**Create a New MVC Database**

SQL Server : (localdb)\MSSQLLocalDB

Database : Movies

**Using the Repository Pattern**

Working directly with LINQ to SQL Class in a controller class creates problems when you need to build a more complex application.

Using LINQ to SQL within a controller class makes it difficult to switch data access technologies in the future. For example, you might decide to switch from using Microsoft LINQ to SQL to using the Microsoft Entity Framework as your data access technology. In that case, you would need to rewrite every controller that accesses the database within your application.

Using LINQ to SQL within a controller class also makes it difficult to build unit tests for your application. Normally, you do not want to interact with a database when performing unit tests.

In order to build an MVC application that is more adaptable to future change and that can be more easily tested, you should consider using the Repository pattern. When you use the Repository pattern, you create a separate repository class that contains all of your database access logic.

Interface for Repository Pattern

using System.Collections.Generic;

namespace MvcApplication1.Models

{

     public interface IMovieRepository

     {

          IList<Movie> ListAll();

     }

}

Class Implementing Interface

using System.Collections.Generic;

using System.Linq;

namespace MvcApplication1.Models

{

     public class MovieRepository : IMovieRepository

     {

          private MovieDataContext \_dataContext;

          public MovieRepository()

          {

                \_dataContext = new MovieDataContext();

          }

          #region IMovieRepository Members

          public IList<Movie> ListAll()

          {

               var movies = from m in \_dataContext.Movies

                    select m;

               return movies.ToList();

          }

          #endregion

     }

}

**Create a Template with a Partial**

When a view gets too complicated, it is a good idea to start breaking the view into partials. Using partials makes your views easier to understand and maintain.

<% foreach (var item in Model) { %>

<% Html.RenderPartial("MovieTemplate", item); %>

<% } %>

<% foreach (var item in Model) { %>

<% Html.Partial("MovieTemplate", item); %>

<% } %>

Html.Partial can store the output in a variable and Html.RenderPartial cannot, It returns Void.

**Performing Simple Validation**

**Understanding Model State**

if(ModelState.IsValid)

{

MoviesEntities entities = new MoviesEntities();

entities.Movies.Add(movie);

entities.SaveChanges();

return RedirectToAction("Index");

}

ModelState is a valid Dictionary type

//

// Summary:

// Represents the state of an attempt to bind a posted form to an action method,

// which includes validation information.

[DefaultMember("Item")]

public class ModelStateDictionary : IDictionary<string, ModelState>, …

{

…

}

## Creating a Partial Class

## The Movie class is a partial class. That means that we can add another partial class with the same name to extend the functionality of the Movie class.

## When the Entity Framework generates an entity class, the Entity Framework adds partial methods to the class automatically. The Entity Framework generates OnChanging and OnChanged partial methods that correspond to each property of the class.

## If IDataErrorInfo interface is implemented in partial class and OnChanging or OnChanged methods are implemented, Validations can be performed.

## The DefaultModelBinder detects whether or not a class implements the IDataErrorInfo interface. If a class implements this interface then the model binder invokes the IDataErrorInfo.this indexer for each property of the class. If the indexer returns an error message then the model binder adds this error message to model state automatically.

The DefaultModelBinder also checks the IDataErrorInfo.Error property.

To Escape this path, Simple Data annotations are used to just use simple attribute level validation.

## 