|  |
| --- |
|  |
| LINQ to SQL II |
| Modifications |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

LINQ to SQL II

# Overview

In this lab, we will use WPF and LINQ to SQL in an application that can add, update, and delete movie reviews. This lab builds on top of the LINQ to SQL I lab. You can use your completed project from the LINQ to SQL I lab, or open the solution in the labs\LINQ\_SQL\_II\before directory. The after directory contains a completed version of this lab. By the end of this lab, you should understand the following:

* How to create, update, and delete data with LINQ to SQL
* How to use the DataContext class as a “unit of work”
* Understand how a DataContext identity map influences the results of a LINQ query.

This lab requires the *moviereviews* database. 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 I – Deleting Reviews

1. Open the MovieReviews solution file and run the application. Make sure the application displays a list of movies and their reviews. If everything is working, you can exit the application.
2. Open the MainWindow.xaml file. Locate the ListBox control that displays movie reviews. The ListBox will be near the end of the MainWindow.xaml file and will have an ItemsSource binding set to Reviews. We will need to ability to interact with this ListBox with our C# code, so give the ListBox an x:Name attribute of “\_reviewsList”.

<ListBox x:Name="\_reviewsList"

ItemsSource="{Binding Reviews}"

…

</ListBox>

1. We need to add buttons for adding, deleting, and editing the currently selected review. We want the buttons to appear underneath the \_reviewsList control. Surround the \_reviewsList with a StackPanel control. The Orientation property of the StackPanel should be set to Vertical.

**<StackPanel Orientation="Vertical">**

<ListBox x:Name="\_reviewsList"

…

</ListBox>

**</StackPanel>**

1. Inside the new StackPanel, add another StackPanel *after* the ListBox control. Set this stack panel’s Orientation property to “Horizontal” and its HorizontalAlignment property to “Center”.

<StackPanel Orientation="Vertical">

<ListBox x:Name="\_reviewsList" …

</ListBox>

**<StackPanel Orientation="Horizontal"**

**HorizontalAlignment="Center" >**

**</StackPanel>**

</StackPanel>

1. Inside the innermost StackPanel, add three Button controls as shown below:

<StackPanel Orientation="Horizontal"

HorizontalAlignment="Center" >

**<Button>Insert</Button>**

**<Button>Edit</Button>**

**<Button>Delete</Button>**

</StackPanel>

1. Let’s concentrate on the delete functionality first. Give the delete button a click event handler in XAML.

<Button **Click="DeleteButton\_Click"**>Delete</Button>

1. Right-click the “DeleteButton\_Click” text in XAML and select “Navigate to Event Handler”. Visual Studio will create the event handling method for you, and take you to the MainWindow.xaml.cs file.
2. Let’s try to delete the currently selected review by using the SelectedItem property of \_reviewsList. Instantiate a new MovieDB data context and pass the currently selected review to the DeleteOnSubmit method of the Reviews property. Finally, call SubmitChanges on the data context object to flush the changes.

private void DeleteButton\_Click(object sender,

RoutedEventArgs e)

{

Review selectedReview = \_reviewsList.SelectedItem

as Review;

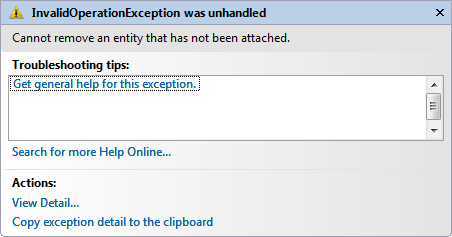
MovieDB db = new MovieDB();

db.Reviews.DeleteOnSubmit(selectedReview);

db.SubmitChanges();

}

1. Run the application using the debugger (F5). Select a movie, then a review. Click the Delete button. The debugger should halt the program and tell you about an unhandled exception.



The exception is slightly misleading. It’s true that we cannot delete an entity that the DataContext class (MoviesDB) does not know about. Either we use the DataContext to retrieve the entity and then delete the entity, or we attach the entity to the DataContext and then delete the entity. There is one catch with the Attach method, however, **we cannot Attach an entity that has been instantiated by another DataContext**.

1. Try making the highlighted change below.

private void DeleteButton\_Click(object sender,

RoutedEventArgs e)

{

Review selectedReview = \_reviewsList.SelectedItem

as Review;

MovieDB db = new MovieDB();

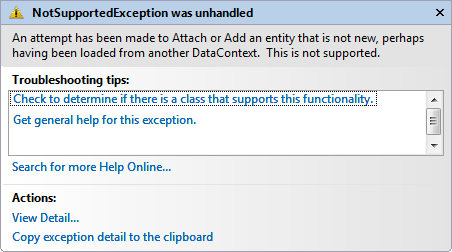
**db.Reviews.Attach(selectedReview);**

db.Reviews.DeleteOnSubmit(selectedReview);

db.SubmitChanges();

}

1. Run the program with the debugger (F5), and you should find another unhandled exception.



The two exceptions we’ve seen demonstrate some fundamental ground rules you need to know when working with LINQ to SQL. A DataContext object actively tracks changes to objects it instantiates when you query the database. You cannot ask one DataContext to manage an entity that has been instantiated by a different DataContext. The Attach API is for entities that move across application domains, like serialized entities sent over the wire in a web service call.

Our fundamental problem is that the application is using a different DataContext object each time it interacts with the database. When using object-relational mapping framework like LINQ to SQL, you need to define your “unit of work” so that you are using a single DataContext per “business conversation”. When we query the database, we should assume the user would want to edit the data they are seeing. We will change our application to hold a reference to the DataContext object we create.

1. Add a private field of type MovieDB to the MainWindow. Name the field \_db.
2. In the FetchMovies method, remove the assignment of the new MovieDB to a local variable and assign the new object to the \_db field. Also, change the other local variable references to use the \_db field.

IQueryable<Movie> FetchMovies()

{

\_**db = new MovieDB();**

**\_db.Log = Console.Out;**

**return \_db.Movies;**

}

1. Remove the Attach method call in the DeleteButton\_Click method, and also remove the code that constructs a new instance of MovieDB. Change the method to use the \_db field instead.

private void DeleteButton\_Click(object sender,

RoutedEventArgs e)

{

**Review selectedReview = \_reviewsList.SelectedItem**

**as Review;**

**\_db.Reviews.DeleteOnSubmit(selectedReview);**

**\_db.SubmitChanges();**

}

1. Run the application with the debugger (F5) and open the Visual Studio Output Window (Ctrl+Alt+O). In the application, select a movie review and click the Delete button. Look for two thing to happen:
   * In the Visual Studio Output Window, you should see LINQ to SQL issue a DELETE command to SQL Server. The Delete command will remove the review from the database.
   * In the application, the WPF interface **does not** remove the review from the \_reviewList control.

Unfortunately, WPF does not know that the reviews collection changed. The EntitySet collection used by the LINQ to SQL generated classes does not implement the INotifyCollectionChanged interface required by WPF to refresh a data bound collection. We are going to fix this problem by foregoing automatic data binding and putting the reviews on display into an ObservableCollection collection.

1. Stop the application and open MainWindow.xaml. Find the \_reviewsList control near the bottom of the file and remove the ItemsSource attribute.

<ListBox x:Name="\_reviewsList"

ItemTemplate="{StaticResource \_reviewTemplate}"

ItemContainerStyle="{StaticResource \_reviewStyle}"

IsSynchronizedWithCurrentItem="True">

</ListBox>

1. Just above the \_reviewsList is another ListBox control that displays the Movie objects. We will need to reference this list from our C# code, so add an x:Name attribute to the list with the value “\_moviesList”. Also, add an event handler for the SelectionChanged event, as shown in the highlighted section below.

<ListBox **x:Name="\_moviesList"**

**SelectionChanged="MovieSelection\_Changed"**

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

ItemsSource="{Binding}"

ItemTemplate="{StaticResource \_movieTemplate}"

ItemContainerStyle="{StaticResource \_movieStyle}"

IsSynchronizedWithCurrentItem="True">

</ListBox>

1. Right-click on the MovieSelection\_Changed text for SelectionChanged and click on “Navigate to Event Handler” in the context menu.
2. You should now be in MainWindow.xaml.cs. Add a using statement at the top of the file for the System.Collections.ObjectModel namespace.
3. Inside the MovieSelection\_Change event handler, we need to create an ObservableCollection<Review> using the reviews of the currently selected movie. The new collection needs to be assigned to the ItemsSource property of the \_reviewsList control.

private void MovieSelection\_Changed(

object sender,

SelectionChangedEventArgs e)

{

Movie selectedMovie = \_moviesList.SelectedItem as Movie;

var reviews = new ObservableCollection<Review>(

selectedMovie.Reviews.ToList());

\_reviewsList.ItemsSource = reviews;

}

1. One more piece of work left to complete. When the user deletes a review, we need to remove the review from the database and also from the observable collection. We can do this in the DeleteButton\_Click method.

private void DeleteButton\_Click(object sender,

RoutedEventArgs e)

{

Review selectedReview = \_reviewsList.SelectedItem

as Review;

\_db.Reviews.DeleteOnSubmit(selectedReview);

\_db.SubmitChanges();

**ObservableCollection<Review> collection =**

**\_reviewsList.ItemsSource as**

**ObservableCollection<Review>;**

**collection.Remove(selectedReview);**

}

1. Finally, run the application in the debugger (F5). You should see a DELETE command issued in the Visual Studio Output Window, and you should also see the review removed from the \_reviewsList display.

# Part II – Editing Review Data

In this section, we will allow the user to edit a review.

The *before* directory includes a new window (EditWindow.xaml) to edit the properties of a review. Right click the MovieReviews project and select Add -> Existing Item.

1. Select **both** EditWindow.xaml and EditWindow.xaml.cs in the resulting Add Existing Item dialog. You may need to change the file filter in the lower right corner of the dialog to display All Files (\*.\*). Select both files and click Add.
2. Open EditWindow.xaml and examine the user interface. Notice the window uses data binding to edit a review. Build the project (Shift+Ctrl+B) to ensure everything is in working order.
3. Open MainWindow.xaml and go to the Edit button we placed in the XAMl earlier. Give the button a Click event handler named EditButton\_Click.

<Button **Click="EditButton\_Click"**>Edit</Button>

1. Right click in the EditButton\_Click text and select “Navigate to Event Handler”.
2. You should now be in the EditButton\_Click event of MainWindow.cs.
3. Because the user might Cancel their editing actions, we need to make a temporary copy of the review being edited. We’ll create a new EditWindow, set the EditWindow’s DataContext property to the temporary review, then show the EditWindow.

private void EditButton\_Click(object sender,

RoutedEventArgs e)

{

Review selectedReivew = \_reviewsList.SelectedItem

as Review;

Review tempReview = new Review

{

Body = selectedReivew.Body,

Rating = selectedReivew.Rating,

Reviewer = selectedReivew.Reviewer,

Summary = selectedReivew.Summary

};

EditWindow window = new EditWindow();

window.DataContext = tempReview;

}

1. After setting the DataContext property, we need to show the EditWindow as a modal dialog. If the ShowDialog method returns true, we’ll need to copy the edited values into the selected review and submit changes to the database.

private void EditButton\_Click(object sender,

RoutedEventArgs e)

{

...

window.DataContext = tempReview;

**if (window.ShowDialog() == true)**

**{**

**selectedReivew.Body = tempReview.Body;**

**selectedReivew.Rating = tempReview.Rating;**

**selectedReivew.Reviewer = tempReview.Reviewer;**

**selectedReivew.Summary = tempReview.Summary;**

**\_db.SubmitChanges();**

**}**

}

1. Run the application with the debugger and watch your Output Window in Visual Studio. You should be able to spot the UPDATE commands issued by LINQ to SQL during a successful edit operation.

# Part IV – Inserting Data

The insert data scenario is similar to the update scenario. First, we need an event handler for the insert button. Open MainWindow.xaml and find the insert button near the bottom of the file. Add a Click event handler named InsertButton\_Click.

<Button Click="InsertButton\_Click">Insert</Button>

1. Right-click on the InsertButton\_Click text and select “Navigate to Event Handler”.
2. You should now be inside the new event handler in MainWindow.xaml.cs. We need to instantiate a new, blank review and bind the review to a new EditWindow. If the edit window’s ShowDialog method returns true, we add the object to the Reviews property of our DataContext and submit changes. Don’t forget we also need to add the review to the observable collection of the \_reviewsList.

private void InsertButton\_Click(object sender,

RoutedEventArgs e)

{

Review newReview = new Review();

EditWindow window = new EditWindow();

window.DataContext = newReview;

if (window.ShowDialog() == true)

{

\_db.Reviews.InsertOnSubmit(newReview);

\_db.SubmitChanges();

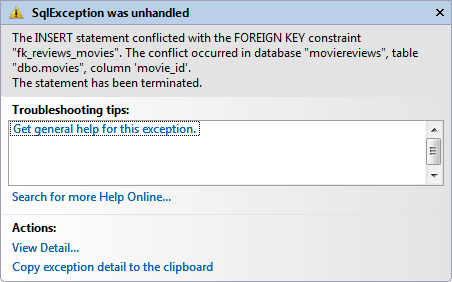
((IList<Review>)\_reviewsList.ItemsSource)

.Add(newReview);

}

}

1. Run the project under the debugger. Trying to save a new review will throw an exception, even if you fill out all the review fields properly (more on validation in a moment).



1. We are responsible for assigning the new review to a movie. We don’t do this using the MovieID property, but with the Movie property (of type EntityRef). LINQ to SQL will take care of the rest.

if (window.ShowDialog() == true)

{

**Movie selectedMovie = \_moviesList.SelectedItem**

**as Movie;**

**newReview.Movie = selectedMovie;**

\_db.Reviews.InsertOnSubmit(newReview);

\_db.SubmitChanges();

((IList<Review>)\_reviewsList.ItemsSource)

.Add(newReview);

}

1. Run the application in the debugger once again. You should verify that saving a new review generates an INSERT command to SQL Server by watching commands in Visual Studio’s Output Window. You should also verify that the user interface updates correctly.
2. While filling out the review edit window, you might have wondered where to place validation code. Although validation in WPF is beyond the scope of this course, we can demonstrate some of the LINQ to SQL extensibility available for validation and business logic.
3. Return to the Model subdirectory of the application and open the MovieDB.designer.cs file that is underneath MovieDB.dbml. Remember from the last lab that this is the code generated from the dbml file.
4. Find the Review class.
5. Inside the class will be a #region entitled Extensibility Method Definitions. Expand the region if it is currently collapsed.
6. Notice we have **partial** Changing and Changed methods defined for each public property. Let’s use this to restrict the value of the ratings stored in our object.
7. Double-click the MovieDB.dbml file. The designer should appear.
8. Right-click the designer and select “View Code”. This should take you to the MovieDB.cs file.
9. Create a partial class definition for Review. Ensure your partial class definition is in the same namespace as the class definition generated by the LINQ tools (MovieReviews.Model).
10. Provide a partial method implementation for OnRatingChanged(int value).

partial class Review

{

partial void OnRatingChanging(int value)

{

}

}

1. Inside the method, check that the incoming value is between 1 and 10 inclusive. If not, throw a new ArgumentException (you might need to add a using statement for System).

partial void OnRatingChanging(int value)

{

**if (value < 1 || value > 10)**

**{**

**throw new ArgumentException(**

**"Rating must be between 1 and 10");**

**}**

}

1. Place a breakpoint inside the *if* statement and run the application using the debugger. When you edit or create a new review and try to use a rating outside of the expected range, this breakpoint should be hit. Although we are not processing these errors in our application, the exception will prevent the data binding code from setting a illegal value in our Rating property.

# Conclusion

Congratulations! You’ve built an application using LINQ to SQL and WPF that can read, edit, delete, and insert reviews. You should have a better understanding of identity maps and the unit of work pattern in LINQ to SQL. Just remember that objects you retrieve from a DataContext are “owned” by that context. You won’t be able to move the objects to another context, nor see database updated reflected in those objects. Coming to grip with this behavior is an important step towards success with LINQ to SQL and object-relational mapping frameworks in general.