**Passing Data to View Master Pages (C#)**

* 7 minutes to read

[Download PDF](https://download.microsoft.com/download/e/f/3/ef3f2ff6-7424-48f7-bdaa-180ef64c3490/ASPNET_MVC_Tutorial_13_CS.pdf)

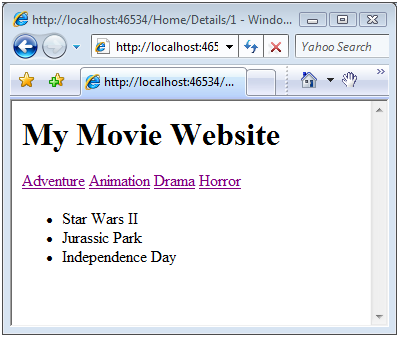
The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

**Passing Data to View Master Pages**

The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

**The Problem**

Imagine that you are building a movie database application and you want to display the list of movie categories on every page in your application (see Figure 1). Imagine, furthermore, that the list of movie categories is stored in a database table. In that case, it would make sense to retrieve the categories from the database and render the list of movie categories within a view master page.

[](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-cs/_static/image1.png)

**Figure 01**: Displaying movie categories in a view master page ([Click to view full-size image](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-cs/_static/image3.png))

Here's the problem. How do you retrieve the list of movie categories in the master page? It is tempting to call methods of your model classes in the master page directly. In other words, it is tempting to include the code for retrieving the data from the database right in your master page. However, bypassing your MVC controllers to access the database would violate the clean separation of concerns that is one of the primary benefits of building an MVC application.

In an MVC application, you want all interaction between your MVC views and your MVC model to be handled by your MVC controllers. This separation of concerns results in a more maintainable, adaptable, and testable application.

In an MVC application, all data passed to a view – including a view master page – should be passed to a view by a controller action. Furthermore, the data should be passed by taking advantage of view data. In the remainder of this tutorial, I examine two methods of passing view data to a view master page.

**The Simple Solution**

Let's start with the simplest solution to passing view data from a controller to a view master page. The simplest solution is to pass the view data for the master page in each and every controller action.

Consider the controller in Listing 1. It exposes two actions named Index() and Details(). The Index() action method returns every movie in the Movies database table. The Details() action method returns every movie in a particular movie category.

**Listing 1 – Controllers\HomeController.cs**

C#Copy

using System.Linq;

using System.Web.Mvc;

using MvcApplication1.Models;

namespace MvcApplication1.Controllers

{

     [HandleError]

     public class HomeController : Controller

     {

          private MovieDataContext \_dataContext = new MovieDataContext();

          /// <summary>

          /// Show list of all movies

          /// </summary>

          public ActionResult Index()

          {

               ViewData["categories"] = from c in \_dataContext.MovieCategories

                         select c;

               ViewData["movies"] = from m in \_dataContext.Movies

                         select m;

               return View();

          }

          /// <summary>

          /// Show list of movies in a category

          /// </summary>

          public ActionResult Details(int id)

          {

               ViewData["categories"] = from c in \_dataContext.MovieCategories

                         select c;

               ViewData["movies"] = from m in \_dataContext.Movies

                         where m.CategoryId == id

                         select m;

               return View();

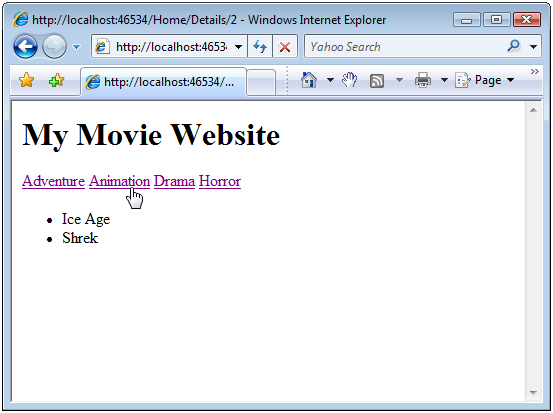
          }

     }

}

Notice that both the Index() and the Details() actions add two items to view data. The Index() action adds two keys: categories and movies. The categories key represents the list of movie categories displayed by the view master page. The movies key represents the list of movies displayed by the Index view page.

The Details() action also adds two keys named categories and movies. The categories key, once again, represents the list of movie categories displayed by the view master page. The movies key represents the list of movies in a particular category displayed by the Details view page (see Figure 2).

[](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-cs/_static/image4.png)

**Figure 02**: The Details view ([Click to view full-size image](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-cs/_static/image6.png))

The Index view is contained in Listing 2. It simply iterates through the list of movies represented by the movies item in view data.

**Listing 2 – Views\Home\Index.aspx**

ASP.NETCopy

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true" CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Views.Home.Index" %>

<%@ Import Namespace="MvcApplication1.Models" %>

<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

<ul>

<% foreach (var m in (IEnumerable<Movie>)ViewData["movies"])

     { %>

     <li><%= m.Title %></li>

<% } %>

</ul>

</asp:Content>

The view master page is contained in Listing 3. The view master page iterates and renders all of the movie categories represented by the categories item from view data.

**Listing 3 – Views\Shared\Site.master**

ASP.NETCopy

<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.Master.cs" Inherits="MvcApplication1.Views.Shared.Site" %>

<%@ Import Namespace="MvcApplication1.Models" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

     <title></title>

     <asp:ContentPlaceHolder ID="head" runat="server">

     </asp:ContentPlaceHolder>

</head>

<body>

     <div>

          <h1>My Movie Website</h1>

          <% foreach (var c in (IEnumerable<MovieCategory>)ViewData["categories"])

                           {%>

               <%= Html.ActionLink(c.Name, "Details", new {id=c.Id} ) %>

          <% } %>

          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

          </asp:ContentPlaceHolder>

     </div>

</body>

</html>

All data is passed to the view and the view master page through view data. That is the correct way to pass data to the master page.

So, what's wrong with this solution? The problem is that this solution violates the DRY (Don't Repeat Yourself) principle. Each and every controller action must add the very same list of movie categories to view data. Having duplicate code in your application makes your application much more difficult to maintain, adapt, and modify.

**The Good Solution**

In this section, we examine an alternative, and better, solution to passing data from a controller action to a view master page. Instead of adding the movie categories for the master page in each and every controller action, we add the movie categories to the view data only once. All view data used by the view master page is added in an Application controller.

The ApplicationController class is contained in Listing 4.

**Listing 4 – Controllers\ApplicationController.cs**

C#Copy

using System.Linq;

using System.Web.Mvc;

using MvcApplication1.Models;

namespace MvcApplication1.Controllers

{

     public abstract class ApplicationController : Controller

     {

          private MovieDataContext \_dataContext = new MovieDataContext();

          public MovieDataContext DataContext

          {

               get { return \_dataContext; }

          }

          public ApplicationController()

          {

               ViewData["categories"] = from c in DataContext.MovieCategories

                         select c;

          }

     }

}

There are three things that you should notice about the Application controller in Listing 4. First, notice that the class inherits from the base System.Web.Mvc.Controller class. The Application controller is a controller class.

Second, notice that the Application controller class is an abstract class. An abstract class is a class that must be implemented by a concrete class. Because the Application controller is an abstract class, you cannot not invoke any methods defined in the class directly. If you attempt to invoke the Application class directly then you'll get a Resource Cannot Be Found error message.

Third, notice that the Application controller contains a constructor that adds the list of movie categories to view data. Every controller class that inherits from the Application controller calls the Application controller's constructor automatically. Whenever you call any action on any controller that inherits from the Application controller, the movie categories is included in the view data automatically.

The Movies controller in Listing 5 inherits from the Application controller.

**Listing 5 – Controllers\MoviesController.cs**

C#Copy

using System.Linq;

using System.Web.Mvc;

namespace MvcApplication1.Controllers

{

     public class MoviesController : ApplicationController

     {

          /// <summary>

          /// Show list of all movies

          /// </summary>

          public ActionResult Index()

          {

               ViewData["movies"] = from m in DataContext.Movies

                         select m;

               return View();

          }

          /// <summary>

          /// Show list of movies in a category

          /// </summary>

          public ActionResult Details(int id)

          {

               ViewData["movies"] = from m in DataContext.Movies

                         where m.CategoryId == id

                         select m;

               return View();

          }

     }

}

The Movies controller, just like the Home controller discussed in the previous section, exposes two action methods named Index() and Details(). Notice that the list of movie categories displayed by the view master page is not added to view data in either the Index() or Details() method. Because the Movies controller inherits from the Application controller, the list of movie categories is added to view data automatically.

Notice that this solution to adding view data for a view master page does not violate the DRY (Don't Repeat Yourself) principle. The code for adding the list of movie categories to view data is contained in only one location: the constructor for the Application controller.

**Summary**

In this tutorial, we discussed two approaches to passing view data from a controller to a view master page. First, we examined a simple, but difficult to maintain approach. In the first section, we discussed how you can add view data for a view master page in each every controller action in your application. We concluded that this was a bad approach because it violates the DRY (Don't Repeat Yourself) principle.

Next, we examined a much better strategy for adding data required by a view master page to view data. Instead of adding the view data in each and every controller action, we added the view data only once within an Application controller. That way, you can avoid duplicate code when passing data to a view master page in an ASP.NET MVC application.

**ASP.NET MVC Views Overview (VB)**

* 7 minutes to read

What is an ASP.NET MVC View and how does it differ from a HTML page? In this tutorial, Stephen Walther introduces you to Views and demonstrates how you can take advantage of View Data and HTML Helpers within a View.

The purpose of this tutorial is to provide you with a brief introduction to ASP.NET MVC views, view data, and HTML Helpers. By the end of this tutorial, you should understand how to create new views, pass data from a controller to a view, and use HTML Helpers to generate content in a view.

**Understanding Views**

Unlike ASP.NET or Active Server Pages, ASP.NET MVC does not include anything that directly corresponds to a page. In an ASP.NET MVC application, there is not a page on disk that corresponds to the path in the URL that you type into the address bar of your browser. The closest thing to a page in an ASP.NET MVC application is something called a *view*.

In an ASP.NET MVC application, incoming browser requests are mapped to controller actions. A controller action might return a view. However, a controller action might perform some other type of action such as redirecting you to another controller action.

Listing 1 contains a simple controller named the HomeController. The HomeController exposes two controller actions named Index() and Details().

**Listing 1 - HomeController.vb**

VBCopy

<HandleError()> \_

Public Class HomeController

Inherits System.Web.Mvc.Controller

Function Index()

Return View()

End Function

Function Details()

Return RedirectToAction("Index")

End Function

End Class

You can invoke the first action, the Index() action, by typing the following URL into your browser address bar:

/Home/Index

You can invoke the second action, the Details() action, by typing this address into your browser:

/Home/Details

The Index() action returns a view. Most actions that you create will return views. However, an action can return other types of action results. For example, the Details() action returns a RedirectToActionResult that redirects incoming request to the Index() action.

The Index() action contains the following single line of code:

View()

This line of code returns a view that must be located at the following path on your web server:

\Views\Home\Index.aspx

The path to the view is inferred from the name of the controller and the name of the controller action.

If you prefer, you can be explicit about the view. The following line of code returns a view named Fred :

View( Fred )

When this line of code is executed, a view is returned from the following path:

\Views\Home\Fred.aspx

**Note**

If you plan to create unit tests for your ASP.NET MVC application then it is a good idea to be explicit about view names. That way, you can create a unit test to verify that the expected view was returned by a controller action.

**Adding Content to a View**

A view is a standard (X)HTML document that can contain scripts. You use scripts to add dynamic content to a view.

For example, the view in Listing 2 displays the current date and time.

**Listing 2 - \Views\Home\Index.aspx**

ASP.NETCopy

<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

<title>Index</title>

</head>

<body>

<div>

The current date and time is

<% Response.Write(DateTime.Now)%>

</div>

</body>

</html>

Notice that the body of the HTML page in Listing 2 contains the following script:

<% Response.Write(DateTime.Now)%>

You use the script delimiters <% and %> to mark the beginning and end of a script. This script is written in Visual basic. It displays the current date and time by calling the Response.Write() method to render content to the browser. The script delimiters <% and %> can be used to execute one or more statements.

Since you call Response.Write() so often, Microsoft provides you with a shortcut for calling the Response.Write() method. The view in Listing 3 uses the delimiters <%= and %> as a shortcut for calling Response.Write().

**Listing 3 - Views\Home\Index2.aspx**

ASP.NETCopy

<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

<title>Index</title>

</head>

<body>

<div>

The current date and time is

<%= DateTime.Now %>

</div>

</body>

</html>

You can use any .NET language to generate dynamic content in a view. Normally, you'll use either Visual Basic .NET or C# to write your controllers and views.

**Using HTML Helpers to Generate View Content**

To make it easier to add content to a view, you can take advantage of something called an *HTML Helper*. An HTML Helper, typically, is a method that generates a string. You can use HTML Helpers to generate standard HTML elements such as textboxes, links, dropdown lists, and list boxes.

For example, the view in Listing 4 takes advantage of three HTML Helpers -- the BeginForm(), the TextBox() and Password() helpers -- to generate a Login form (see Figure 1).

**Listing 4 -- \Views\Home\Login.aspx**

ASP.NETCopy

<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

<title>Login Form</title>

</head>

<body>

<div>

<% Using Html.BeginForm()%>

<label for="UserName">User Name:</label>

<br />

<%= Html.TextBox("UserName") %>

<br /><br />

<label for="Password">Password:</label>

<br />

<%= Html.Password("Password") %>

<br /><br />

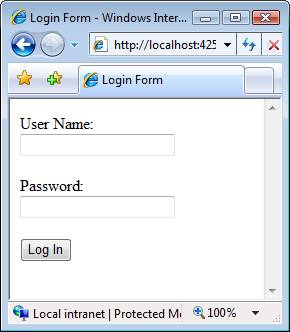
<input type="submit" value="Log in" />

<% End Using%>

</div>

</body>

</html>

[](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/asp-net-mvc-views-overview-vb/_static/image1.png)

**Figure 01**: A standard Login form ([Click to view full-size image](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/asp-net-mvc-views-overview-vb/_static/image2.png))

All of the HTML Helpers methods are called on the Html property of the view. For example, you render a TextBox by calling the Html.TextBox() method.

Notice that you use the script delimiters <%= and %> when calling both the Html.TextBox() and Html.Password() helpers. These helpers simply return a string. You need to call Response.Write() in order to render the string to the browser.

Using HTML Helper methods is optional. They make your life easier by reducing the amount of HTML and script that you need to write. The view in Listing 5 renders the exact same form as the view in Listing 4 without using HTML Helpers.

**Listing 5 -- \Views\Home\Login.aspx**

ASP.NETCopy

<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

<title>Login Form</title>

</head>

<body>

<div>

<form method="post" action="/Home/Login">

<label for="userName">User Name:</label>

<br />

<input name="userName" />

<br /><br />

<label for="password">Password:</label>

<br />

<input name="password" type="password" />

<br /><br />

<input type="submit" value="Log In" />

</form>

</div>

</body>

</html>

You also have the option of creating your own HTML Helpers. For example, you can create a GridView() helper method that displays a set of database records in an HTML table automatically. We explore this topic in the tutorial **Creating Custom HTML Helpers**.

**Using View Data to Pass Data to a View**

You use view data to pass data from a controller to a view. Think of view data like a package that you send through the mail. All data passed from a controller to a view must be sent using this package. For example, the controller in Listing 6 adds a message to view data.

**Listing 6 - ProductController.vb**

VBCopy

Public Class ProductController

Inherits System.Web.Mvc.Controller

Function Index()

ViewData("message") = "Hello World!"

Return View()

End Function

End Class

The controller ViewData property represents a collection of name and value pairs. In Listing 6, the Index() method adds an item to the view data collection named message with the value Hello World!. When the view is returned by the Index() method, the view data is passed to the view automatically.

The view in Listing 7 retrieves the message from the view data and renders the message to the browser.

**Listing 7 -- \Views\Product\Index.aspx**

ASP.NETCopy

<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

<title>Product Index</title>

</head>

<body>

<div>

<%=Html.Encode(ViewData("message"))%>

</div>

</body>

</html>

Notice that the view takes advantage of the Html.Encode() HTML Helper method when rendering the message. The Html.Encode() HTML Helper encodes special characters such as < and > into characters that are safe to display in a web page. Whenever you render content that a user submits to a website, you should encode the content to prevent JavaScript injection attacks.

(Because we created the message ourselves in the ProductController, we don't really need to encode the message. However, it is a good habit to always call the Html.Encode() method when displaying content retrieved from view data within a view.)

In Listing 7, we took advantage of view data to pass a simple string message from a controller to a view. You also can use view data to pass other types of data, such as a collection of database records, from a controller to a view. For example, if you want to display the contents of the Products database table in a view, then you would pass the collection of database records in view data.

You also have the option of passing strongly typed view data from a controller to a view. We explore this topic in the tutorial **Understanding Strongly Typed View Data and Views**.

**Summary**

This tutorial provided a brief introduction to ASP.NET MVC views, view data, and HTML Helpers. In the first section, you learned how to add new views to your project. You learned that you must add a view to the right folder in order to call it from a particular controller. Next, we discussed the topic of HTML Helpers. You learned how HTML Helpers enable you to easily generate standard HTML content. Finally, you learned how to take advantage of view data to pass data from a controller to a view.

**Passing Data to View Master Pages (VB)**

* 7 minutes to read

[Download PDF](https://download.microsoft.com/download/e/f/3/ef3f2ff6-7424-48f7-bdaa-180ef64c3490/ASPNET_MVC_Tutorial_13_VB.pdf)

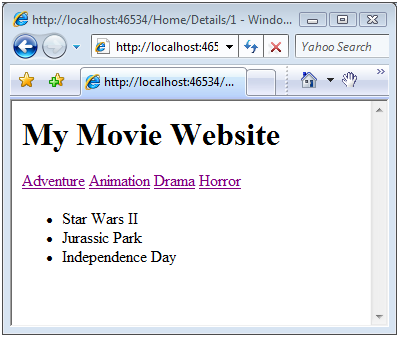
The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

**Passing Data to View Master Pages**

The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

**The Problem**

Imagine that you are building a movie database application and you want to display the list of movie categories on every page in your application (see Figure 1). Imagine, furthermore, that the list of movie categories is stored in a database table. In that case, it would make sense to retrieve the categories from the database and render the list of movie categories within a view master page.

[](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-vb/_static/image1.png)

**Figure 01**: Displaying movie categories in a view master page ([Click to view full-size image](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-vb/_static/image3.png))

Here's the problem. How do you retrieve the list of movie categories in the master page? It is tempting to call methods of your model classes in the master page directly. In other words, it is tempting to include the code for retrieving the data from the database right in your master page. However, bypassing your MVC controllers to access the database would violate the clean separation of concerns that is one of the primary benefits of building an MVC application.

In an MVC application, you want all interaction between your MVC views and your MVC model to be handled by your MVC controllers. This separation of concerns results in a more maintainable, adaptable, and testable application.

In an MVC application, all data passed to a view – including a view master page – should be passed to a view by a controller action. Furthermore, the data should be passed by taking advantage of view data. In the remainder of this tutorial, I examine two methods of passing view data to a view master page.

**The Simple Solution**

Let's start with the simplest solution to passing view data from a controller to a view master page. The simplest solution is to pass the view data for the master page in each and every controller action.

Consider the controller in Listing 1. It exposes two actions named Index() and Details(). The Index() action method returns every movie in the Movies database table. The Details() action method returns every movie in a particular movie category.

**Listing 1 – Controllers\HomeController.vb**

VBCopy

<HandleError()> \_

Public Class HomeController

     Inherits System.Web.Mvc.Controller

     Private \_dataContext As New MovieDataContext()

     ''' <summary>

     ''' Show list of all movies

     ''' </summary>

     Function Index()

          ViewData("categories") = From c In \_dataContext.MovieCategories \_

                    Select c

          ViewData("movies") = From m In \_dataContext.Movies \_

                    Select m

          Return View()

     End Function

     ''' <summary>

     ''' Show list of movies in a category

     ''' </summary>

     Function Details(ByVal id As Integer)

          ViewData("categories") = From c In \_dataContext.MovieCategories \_

                    Select c

          ViewData("movies") = From m In \_dataContext.Movies \_

                    Where m.CategoryId = id \_

                    Select m

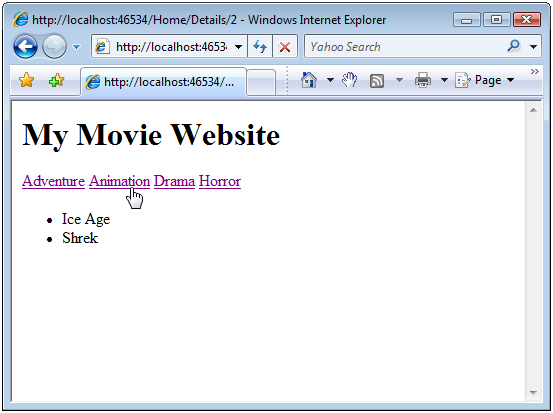
          Return View()

     End Function

End Class

Notice that both the Index() and the Details() actions add two items to view data. The Index() action adds two keys: categories and movies. The categories key represents the list of movie categories displayed by the view master page. The movies key represents the list of movies displayed by the Index view page.

The Details() action also adds two keys named categories and movies. The categories key, once again, represents the list of movie categories displayed by the view master page. The movies key represents the list of movies in a particular category displayed by the Details view page (see Figure 2).

[](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-vb/_static/image4.png)

**Figure 02**: The Details view ([Click to view full-size image](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/passing-data-to-view-master-pages-vb/_static/image6.png))

The Index view is contained in Listing 2. It simply iterates through the list of movies represented by the movies item in view data.

**Listing 2 – Views\Home\Index.aspx**

ASP.NETCopy

<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false" CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index" %>

<%@ Import Namespace="MvcApplication1" %>

<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

<ul>

<% For Each m In ViewData("movies")%>

     <li><%= m.Title %></li>

<% Next%>

</ul>

</asp:Content>

The view master page is contained in Listing 3. The view master page iterates and renders all of the movie categories represented by the categories item from view data.

**Listing 3 – Views\Shared\Site.master**

ASP.NETCopy

<%@ Master Language="VB" AutoEventWireup="false" CodeBehind="Site.Master.vb" Inherits="MvcApplication1.Site" %>

<%@ Import Namespace="MvcApplication1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head id="Head1" runat="server">

     <title></title>

     <asp:ContentPlaceHolder ID="head" runat="server">

     </asp:ContentPlaceHolder>

</head>

<body>

     <div>

          <h1>My Movie Website</h1>

          <% For Each c In ViewData("categories")%>

               <%=Html.ActionLink(c.Name, "Details", New With {.id = c.Id})%>

          <% Next%>

          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

          </asp:ContentPlaceHolder>

     </div>

</body>

</html>

All data is passed to the view and the view master page through view data. That is the correct way to pass data to the master page.

So, what's wrong with this solution? The problem is that this solution violates the DRY (Don't Repeat Yourself) principle. Each and every controller action must add the very same list of movie categories to view data. Having duplicate code in your application makes your application much more difficult to maintain, adapt, and modify.

**The Good Solution**

In this section, we examine an alternative, and better, solution to passing data from a controller action to a view master page. Instead of adding the movie categories for the master page in each and every controller action, we add the movie categories to the view data only once. All view data used by the view master page is added in an Application controller.

The ApplicationController class is contained in Listing 4.

The ApplicationController class is contained in Listing 4.

**Listing 4 – Controllers\ApplicationController.vb**

VBCopy

Public MustInherit Class ApplicationController

     Inherits System.Web.Mvc.Controller

     Private \_dataContext As New MovieDataContext()

     Public ReadOnly Property DataContext() As MovieDataContext

          Get

               Return \_dataContext

          End Get

     End Property

     Sub New()

          ViewData("categories") = From c In DataContext.MovieCategories \_

                    Select c

     End Sub

End Class

There are three things that you should notice about the Application controller in Listing 4. First, notice that the class inherits from the base System.Web.Mvc.Controller class. The Application controller is a controller class.

Second, notice that the Application controller class is a MustInherit class. An MustInherit class is a class that must be implemented by a concrete class. Because the Application controller is an MustInherit class, you cannot not invoke any methods defined in the class directly. If you attempt to invoke the Application class directly then you'll get a Resource Cannot Be Found error message.

Third, notice that the Application controller contains a constructor that adds the list of movie categories to view data. Every controller class that inherits from the Application controller calls the Application controller's constructor automatically. Whenever you call any action on any controller that inherits from the Application controller, the movie categories is included in the view data automatically.

The Movies controller in Listing 5 inherits from the Application controller.

**Listing 5 – Controllers\MoviesController.vb**

VBCopy

<HandleError()> \_

Public Class MoviesController

     Inherits ApplicationController

     ''' <summary>

     ''' Show list of all movies

     ''' </summary>

     Function Index()

          ViewData("movies") = From m In DataContext.Movies \_

                    Select m

          Return View()

      End Function

     ''' <summary>

     ''' Show list of movies in a category

     ''' </summary>

     Function Details(ByVal id As Integer)

          ViewData("movies") = From m In DataContext.Movies \_

                    Where m.CategoryId = id \_

                    Select m

          Return View()

     End Function

End Class

The Movies controller, just like the Home controller discussed in the previous section, exposes two action methods named Index() and Details(). Notice that the list of movie categories displayed by the view master page is not added to view data in either the Index() or Details() method. Because the Movies controller inherits from the Application controller, the list of movie categories is added to view data automatically.

Notice that this solution to adding view data for a view master page does not violate the DRY (Don't Repeat Yourself) principle. The code for adding the list of movie categories to view data is contained in only one location: the constructor for the Application controller.

**Summary**

In this tutorial, we discussed two approaches to passing view data from a controller to a view master page. First, we examined a simple, but difficult to maintain approach. In the first section, we discussed how you can add view data for a view master page in each every controller action in your application. We concluded that this was a bad approach because it violates the DRY (Don't Repeat Yourself) principle.

Next, we examined a much better strategy for adding data required by a view master page to view data. Instead of adding the view data in each and every controller action, we added the view data only once within an Application controller. That way, you can avoid duplicate code when passing data to a view master page in an ASP.NET MVC application.