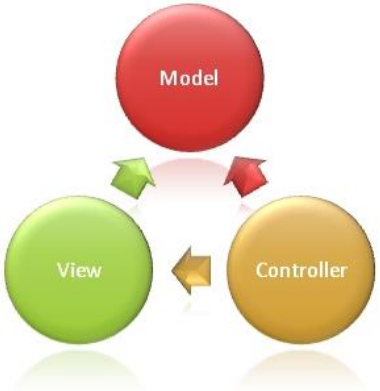
ASP.NET MVC

Model- View- Controller

What is MVC



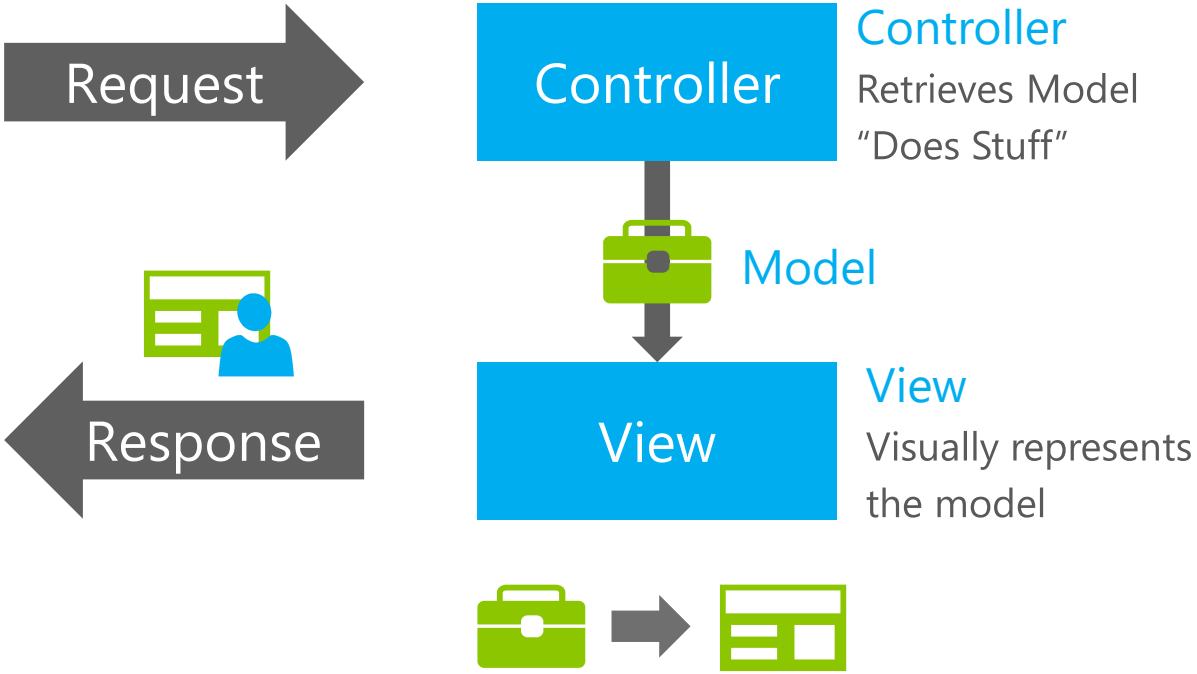
- MVC stands for *model-view-controller*
- MVC is a **pattern** for developing applications that are well architected, testable and easy to maintain.
- **M**odels: Classes that **represent** the data of the application and that use validation logic to enforce business rules for that data.
- **V**iews: Template files that your application uses to dynamically generate **HTML responses**.
- **C**ontrollers: Classes that handle **incoming** browser **requests**, retrieve model data, and then specify view templates that return a **response** to the browser.

The MVC Programming Model

- **The Model** is the part of the application that handles the logic for the application data.
  Often model objects retrieve data (and store data) from a database.
- **The View** is the parts of the application that handles the display of the data.
  Most often the views are created from the model data.
- **The Controller** is the part of the application that handles user interaction.
  Typically controllers read data from a view, control user input, and send input data to the model.

Models, Views, and Controllers

What does MVC look like?



Seems complicated. What's the point?

- Every web application needs some **structure**
- MVC helps you **stay organized**, start to finish
- Often end up with **less code**, not more
- **Smoother learning curve** as your project grows

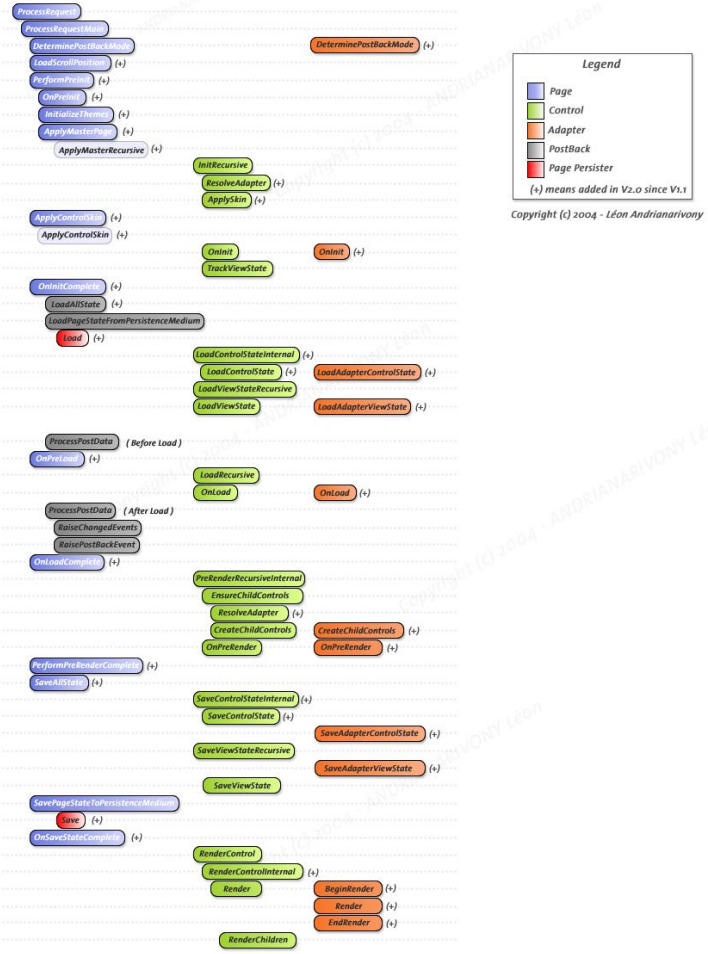
Some comparisons to ASP.NET Web Forms

ASP.NET Web Forms Values

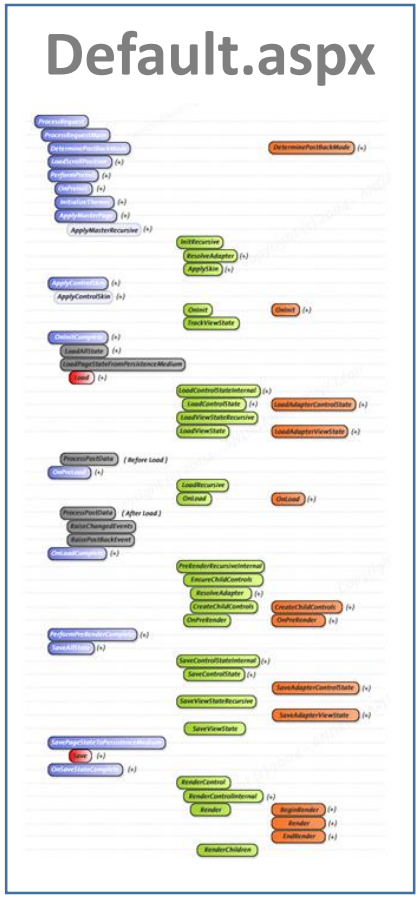
- Productive way to build web applications
- Control and event-based programming model
- Controls that abstract HTML, JS and CSS
- Rich UI controls – datagrids, charts, Ajax
- Browser differences are handled for you

Summary: Web Forms handles a lot of things for you.

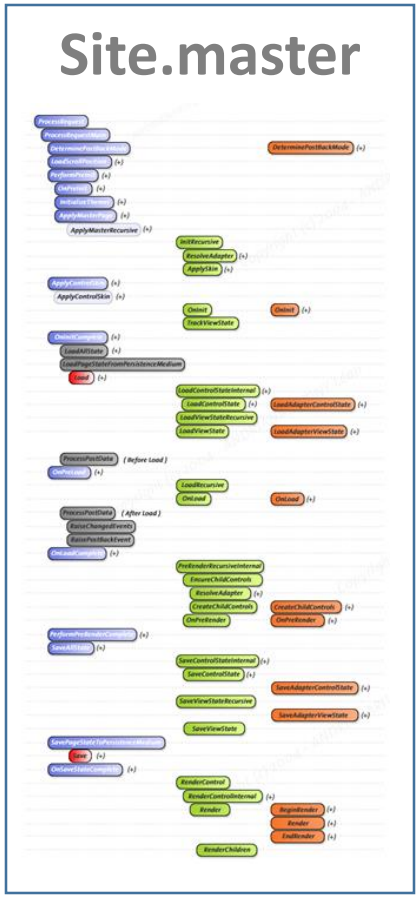
ASP.NET Page LifeCycle

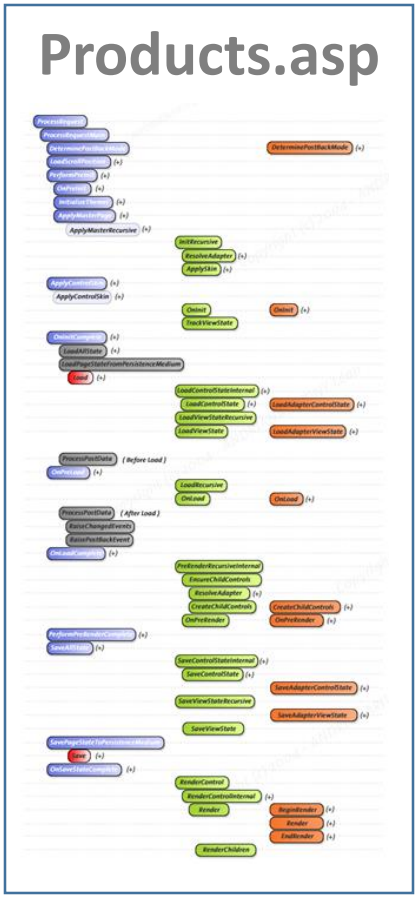


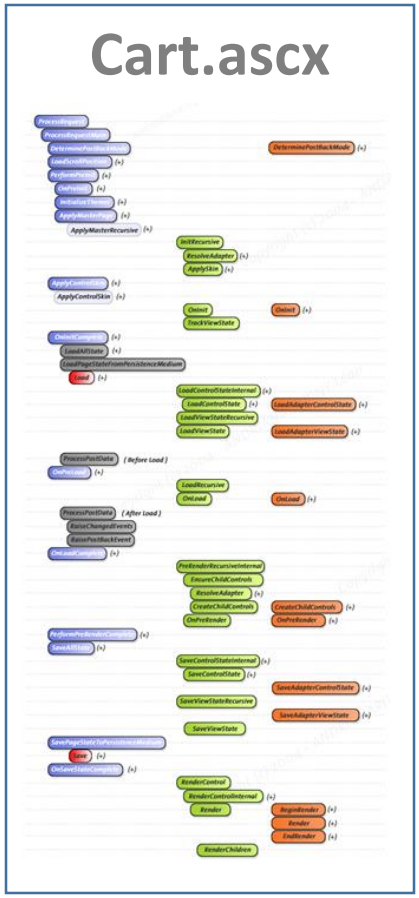
A Single Web Forms Page



A Single Web Forms Page with a Master Page and a User Control

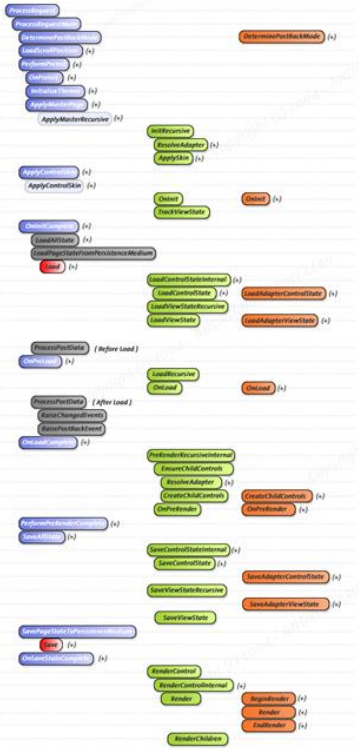




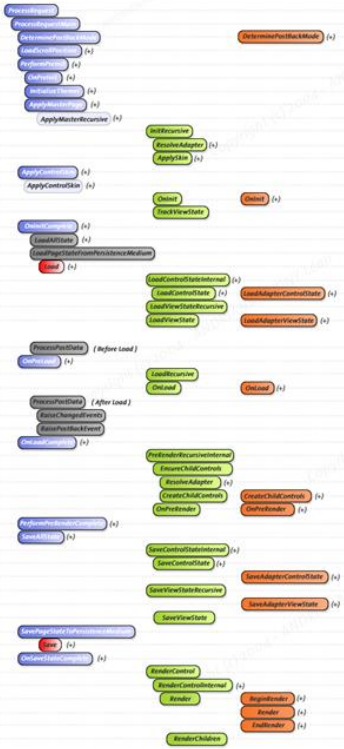


Multiple Web Forms Pages

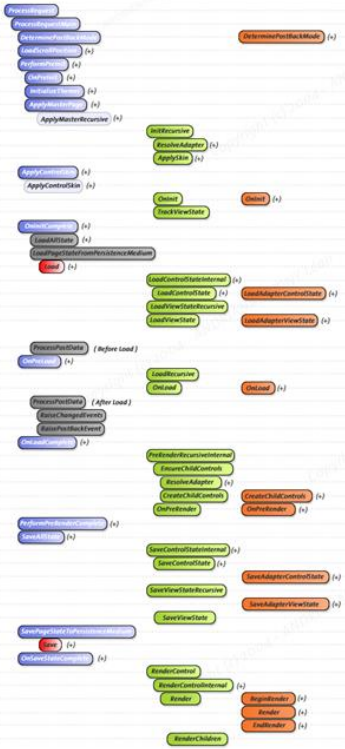
Default.aspx



Products.asp



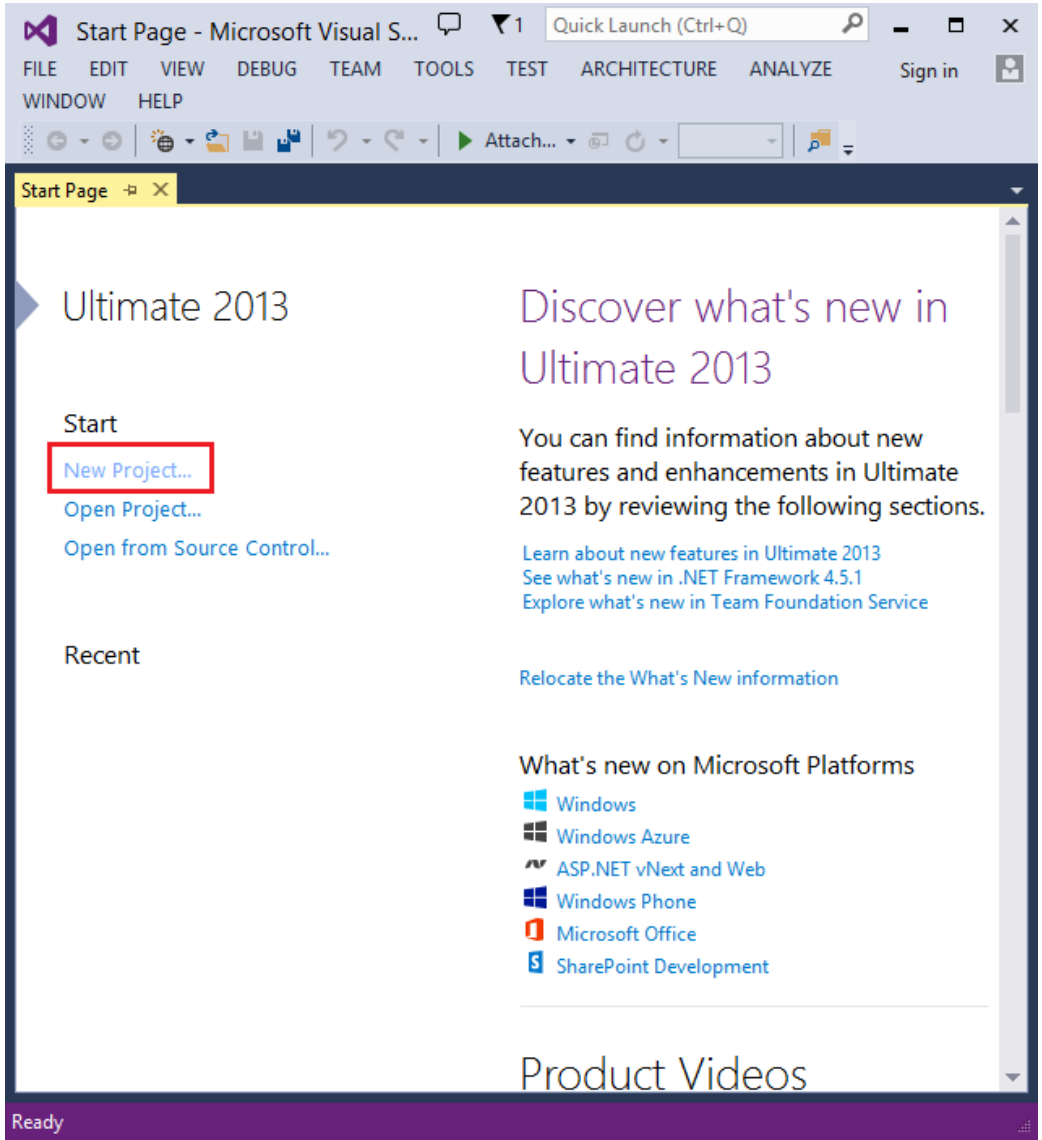
About.aspx



Testability

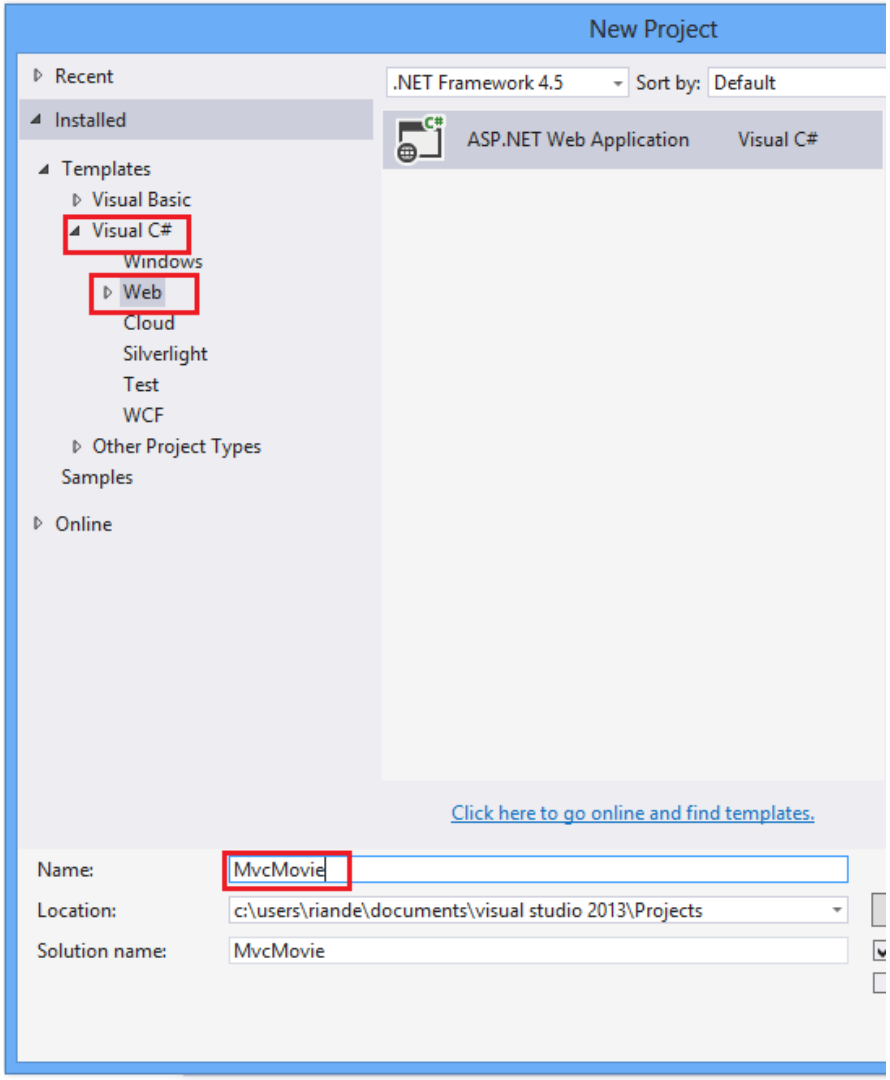
- Unit testing helps you change code with confidence
- ASP.NET MVC is designed to make unit testing easy

ASP.NET MVC DEMO

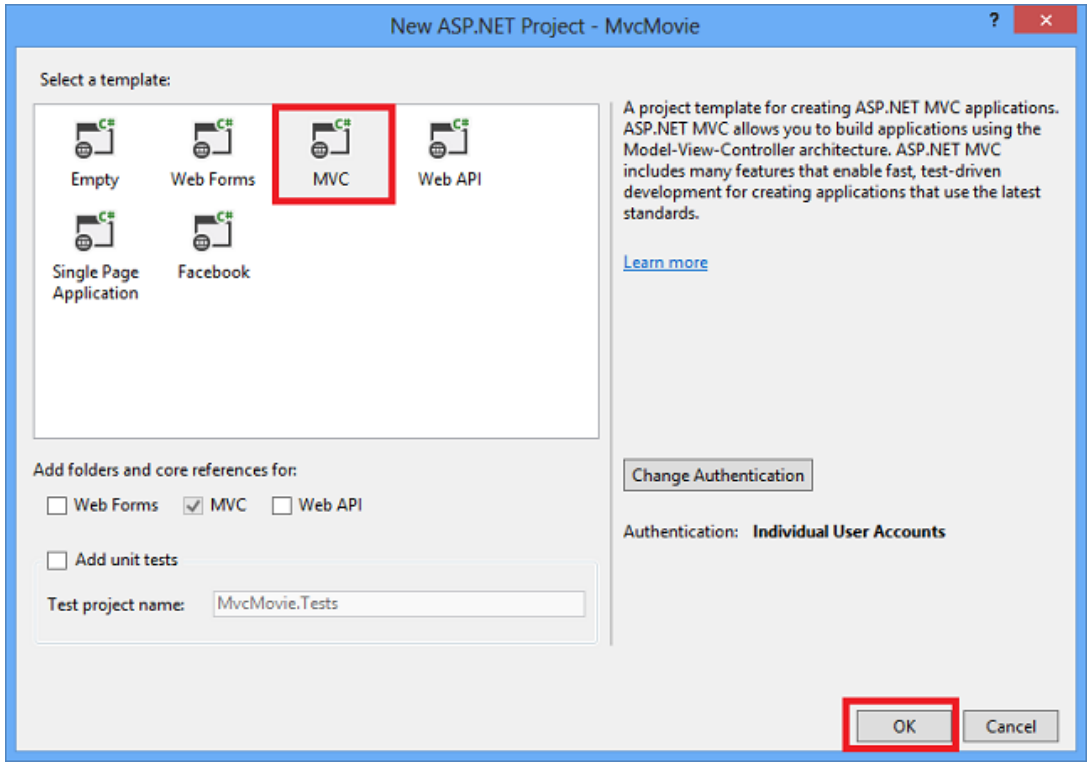


Creating Your First Application

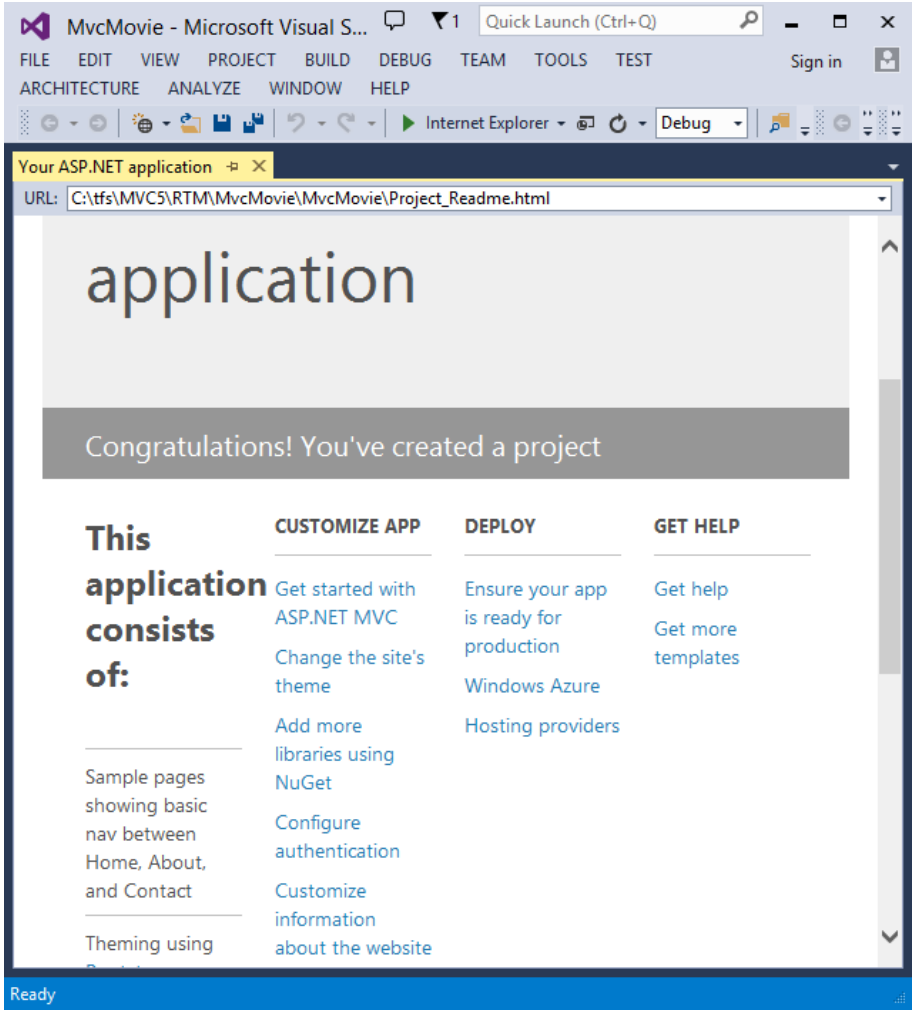
- Click **New Project**, then select Visual C# on the left, then **Web** and then select **ASP.NET Web Application**. Name your project "MvcMovie" and then click **OK**.



- In the **New ASP.NET Project** dialog , click **MVC** and then click **OK**.

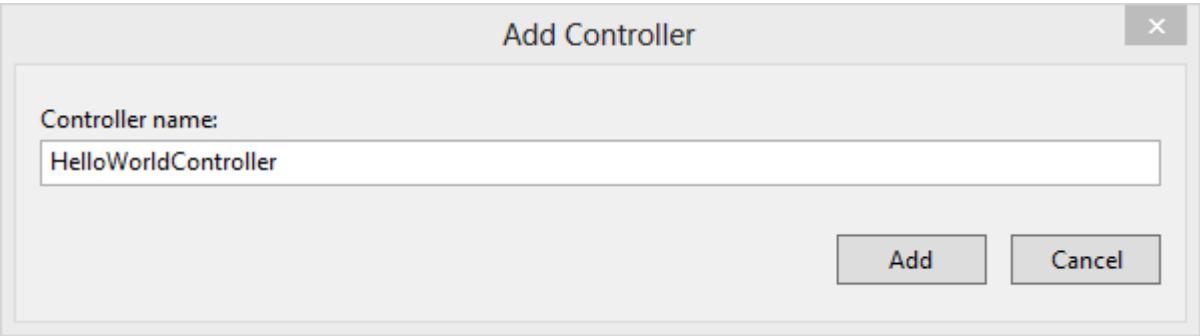


- Visual Studio used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.



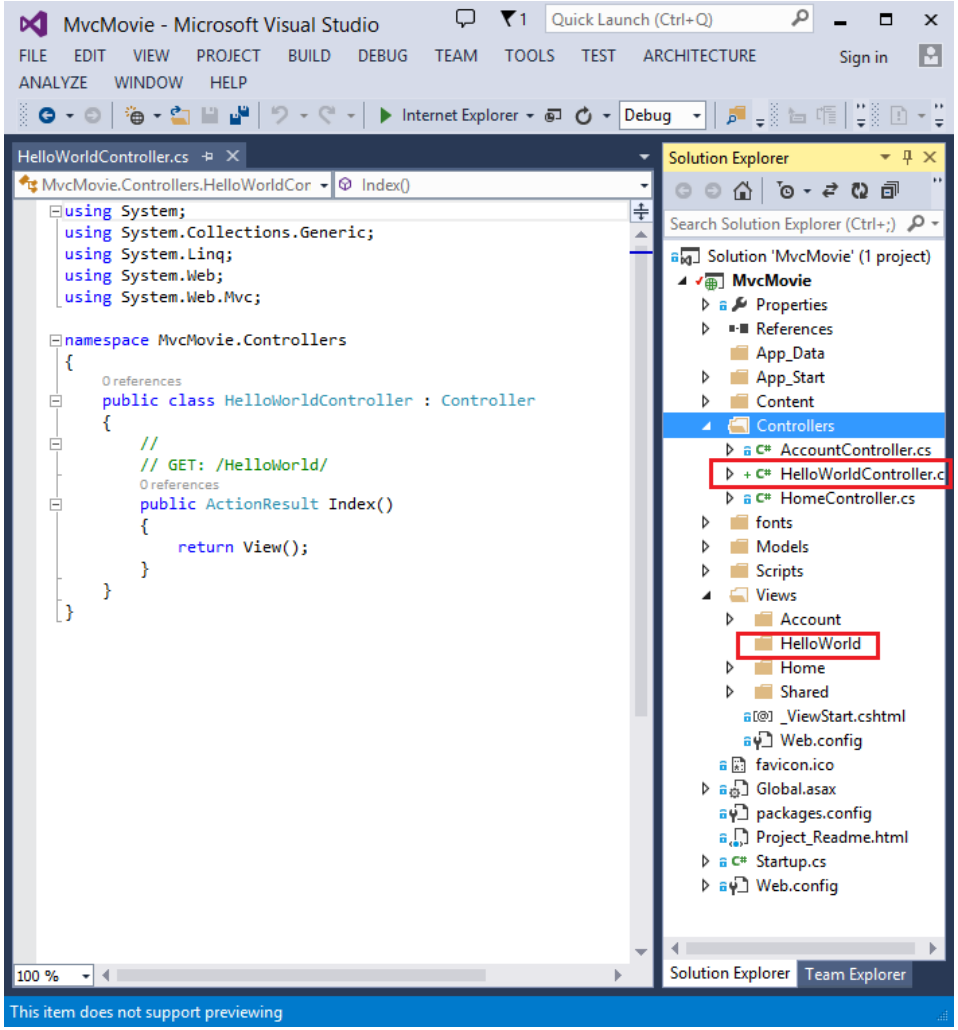
Adding a Controller

- Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then click **Add**, then **Controller**.
- In the **Add Scaffold** dialog box, click **MVC 5 Controller - Empty**, and then click **Add**.
- Name your new controller "HelloWorldController" and click **Add**.



Adding a Controller

Notice
in **Solution Explorer** that a new file has been created named *HelloWorldController.cs* and a new folder *Views\HelloWorld*. The controller is open in the IDE.



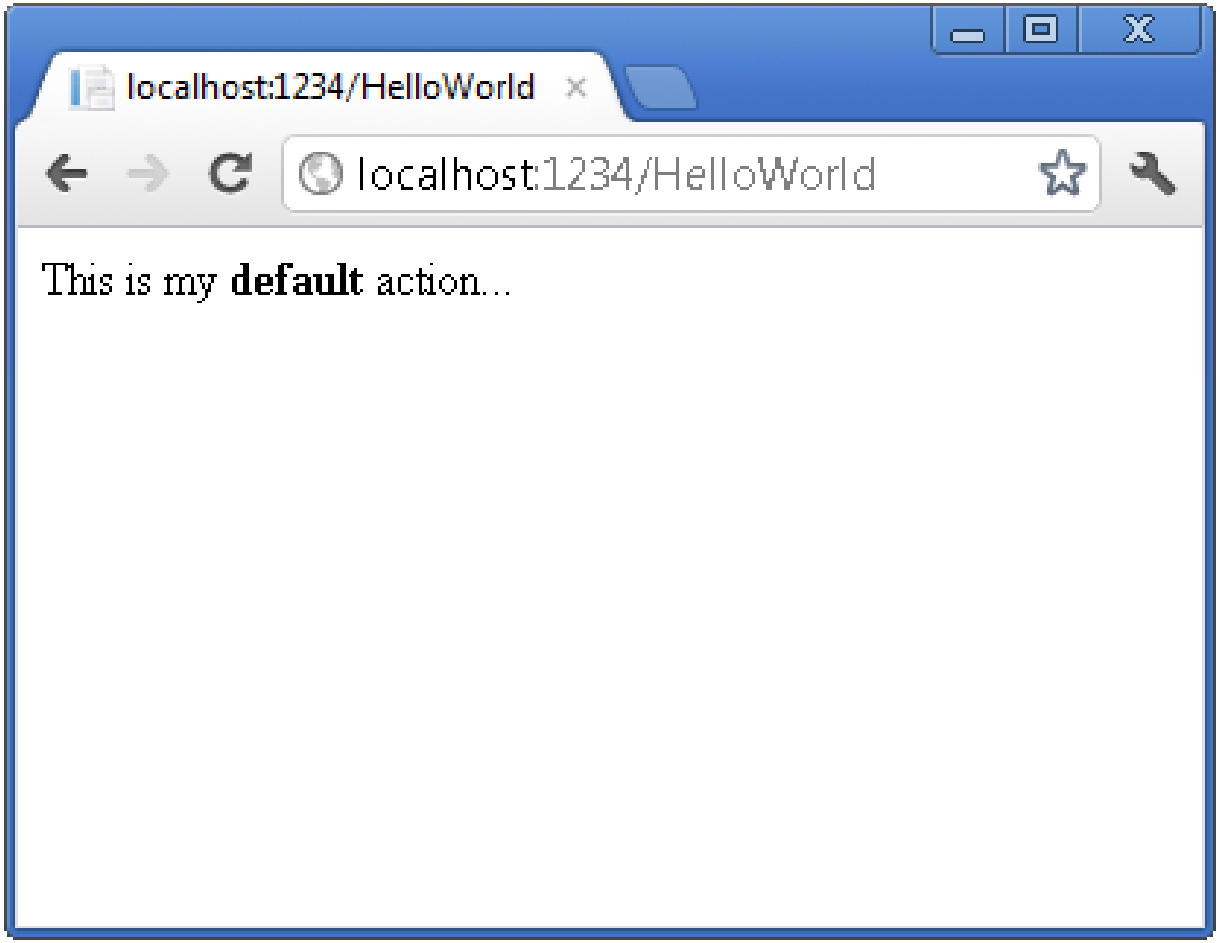
Adding a Controller

```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my <b>default</b> action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

HelloWorldController.cs

Replace the contents of the file with this code.

Adding a Controller



Adding a Controller

- ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default URL routing logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

- /[Controller]/[ActionName]/[Parameters]

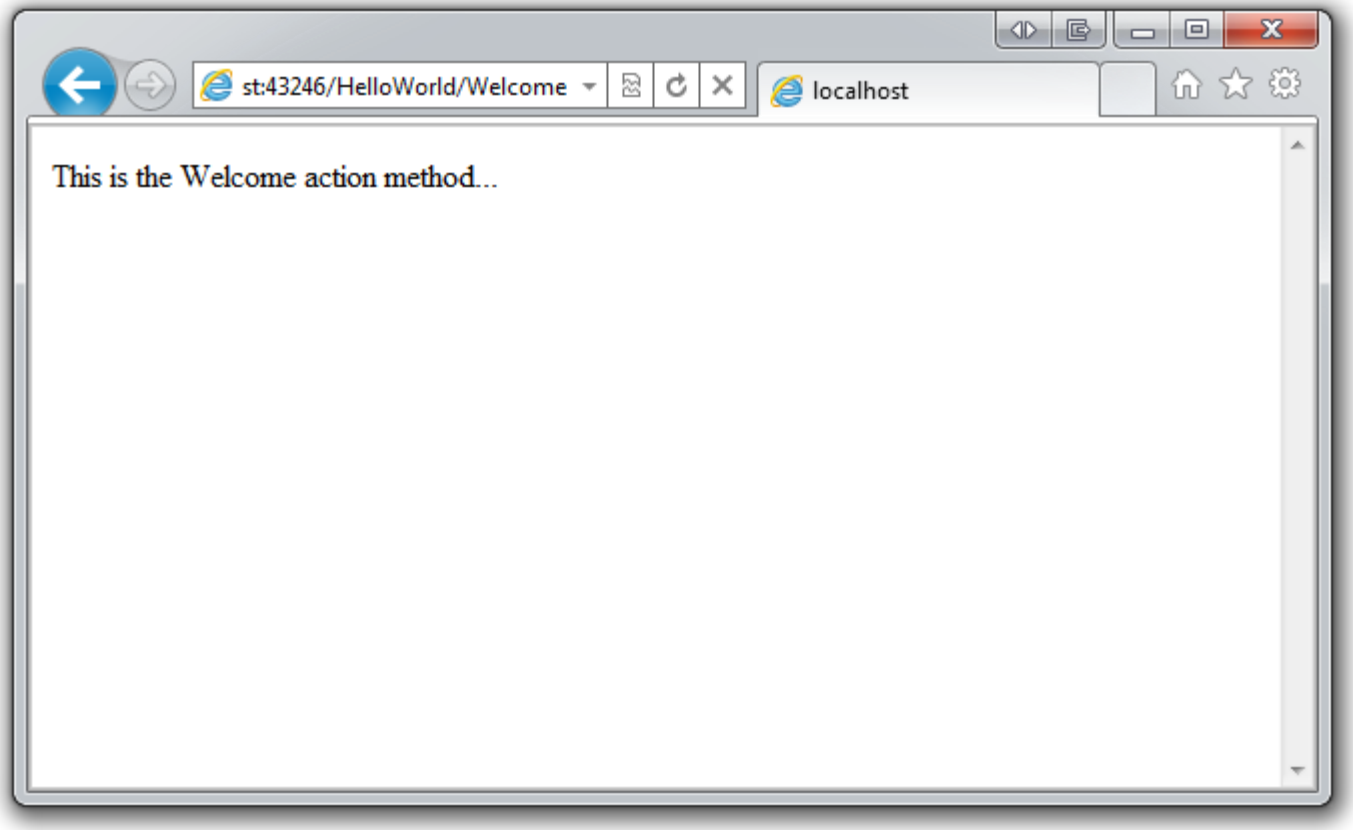
Adding a Controller

- You set the format for routing in the *App_Start/RouteConfig.cs* file.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

Adding a Controller

Browse
to *http://localhost:xxxx/HelloWorld/Welcome*



Adding a Controller

- Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, /HelloWorld/Welcome?name=Scott&numtimes=4).

```
public string Welcome(string name, int numTimes = 1) {
    return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```

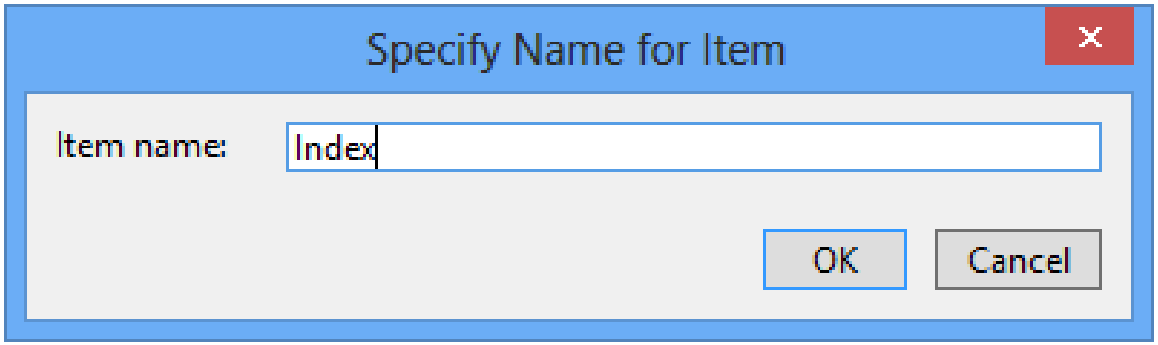
Adding a View

- Currently the Index method returns a string with a message that is hard-coded in the controller class. Change the Index method to return a View object, as shown in the following code:

```
public ActionResult Index()
{
    return View();
}
```

Adding a View

- Right click the *Views\HelloWorld* folder and click **Add**, then click **MVC 5 View Page with Layout (Razor)**.
- In the **Specify Name for Item** dialog box, enter *Index*, and then click **OK**.
- In the **Select a Layout Page** dialog, accept the default **_Layout.cshtml** and click **OK**.



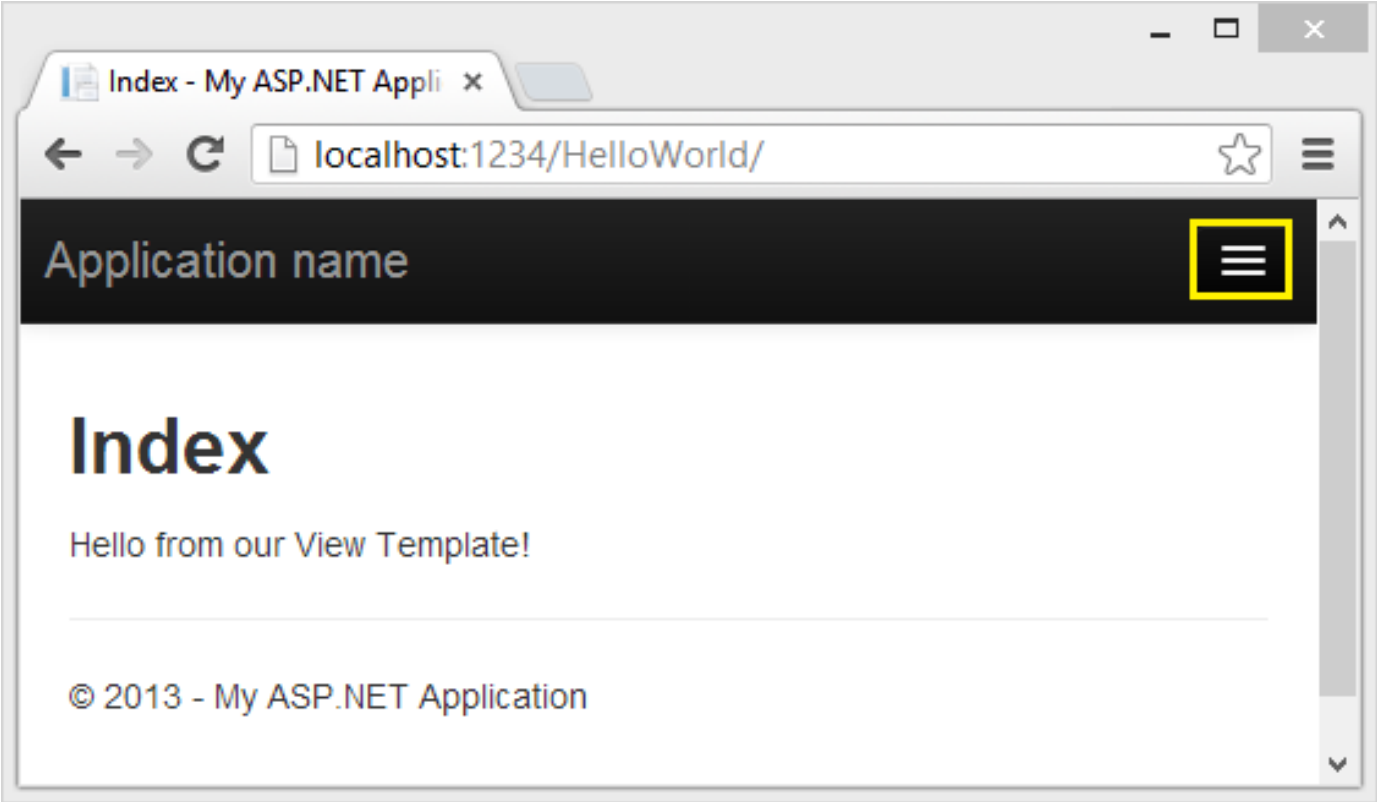
Adding a View

- Add the following highlighed markup.

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Adding a View

- Right click the *Index.cshtml* file and select **View in Browser**.



Changing Views and Layout Pages

- Go to the */Views/Shared* folder in **Solution Explorer**and open the _*Layout.cshtml* file.
- Find the @RenderBody() line. RenderBody is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. For example, if you select the About link, the Views\Home\About.cshtml view is rendered inside the RenderBody method.

Changing Views and Layout Pages

- Change the contents of the title element.
- Change the ActionLink in the layout template from "Application name" to "MVC Movie" and the controller from Home to Movies.

```
<meta charset="utf-8" />
<meta name="viewport" content="width=device-
<title>@ViewBag.Title - Movie App</title>
@Styles.Render("~/Content/css")
@Scripts.Render("~/bundles/modernizr")
```

```
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
    </button>
    @Html.ActionLink("MVC Movie", "Index", "Movies", null, new { @class = "navbar-brand" })
</div>
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li>@Html.ActionLink("Home", "Index", "Home")</li>
        <li>@Html.ActionLink("About", "About", "Home")</li>
```

Adding a Model

- You'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes.
- The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes.

Adding a Model

- In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.
- Enter the *class* name "Movie".

Adding a Model

- Add the following five properties to the Movie class:

```
using System;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

Adding a Model

- In the same file, add the following MovieDBContext class:

```
using System;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

Creating a Connection String and Working with SQL Server LocalDB

- Add the following connection string to the <connectionStrings> element in the Web.config file.
- The name of the connection string must match the name of the DbContext class.

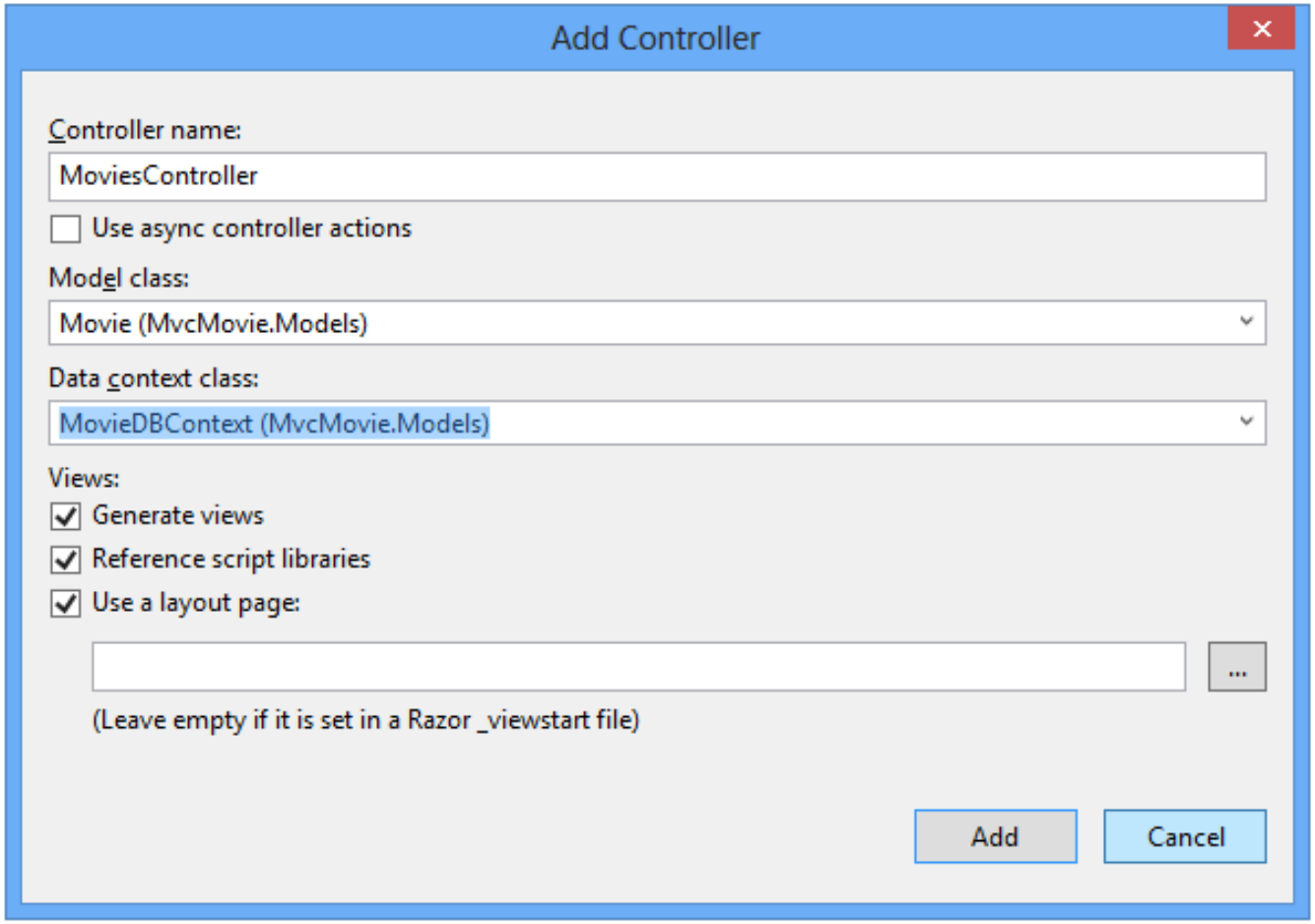
```
<add name="MovieDBContext"
connectionString="Data Source=(LocalDB)\v11.0;
AttachDbFilename=|DataDirectory|\Movies.mdf;
Integrated Security=True"
providerName="System.Data.SqlClient" />
```

Accessing Your Model's Data from a Controller

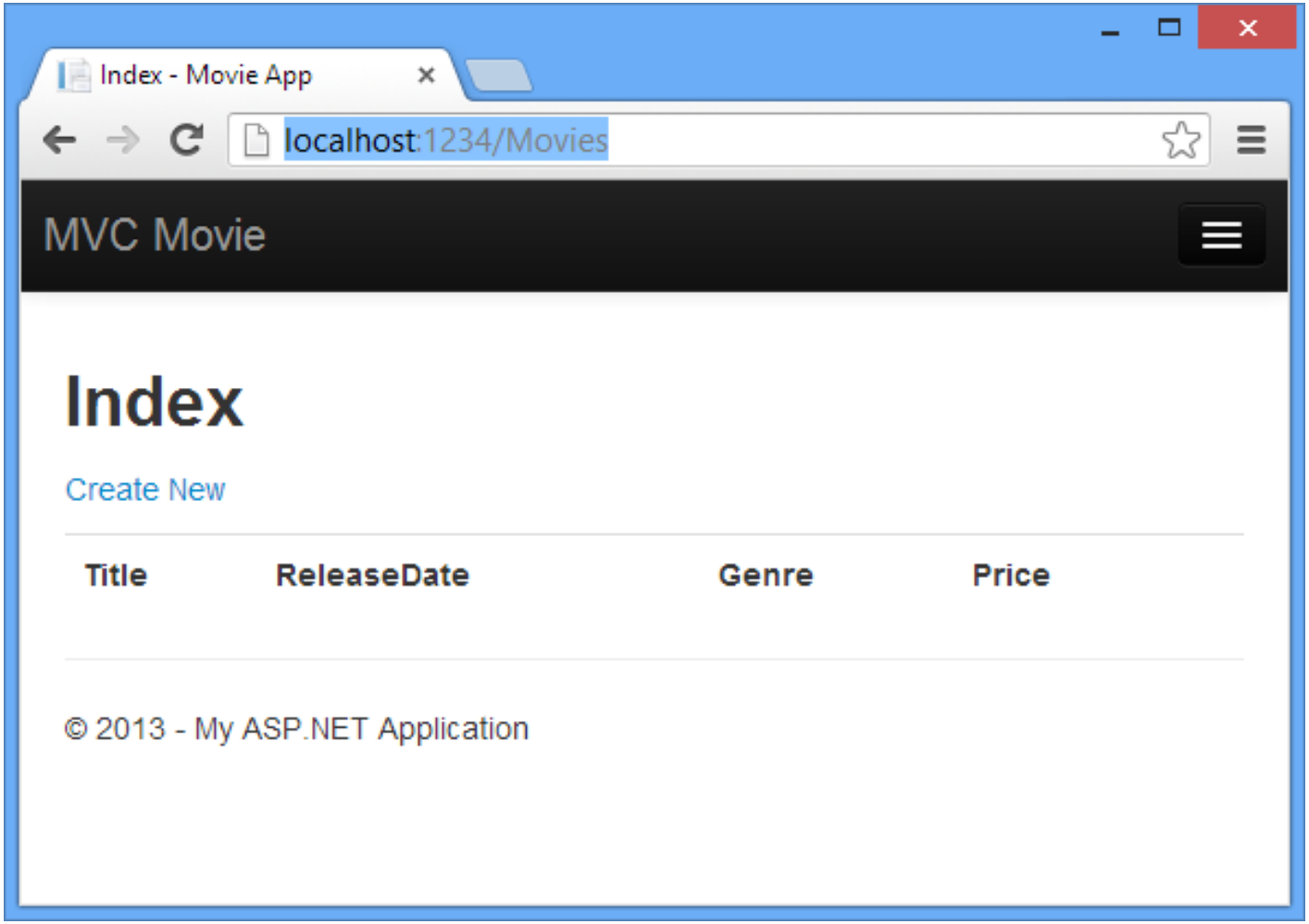
- **Build the application** before going on to the next step. If you don't build the application, you'll get an error adding a controller.
- In Solution Explorer, right-click the *Controllers* folder and then click **Add**, then **Controller**.
- In the **Add Scaffold** dialog box, click **MVC 5 Controller with views, using Entity Framework**, and then click **Add**.
- For the Controller name enter **MoviesController**.
- Select **Movie (MvcMovie.Models)** for the Model class.
- Select **MovieDBContext (MvcMovie.Models)** for the Data context class.

Accessing Your Model's Data from a Controller

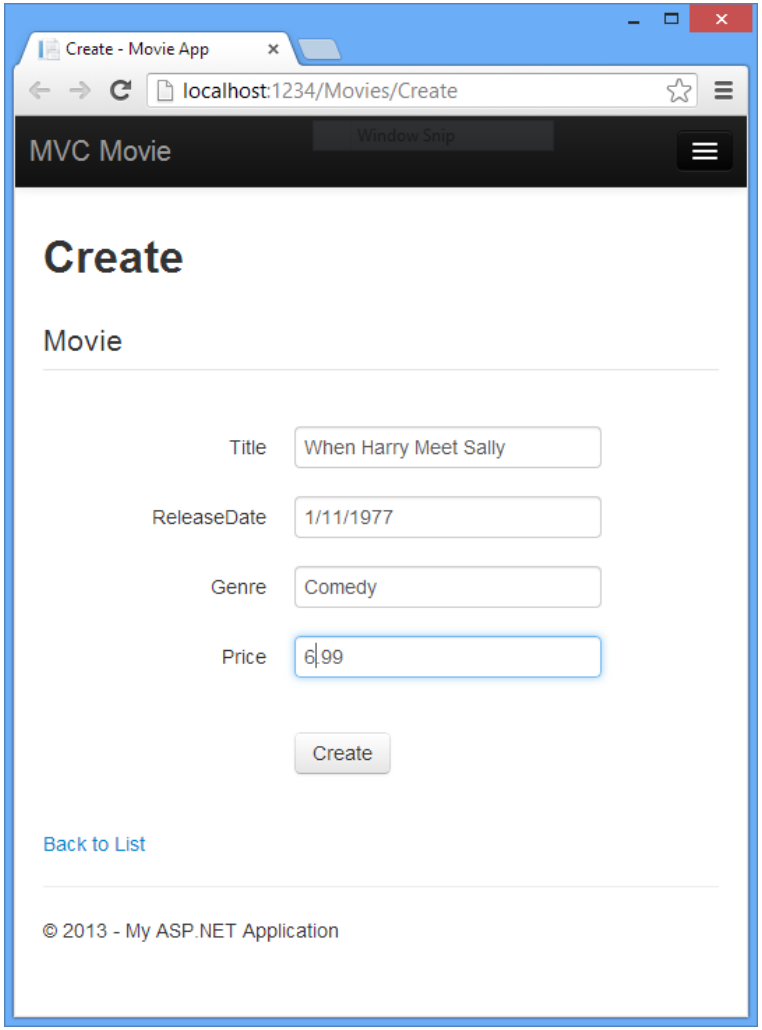
- The image below shows the completed dialog.



Run the application by appending */Movies* to the URL



Creating a Movie



REFERENCES

- http://www.asp.net/mvc/overview/getting-started/introduction/getting-started
- https://mva.microsoft.com/en-US/training-courses/introduction-to-asp-net-mvc-8322