Displaying a Table of Database Data

This is the **C#** tutorial (Switch to the Visual Basic tutorial)

In this tutorial, I demonstrate two methods of displaying a set of database records. I show two methods of formatting a set of database records in an HTML table. First, I show how you can format the database records directly within a view. Next, I demonstrate how you can take advantage of partials when formatting database records.

Download the code for this tutorial | Download the tutorial in PDF format

« Previous Tutorial | Next Tutorial »

Displaying a Table of Database Data (C#)

The goal of this tutorial is to explain how you can display an HTML table of database data in an ASP.NET MVC application. First, you learn how to use the scaffolding tools included in Visual Studio to generate a view that displays a set of records automatically. Next, you learn how to use a partial as a template when formatting database records.

Create the Model Classes

We are going to display the set of records from the Movies database table. The Movies database table contains the following columns:

Column Name	Data Type	Allow Nulls
Id	Int	False
Title	Nvarchar(200)	False

Director	NVarchar(50)	False
DateReleased	DateTime	False

In order to represent the Movies table in our ASP.NET MVC application, we need to create a model class. In this tutorial, we use the Microsoft Entity Framework to create our model classes.

In this tutorial, we use the Microsoft Entity Framework. However, it is important to understand that you can use a variety of different technologies to interact with a database from an ASP.NET MVC application including LINQ to SQL, NHibernate, or ADO.NET.

Follow these steps to launch the Entity Data Model Wizard:

1. Right-click the Models folder in the Solution Explorer window and the select the menu option **Add, New Item**.
2. Select the **Data** category and select the **ADO.NET Entity Data Model** template.
3. Give your data model the name *MoviesDBModel.edmx* and click the **Add** button.

After you click the Add button, the Entity Data Model Wizard appears (see Figure 1). Follow these steps to complete the wizard:

1. In the **Choose Model Contents** step, select the **Generate from database** option.
2. In the **Choose Your Data Connection** step, use the *MoviesDB.mdf* data connection and the name *MoviesDBEntities* for the connection settings. Click the **Next** button.
3. In the **Choose Your Database Objects** step, expand the Tables node, select the Movies table. Enter the namespace *Models* and click the **Finish** button.

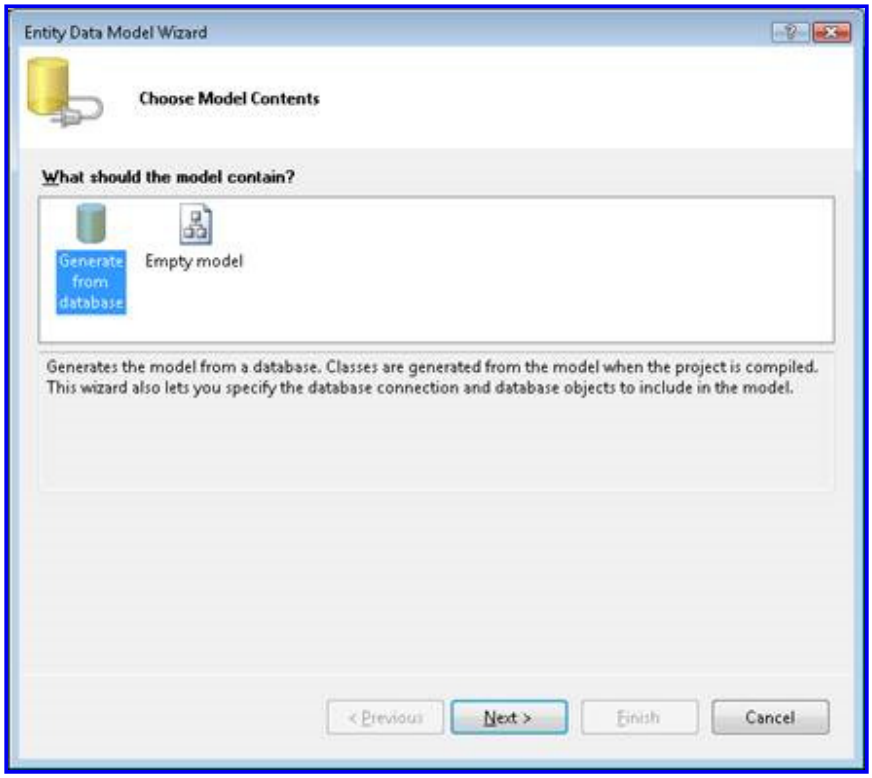

Figure 01: Creating LINQ to SQL classes (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer opens. The Designer should display the Movies entity (see Figure 2).

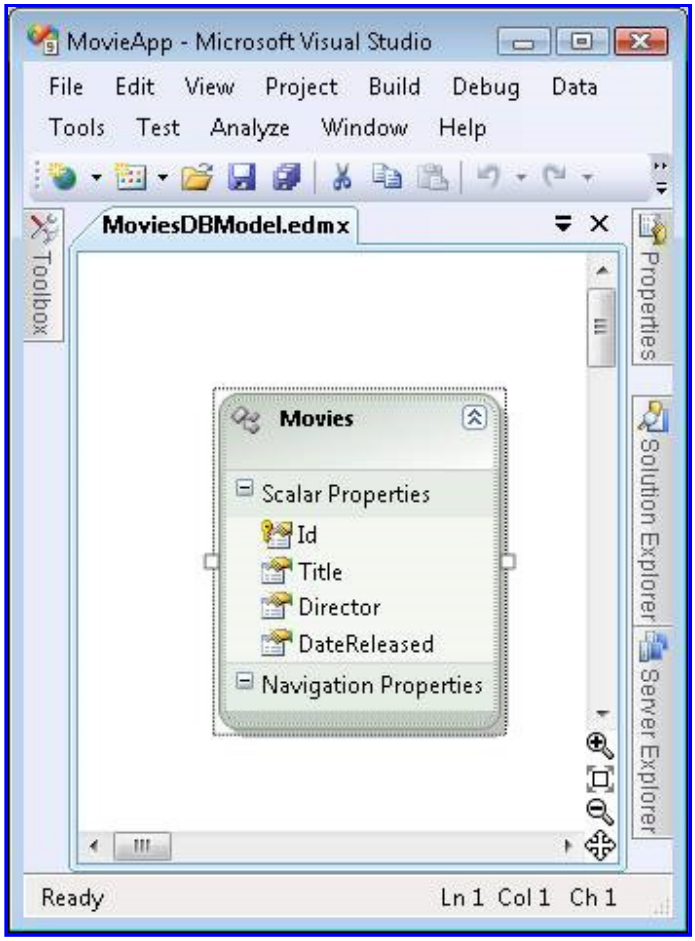

Figure 02: The Entity Data Model Designer (Click to view full-size image)

We need to make one change before we continue. The Entity Data Wizard generates a model class named *Movies* that represents the Movies database table. Because we'll use the Movies class to represent a particular movie, we need to modify the name of the class to be *Movie* instead of *Movies* (singular rather than plural).

Double-click the name of the class on the designer surface and change the name of the class from Movies to Movie. After making this change, click the **Save** button (the icon of the floppy disk) to generate the Movie class.

Create the Movies Controller

Now that we have a way to represent our database records, we can create a controller that returns the collection of movies. Within the Visual Studio Solution Explorer window, right-click the Controllers folder and select the menu option **Add, Controller** (see Figure 3).

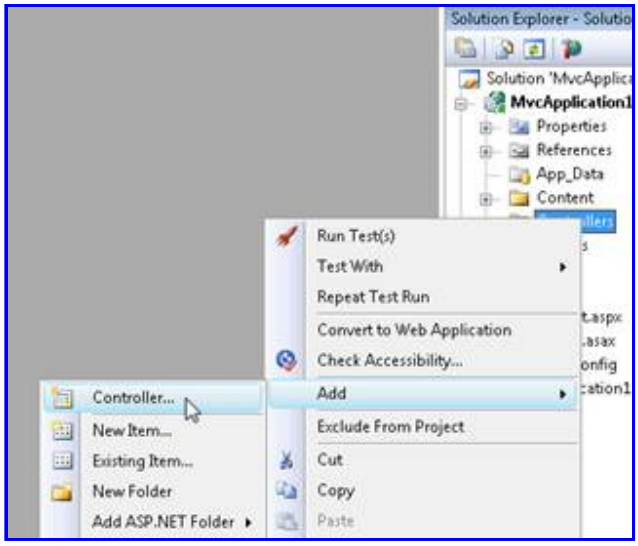

Figure 03: The Add Controller Menu (Click to view full-size image)

When the **Add Controller** dialog appears, enter the controller name MovieController (see Figure 4). Click the **Add** button to add the new controller.

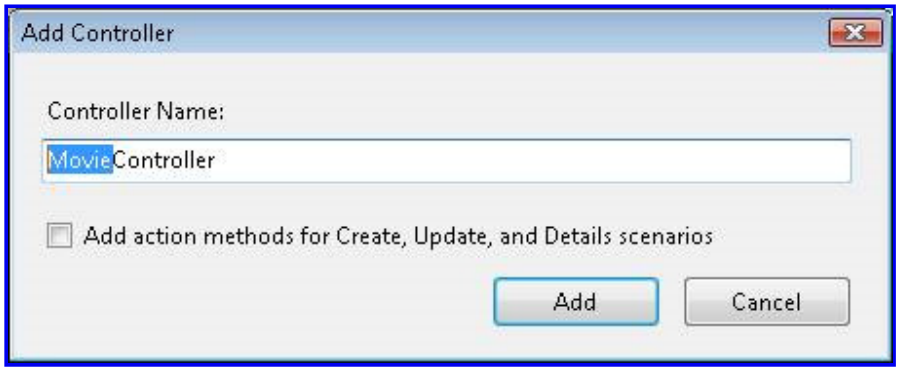

Figure 04: The Add Controller dialog(Click to view full-size image)

We need to modify the Index() action exposed by the Movie controller so that it returns the set of database records. Modify the controller so that it looks like the controller in Listing 1.

Listing 1 – Controllers\MovieController.cs

```
using System.Linq;

using System.Web.Mvc;

using MvcApplication1.Models;

namespace MvcApplication1.Controllers

{

    public class MovieController : Controller

    {

        //

        // GET: /Movie/
```

```
        public ActionResult Index()
        {
            var entities = new MoviesDBEntities();
            return View(entities.MovieSet.ToList());
        }
    }
}
```

In Listing 1, the MoviesDBEntities class is used to represent the MoviesDB database. To use this class, you need to import the MvcApplication1.Models namespace like this:

using MvcApplication1.Models;

The expression *entities.MovieSet.ToList()* returns the set of all movies from the Movies database table.

Create the View

The easiest way to display a set of database records in an HTML table is to take advantage of the scaffolding provided by Visual Studio.

Build your application by selecting the menu option **Build, Build Solution**. You must build your application before opening the **Add View** dialog or your data classes won't appear in the dialog.

Right-click the Index() action and select the menu option **Add View** (see Figure 5).

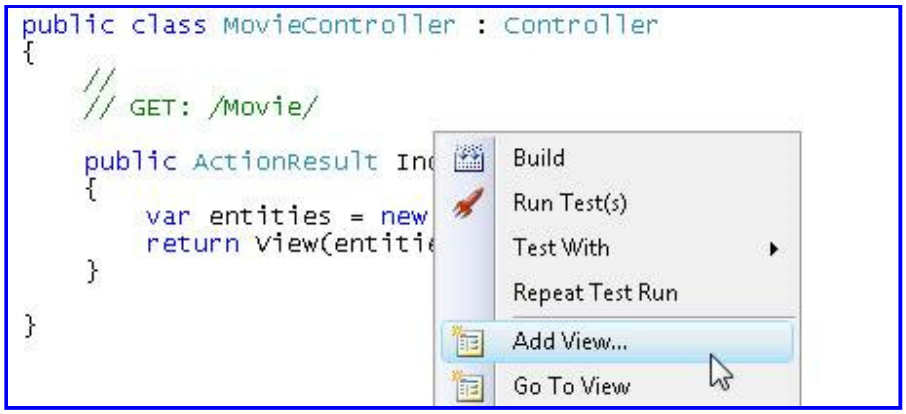

Figure 05: Adding a view (Click to view full-size image)

In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view**. Select the Movie class as the **view data class**. Select *List* as the **view content** (see Figure 6). Selecting these options will generate a strongly-typed view that displays a list of movies.

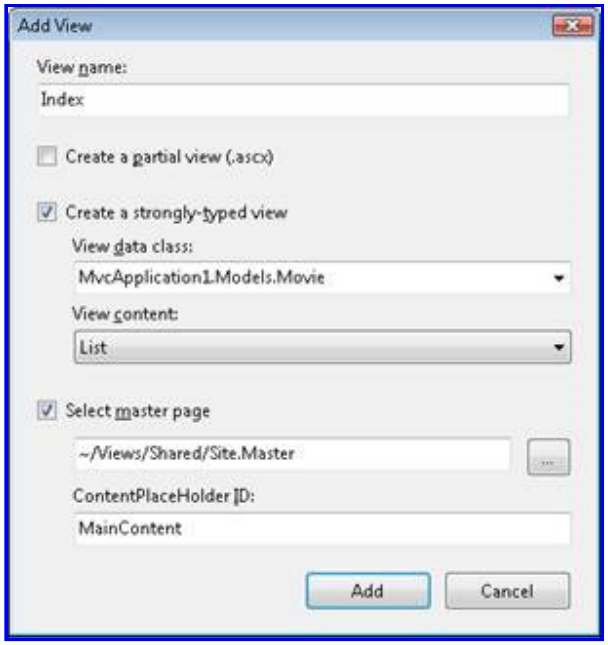

Figure 06: The Add View dialog(Click to view full-size image)

After you click the **Add** button, the view in Listing 2 is generated automatically. This view contains the code required to iterate through the collection of movies and display each of the properties of a movie.

Listing 2 – Views\Movie\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="/Views/Shared/Site.Master" Inherits="System.

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

      Index

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <table>

        <tr>

            <th></th>

            <th>

                Id

            </th>

            <th>

                Title

            </th>

            <th>

                Director

            </th>
```

```
            <th>
                DateReleased
            </th>
        </tr>
    <% foreach (var item in Model) { %>

        <tr>
            <td>
                <%= Html.ActionLink("Edit", "Edit", new { id=item.Id }) %> |
                <%= Html.ActionLink("Details", "Details", new { id=item.Id })%>
            </td>
            <td>
                <%= Html.Encode(item.Id) %>
            </td>
            <td>
                <%= Html.Encode(item.Title) %>
            </td>
            <td>
                <%= Html.Encode(item.Director) %>
            </td>
            <td>
                <%= Html.Encode(String.Format("{0:g}", item.DateReleased)) %>
            </td>
        </tr>

    <% } %>
    </table>
    <p>
        <%= Html.ActionLink("Create New", "Create") %>
    </p>
</asp:Content>
```

You can run the application by selecting the menu option **Debug, Start Debugging** (or hitting the F5 key). Running the application launches Internet Explorer. If you navigate to the /Movie URL then you'll see the page in Figure 7.

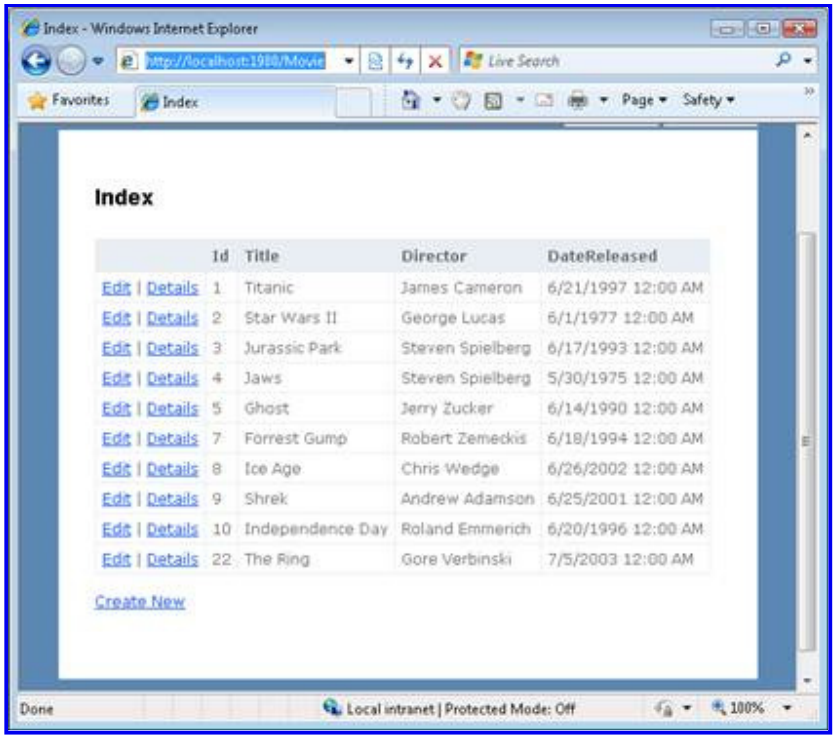

Figure 07: A table of movies(Click to view full-size image)

If you don't like anything about the appearance of the grid of database records in Figure 7 then you can simply modify the Index view. For example, you can change the *DateReleased* header to *Date Released* by modifying the Index view.

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. We'll create a partial that we can use as a template to format each of the movie database records.

Follow these steps to create the partial:

1. Right-click the Views\Movie folder and select the menu option **Add View**.
2. Check the checkbox labeled *Create a partial view (.ascx)*.
3. Name the partial *MovieTemplate*.
4. Check the checkbox labeled **Create a strongly-typed view**.
5. Select Movie as the *view data class*.
6. Select Empty as the *view content*.
7. Click the **Add** button to add the partial to your project.

After you complete these steps, modify the MovieTemplate partial to look like Listing 3.

Listing 3 – Views\Movie\MovieTemplate.ascx

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<MvcApplication1.Models.Mc

<tr>

    <td>

        <%= Html.Encode(Model.Id) %>
```

```
    </td>
    <td>
        <%= Html.Encode(Model.Title) %>
    </td>
    <td>
        <%= Html.Encode(Model.Director) %>
    </td>
    <td>
        <%= Html.Encode(String.Format("{0:g}", Model.DateReleased)) %>
    </td>
</tr>
```

The partial in Listing 3 contains a template for a single row of records.

The modified Index view in Listing 4 uses the MovieTemplate partial.

Listing 4 – Views\Movie\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="/Views/Shared/Site.Master" Inherits="System.
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Index</h2>
    <table>
        <tr>
            <th>
                Id
            </th>
            <th>
                Title
            </th>
            <th>
                Director
            </th>
            <th>
                DateReleased
            </th>
        </tr>
    <% foreach (var item in Model) { %>
```

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


    <% } %>

    </table>

</asp:Content>
```

The view in Listing 4 contains a foreach loop that iterates through all of the movies. For each movie, the MovieTemplate partial is used to format the movie. The MovieTemplate is rendered by calling the RenderPartial() helper method.

The modified Index view renders the very same HTML table of database records. However, the view has been greatly simplified.

The RenderPartial() method is different than most of the other helper methods because it does not return a string. Therefore, you must call the RenderPartial() method using <% Html.RenderPartial(); %> instead of <%= Html.RenderPartial(); %>.

Summary

The goal of this tutorial was to explain how you can display a set of database records in an HTML table. First, you learned how to return a set of database records from a controller action by taking advantage of the Microsoft Entity Framework. Next, you learned how to use Visual Studio scaffolding to generate a view that displays a collection of items automatically. Finally, you learned how to simplify the view by taking advantage of a partial. You learned how to use a partial as a template so that you can format each database record.

« Previous Tutorial | Next Tutorial »

Comments (7)

C# Tutorials

(Switch to Visual Basic tutorials)

ASP.NET MVC Overview

- Create a Movie Database Application in 15 Minutes with ASP.NET MVC
- ASP.NET MVC Overview
- Understanding the ASP.NET MVC Execution Process
- Understanding Models, Views, and Controllers
- Creating a MVC 3 Application with Razor and Unobtrusive JavaScript

ASP.NET MVC Routing

- ASP.NET MVC Routing Overview
- Creating Custom Routes
- Creating a Route Constraint
- Creating a Custom Route Constraint

ASP.NET MVC Controllers

- ASP.NET MVC Controller Overview
- Creating a Controller
- Creating an Action

ASP.NET MVC Views

- ASP.NET MVC Views Overview
- Creating Custom HTML Helpers
- Displaying a Table of Database Data
- Using the TagBuilder Class to Build HTML Helpers

ASP.NET MVC Models

- Creating Model Classes with the Entity Framework
- Creating Model Classes with LINQ to SQL

ASP.NET MVC Validation

- Performing Simple Validation
- Validating with the IDataErrorInfo Interface
- Validating with a Service Layer
- Validation with the Data Annotation Validators

Master Pages

- Creating Page Layouts with View Master Pages
- Passing Data to View Master Pages

Action Filters and Model Binders

- Understanding Action Filters

Improving Performance with Caching

- Improving Performance with Output Caching
- Adding Dynamic Content to a Cached Page

Security

- Authenticating Users with Forms Authentication
- Authenticating Users with Windows Authentication
- Preventing JavaScript Injection Attacks
- Preventing Open Redirection Attacks

ASP.NET MVC Testing

- Creating Unit Tests for ASP.NET MVC Applications

Navigation

- Providing Website Navigation with SiteMaps

Deploying ASP.NET MVC Applications

- Using ASP.NET MVC with Different Versions of IIS

Tutorials for Contact Manager

- Iteration #1 – Create the Application
- Iteration #2 – Make the application look nice
- Iteration #3 – Add form validation
- Iteration #4 – Make the application loosely coupled
- Iteration #5 – Create unit tests
- Iteration #6 – Use test-driven development
- Iteration #7 – Add Ajax functionality

ASP.NET MVC Music Store

- Part 1: Overview and File->New Project
- Part 2: Controllers
- Part 3: Views and ViewModels
- Part 4: Models and Data Access
- Part 5: Edit Forms and Templating
- Part 6: Using Data Annotations for Model Validation
- Part 7: Membership and Authorization
- Part 8: Shopping Cart with Ajax Updates
- Part 9: Registration and Checkout
- Part 10: Final Updates to Navigation and Site Design, Conclusion

NerdDinner Tutorials

- Introducing the NerdDinner Tutorial
- Create a New ASP.NET MVC Project
- Create a Database
- Build a Model with Business Rule Validations

- Use Controllers and Views to Implement a Listing/Details UI
- Provide CRUD (Create, Read, Update, Delete) Data Form Entry Support
- Use ViewData and Implement ViewModel Classes
- Re-use UI Using Master Pages and Partials
- Implement Efficient Data Paging
- Secure Applications Using Authentication and Authorization
- Use AJAX to Deliver Dynamic Updates
- Use AJAX to Implement Mapping Scenarios
- Enable Automated Unit Testing
- NerdDinner Wrap Up

Getting Started With MVC3

- Intro to ASP.NET MVC 3
- Adding a Controller
- Adding a View
- Adding a Model
- Accessing your Model's Data from a Controller
- Examining the Edit Methods and Edit View
- Adding a New Field to the Movie Model and Table
- Adding Validation to the Model
- Improving the Details and Delete Methods

-
-
-
-