the ReleaseDateClick event handler. We need a variable to hold our LINQ query so we can inspect the query in the debugger.

void ReleaseDateClick(object sender,

RoutedEventArgs e)

{

var query = FetchMovies().OrderBy(m => m.ReleaseDate);

\_mainWindow.DataContext = query;

}

1. Place a breakpoint on the line that assigns the query to the DataContext property.
2. Press F5 to run the project with debugging, and click the Release Date button.
3. Hover your mouse over the query variable. The debugger should show you the SQL string the query will execute when we try to enumerate the results. The query should look like the following.

SELECT [t0].[movie\_id] AS [MovieID], [t0].[title] AS [Title],

[t0].[release\_date] AS [ReleaseDate]

FROM [dbo].[movies] AS [t0]

ORDER BY [t0].[release\_date]

1. As you can see – LINQ to SQL has put our OrderBy operator into the SQL statement that it will send to SQL Server.

We composed a query by building on the base query in FetchMovies. This demonstration should also make the deferred execution of LINQ to SQL obvious. Stop debugging, exit the application, and remove the debugging breakpoint.

1. There is another technique we can use to watch the generated SQL of LINQ to SQL. Add the highlighted line to the FetchMovies method.

IQueryable<Movie> FetchMovies()

{

MovieDB db = new MovieDB();

**db.Log = Console.Out;**

return db.Movies;

}

1. The Log property allows us to redirect the logging output of LINQ to SQL. We are going to send the output to Console.Out, which will appear in Visual Studio’s Output window when debugging.
2. Press F5 to run the application again. Open the Output window of Visual Studio (Ctrl+Alt+O).
3. As you click the buttons in the application, you should be able to observe SQL commands in the Output window. This will come in useful for the next section.

# Part VI – Master Detail / Deferred Execution

In this section, we will also add each Movie’s review to the display.

Open MainWindow.xaml and add the following XAML just below the ListBox that displays the movies.

<ListBox ItemsSource="{Binding Reviews}"

ItemTemplate="{StaticResource \_reviewTemplate}"

ItemContainerStyle="{StaticResource \_reviewStyle}"

IsSynchronizedWithCurrentItem="True">

</ListBox>

1. Notice the ItemsSource for this ListBox is set to bind to the Reviews property of the currently bound data object. Since we are using IsSynchronizedWithCurrentItem, this should display the reviews of the currently selected movie.
2. Press F5 to run the application.
3. Open the Output window again, if it is not already open (Ctrl+Alt+O).
4. Click on different movies. Notice that each time you click on a new movie, LINQ to SQL issues a new query to the database. This is an example of implicit lazy loading. The Reviews property of each movie is not populated until something actually asks for the Reviews.

# Conclusion

Congratulations! You’ve built an application using LINQ to SQL. You’ve seen how deferred execution, query composition, and deferred lazy loading works. At this point you should be comfortable bringing all of your LINQ knowledge to bear on relational databases.