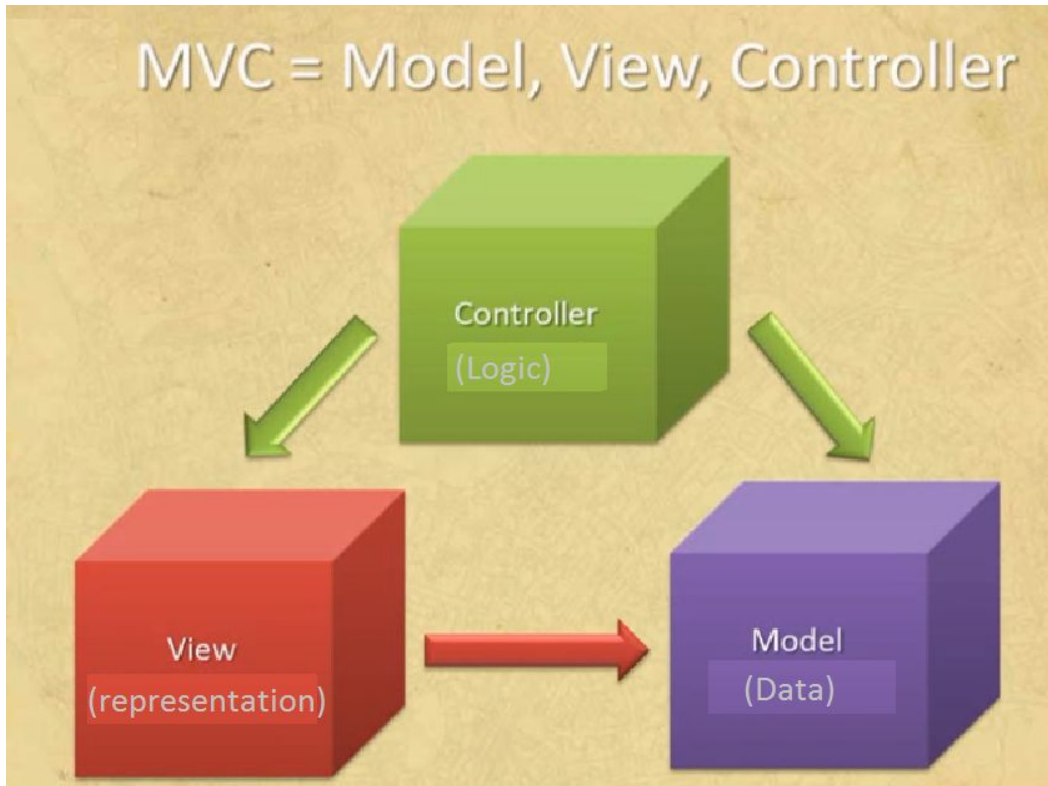


# ASP.NET MVC 5. Part 1

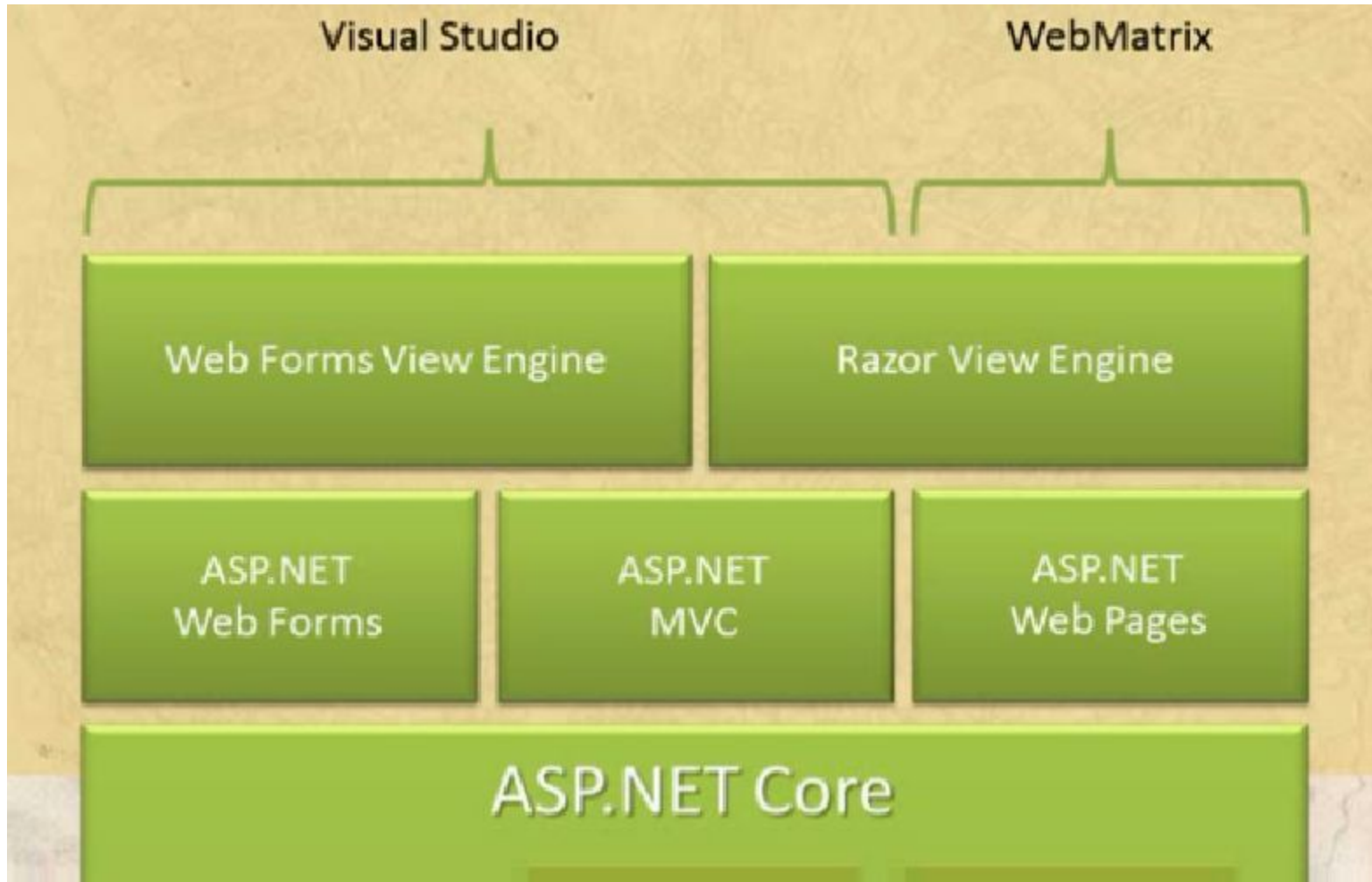
Overview. Controllers. Views.

2014-11-25 by O. Shvets  
Reviewed by O. Konovalenko

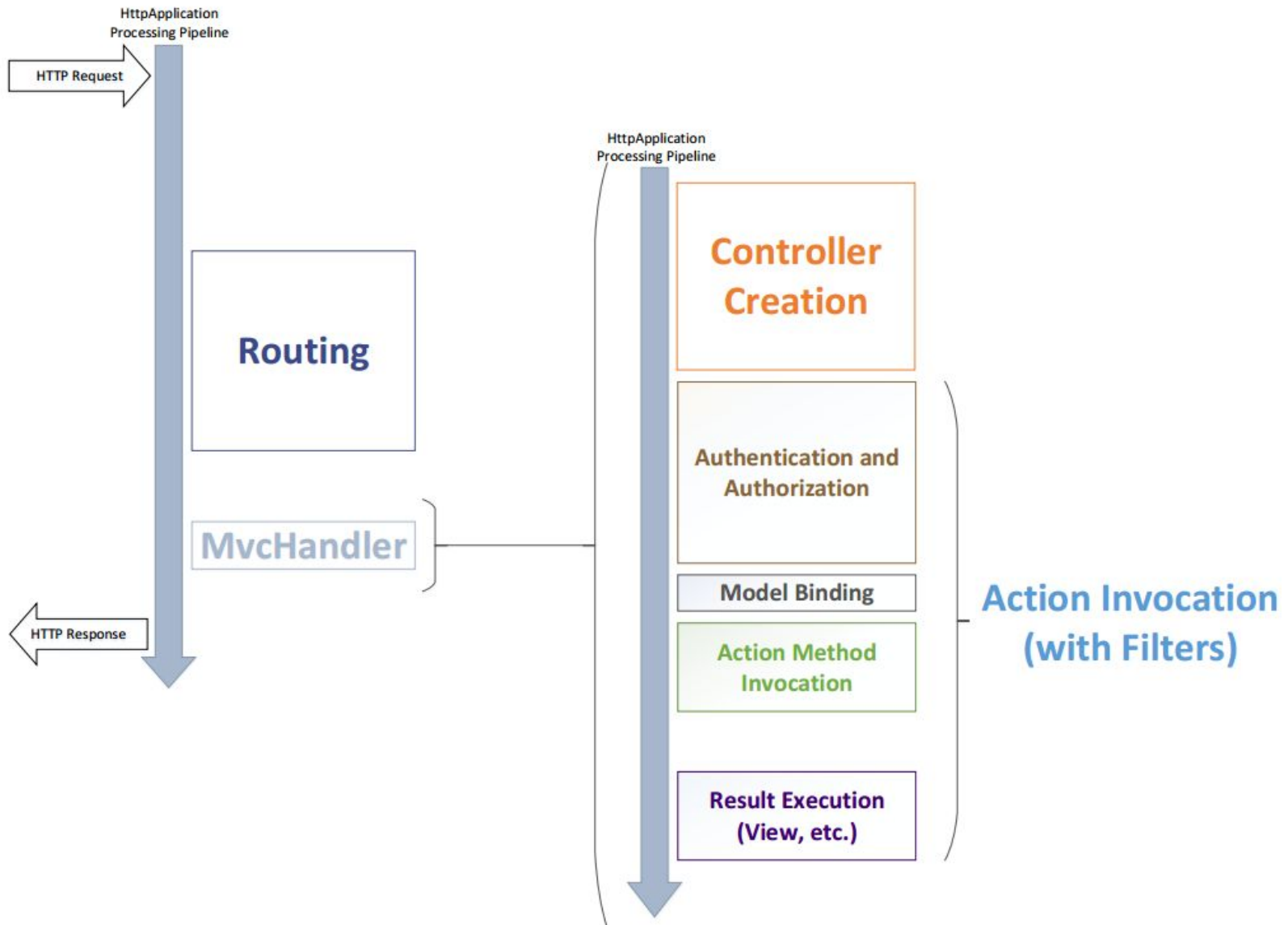
- ASP.NET Architecture
- ASP.NET MVC 3, 4, 5
- Controllers
- Views



- **Controller** – application logic. Communicate with user. It receives and handles user queries, interrupts with Model, and returns results by View objects
- **Model** – contains **classes** that represent **data**, performs operations with data-bases and organizes relations between data-classes.
- **View** – performs **UI** representation. Works with model.



# Lifecycle of an ASP.NET MVC 5 Application



- Higher quality requirements
  - Test Driven Development
- Cross platforms support
  - Windows, PDA, iPhone, ...
- HTML code control
- Clear ULR navigation
  - <http://musica.ua/groups/metallica>
- Maintainable code and command work

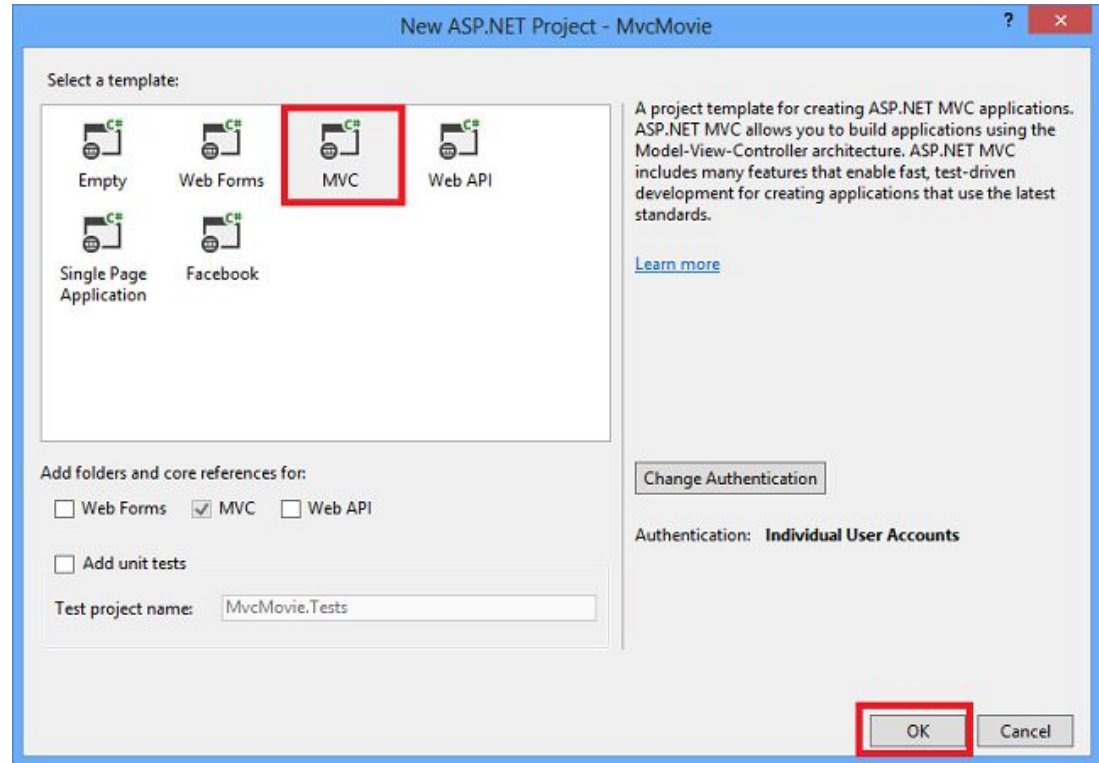
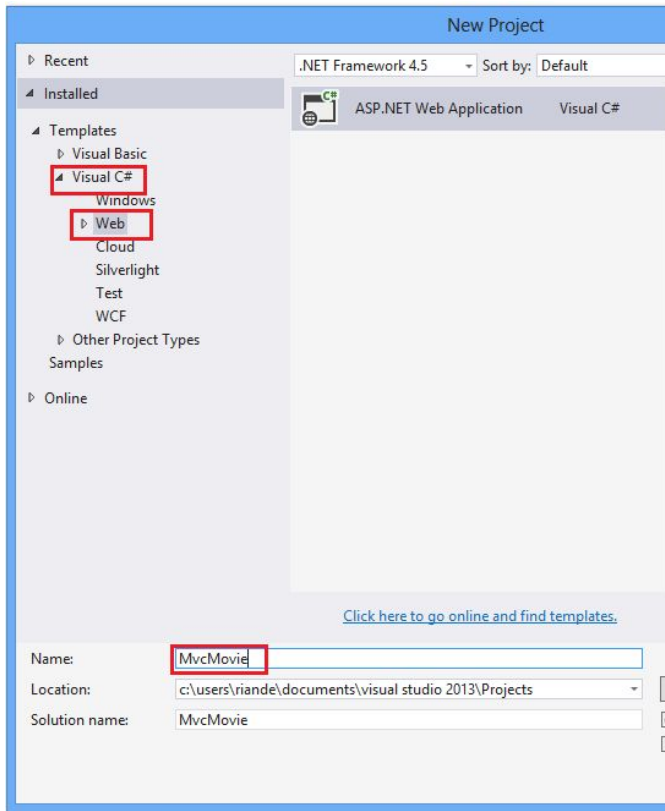
- Extensible Scaffolding with MvcScaffold integration
- HTML 5 enabled project templates
- The Razor View Engine
- Support for Multiple View Engines
- Controller Improvements
- JavaScript and Ajax
- Model Validation Improvements
- Dependency Injection Improvements

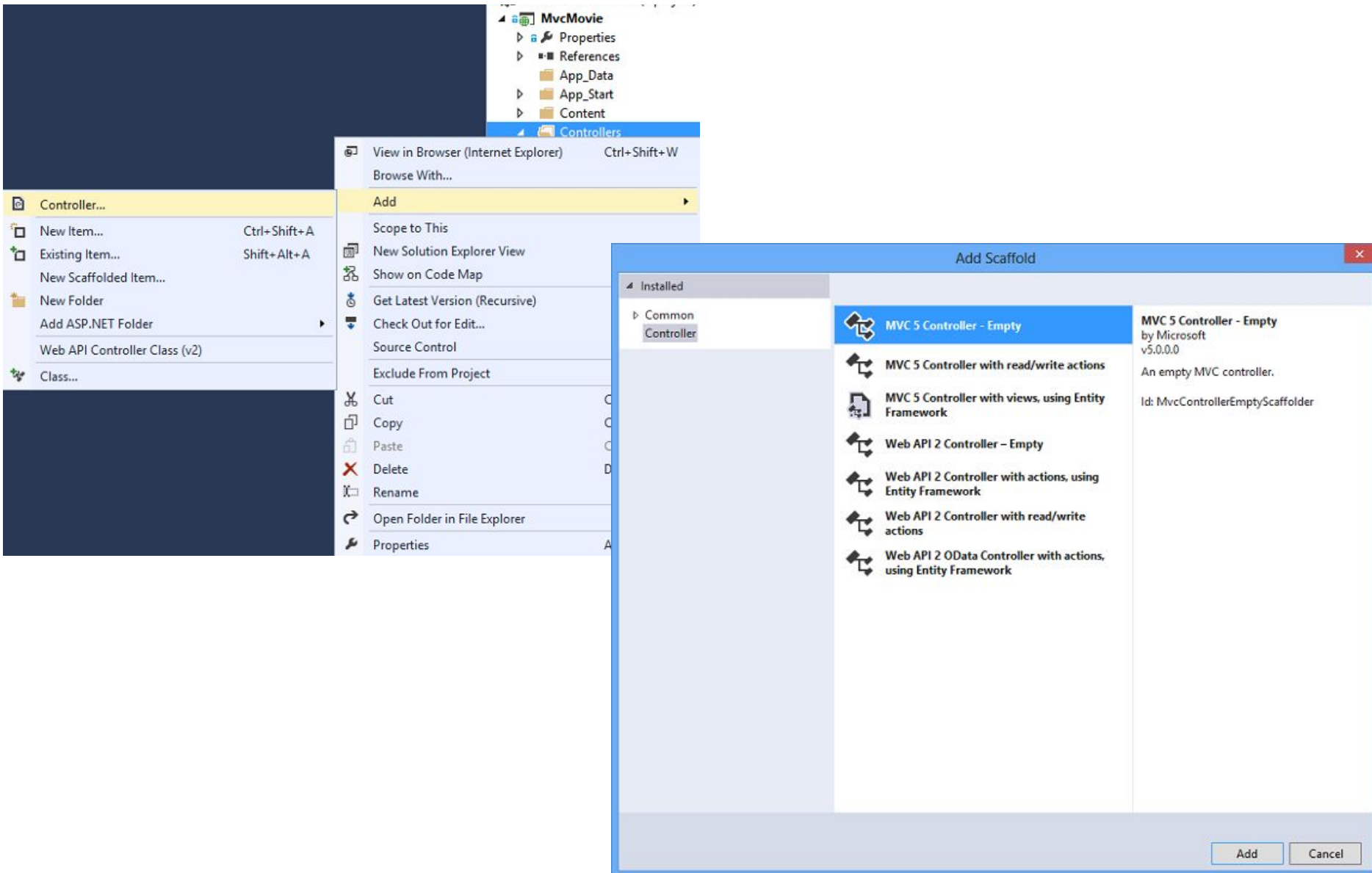
- ASP.NET Web API
- Enhancements to Default Project Templates
- Mobile Project Template and Empty Project Template
- jQuery Mobile, the View Switcher, and Browser Overriding
- Task Support for Asynchronous Controllers
- Azure SDK
- Database Migrations
- Add Controller to any project folder
- Bundling and Minification
- Enabling Logins from Facebook and Other Sites Using OAuth and OpenID



- One ASP.NET project template
- ASP.NET Identity
- Bootstrap
- Authentication filters
- Filter overrides
- Attribute routing

- **New Features in ASP.NET MVC 5.1**
  - Attribute routing improvements
  - Bootstrap support for editor templates
  - Enum support in views
  - Unobtrusive validation for MinLength/MaxLength Attributes
  - Supporting the 'this' context in Unobtrusive Ajax
- **New Features in ASP.NET MVC 5.2**
  - Attribute routing improvements





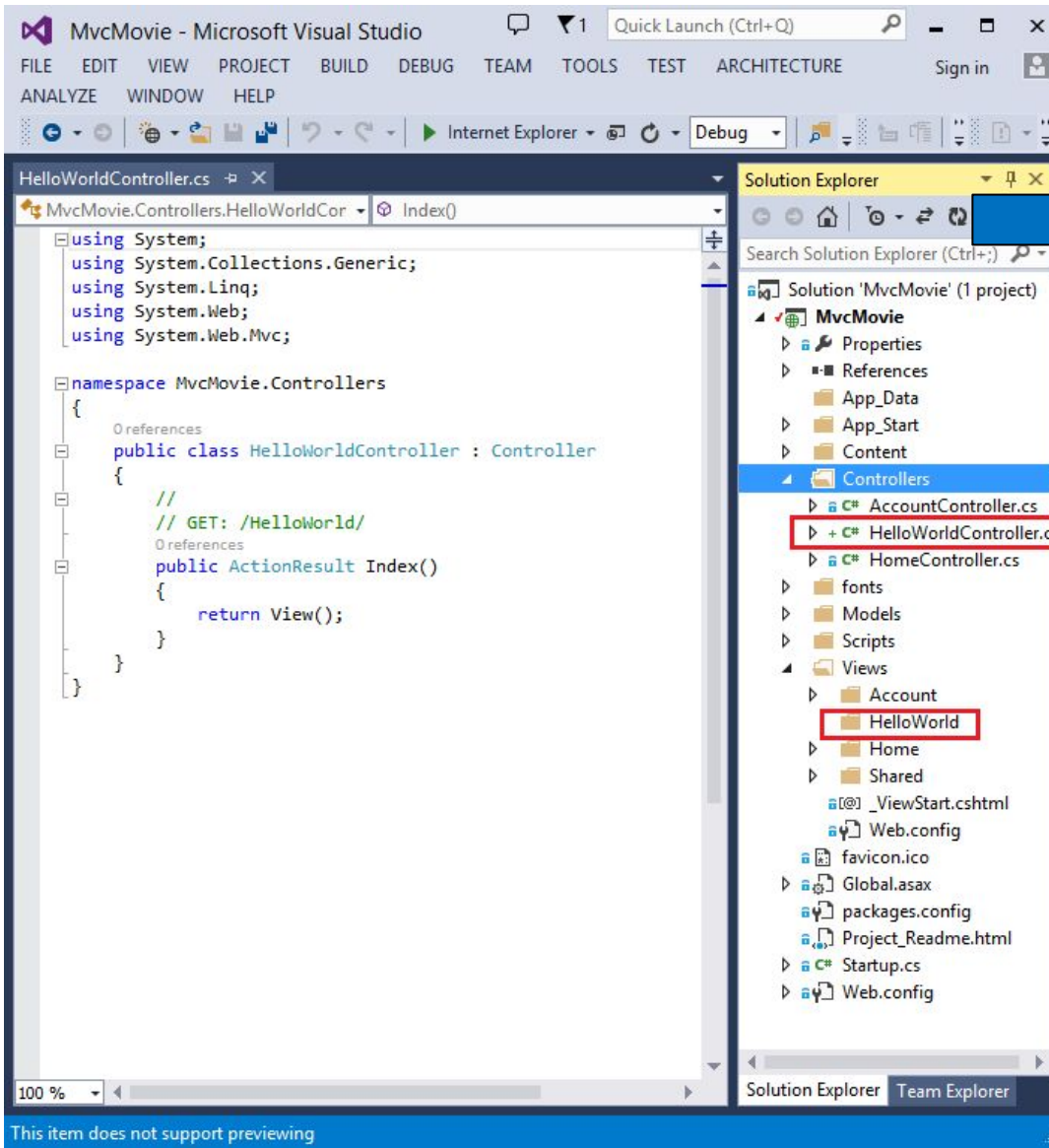
The screenshot illustrates the process of adding a controller to a project in Visual Studio. The background shows a project named 'MvcMovie' with a 'Controllers' folder. A context menu is open over the 'Controllers' folder, with the 'Add' option selected. The 'Add' submenu is also open, showing various options like 'New Item...', 'Existing Item...', and 'New Scaffolded Item...'. The 'New Scaffolded Item...' option is selected, leading to the 'Add Scaffold' dialog box.

The 'Add Scaffold' dialog box is titled 'Add Scaffold' and shows a list of installed scaffolds. The 'Common' folder is expanded, and the 'Controller' sub-folder is selected. The list of scaffolds includes:

- MVC 5 Controller - Empty** (selected): MVC 5 Controller - Empty by Microsoft v5.0.0.0. An empty MVC controller. Id: MvcControllerEmptyScaffolder
- MVC 5 Controller with read/write actions
- MVC 5 Controller with views, using Entity Framework
- Web API 2 Controller - Empty
- Web API 2 Controller with actions, using Entity Framework
- Web API 2 Controller with read/write actions
- Web API 2 OData Controller with actions, using Entity Framework

The 'Add' button is highlighted at the bottom right of the dialog box.

# Our New HelloWorldController



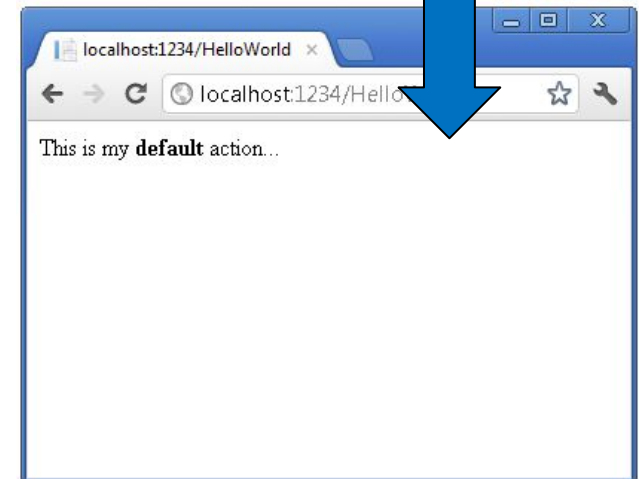
```
using System.Web;
using System.Web.Mvc;

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

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

        //
        // GET: /HelloWorld/Welcome/

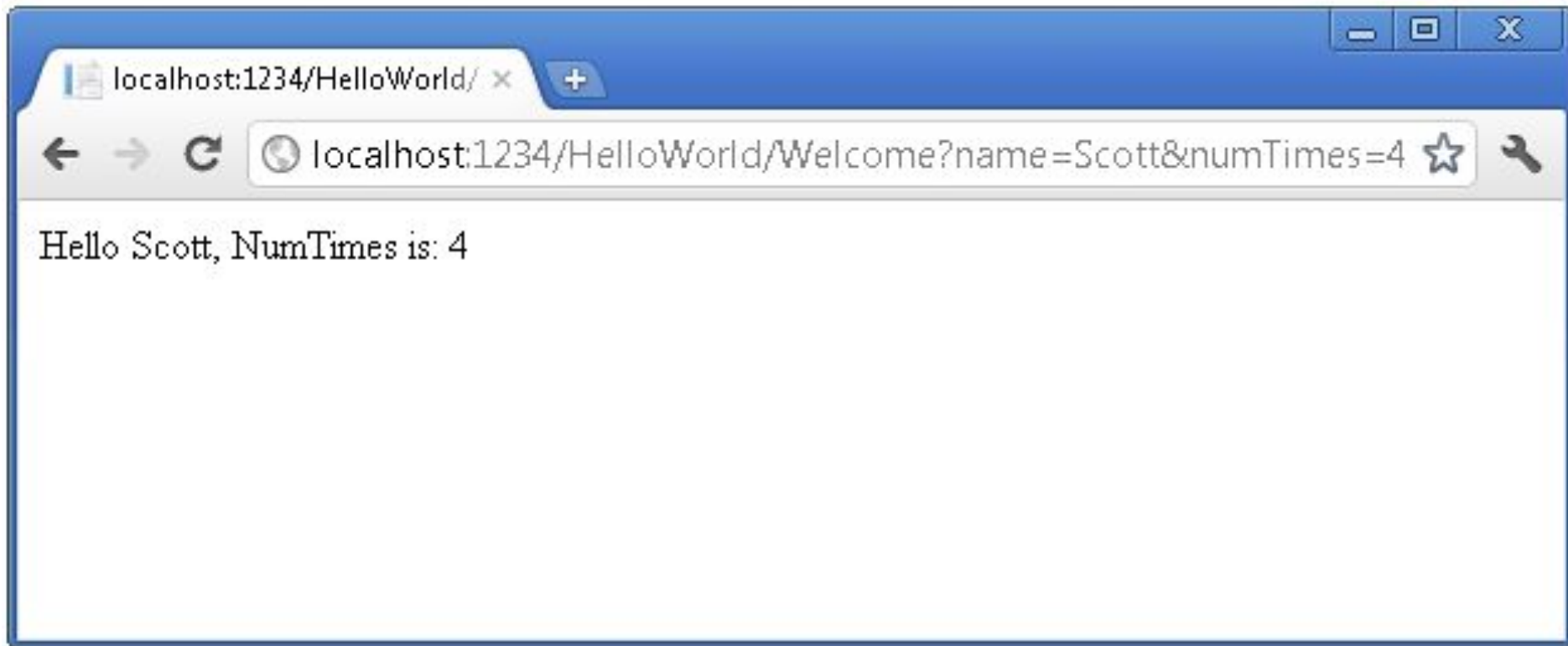
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```



```
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 }
    );
}
```

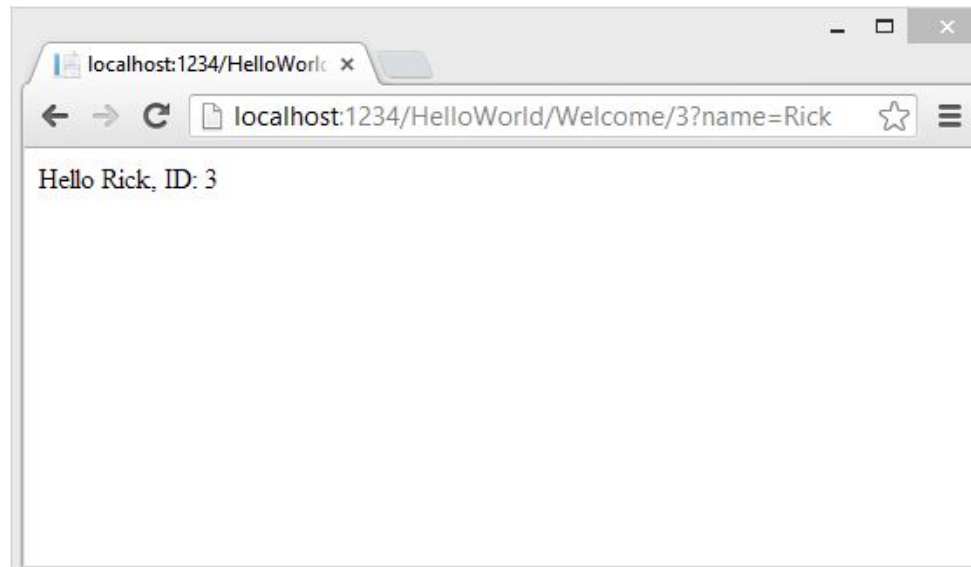
```
public string Welcome(string name, int numTimes = 1) {  
    return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);  
}
```



```
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 }
    );
}
```

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



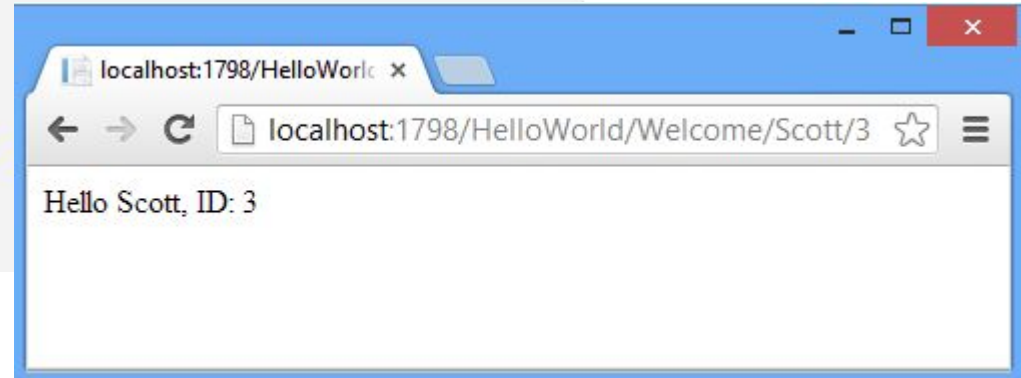


- In ASP.NET MVC applications, it's more typical to pass in parameters as route data than passing them as query strings

```
public class RouteConfig
{
    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 }
        );

        routes.MapRoute(
            name: "Hello",
            url: "{controller}/{action}/{name}/{id}"
        );
    }
}
```



- You can include "-", ".", ";" or any other characters you want as part of your route rules

- This would pass appropriate "language", "locale", and "category" parameters to a ProductsController:

```
{language}-{locale}/products/browse/{category}
```

```
/en-us/products/browse/food
```

```
language=en, locale=us, category=food
```

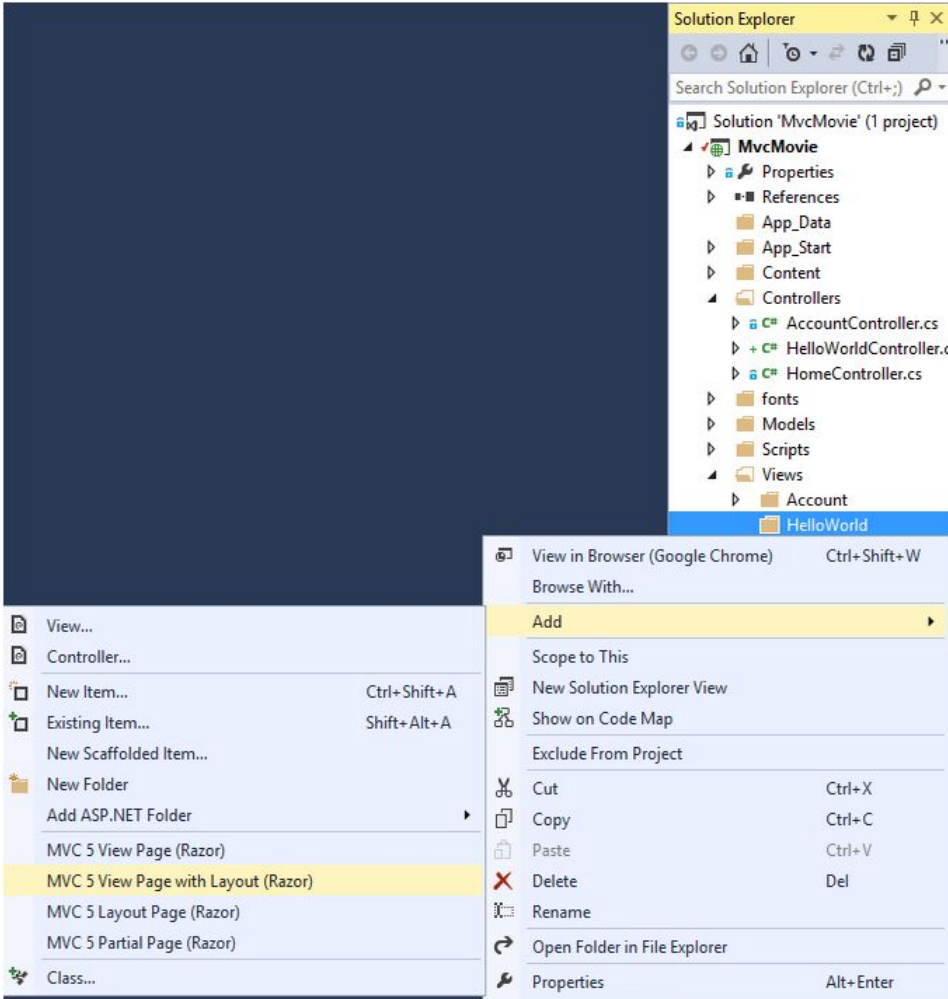
- You can use the "." file extension type at the end of a URL to determine whether to render back the result in either a XML or HTML format

```
products/browse/{category}.{format}
```

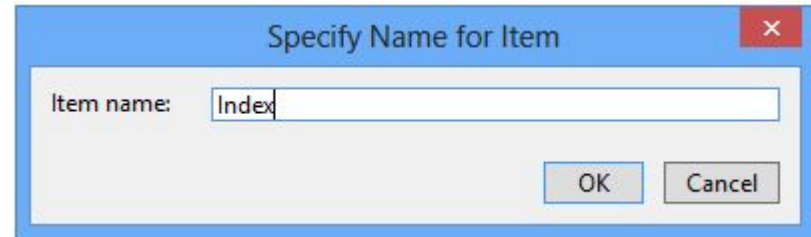
```
/products/browse/food.xml category=food, format=xml
```

```
/products/browse/food.html category=food, format=html
```

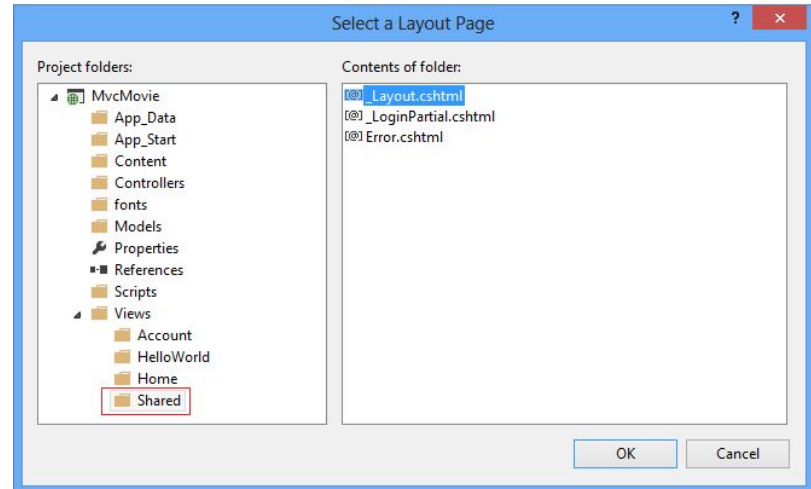
```
public ActionResult Index()  
{  
    return View();  
}
```



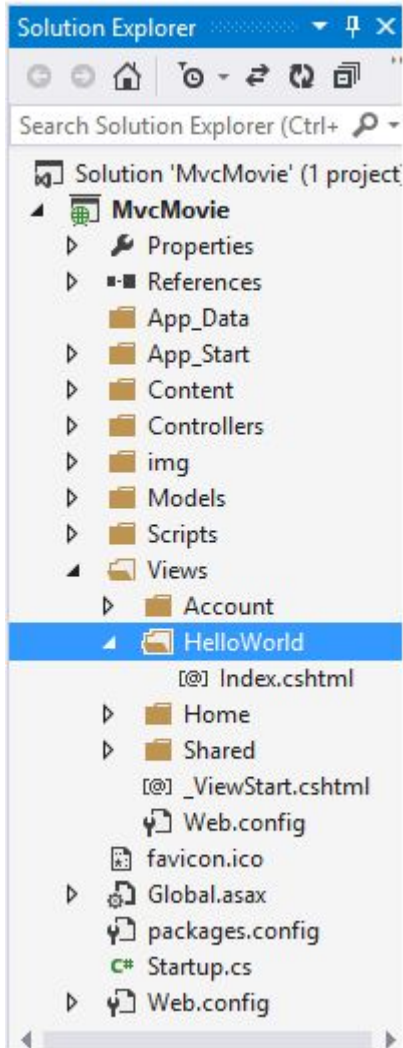
The screenshot shows the Visual Studio interface with the Solution Explorer on the right displaying the project structure. The 'Views' folder is selected, and a context menu is open. The menu options include 'View in Browser (Google Chrome)', 'Add', 'Scope to This', 'New Solution Explorer View', 'Show on Code Map', 'Exclude From Project', 'Cut', 'Copy', 'Paste', 'Delete', 'Rename', 'Open Folder in File Explorer', and 'Properties'. The 'Add' option is highlighted, and a sub-menu is visible with the following options: 'View...', 'Controller...', 'New Item...', 'Existing Item...', 'New Scaffolded Item...', 'New Folder', 'Add ASP.NET Folder', 'MVC 5 View Page (Razor)', 'MVC 5 View Page with Layout (Razor)', 'MVC 5 Layout Page (Razor)', 'MVC 5 Partial Page (Razor)', and 'Class...'.



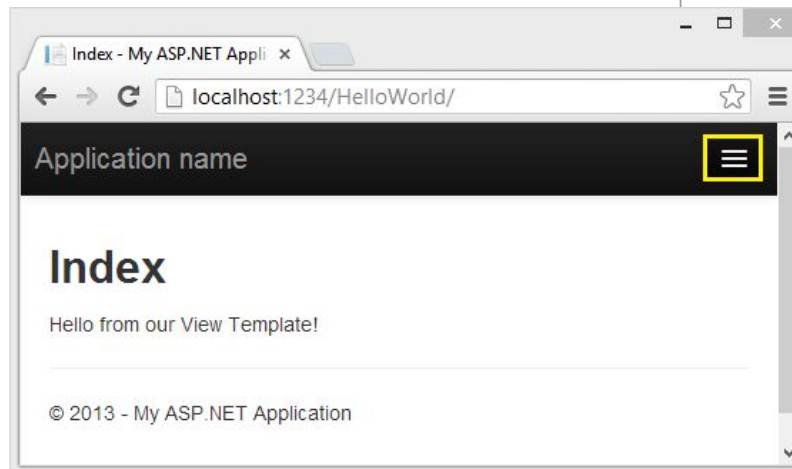
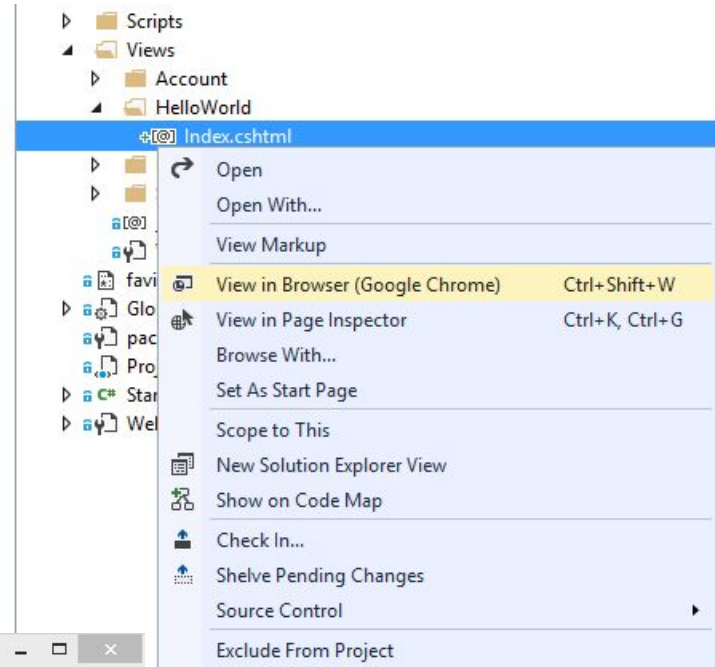
The 'Specify Name for Item' dialog box is shown. It has a title bar with a close button. The main area contains the text 'Item name:' followed by a text box containing the word 'Index'. At the bottom right, there are two buttons: 'OK' and 'Cancel'.

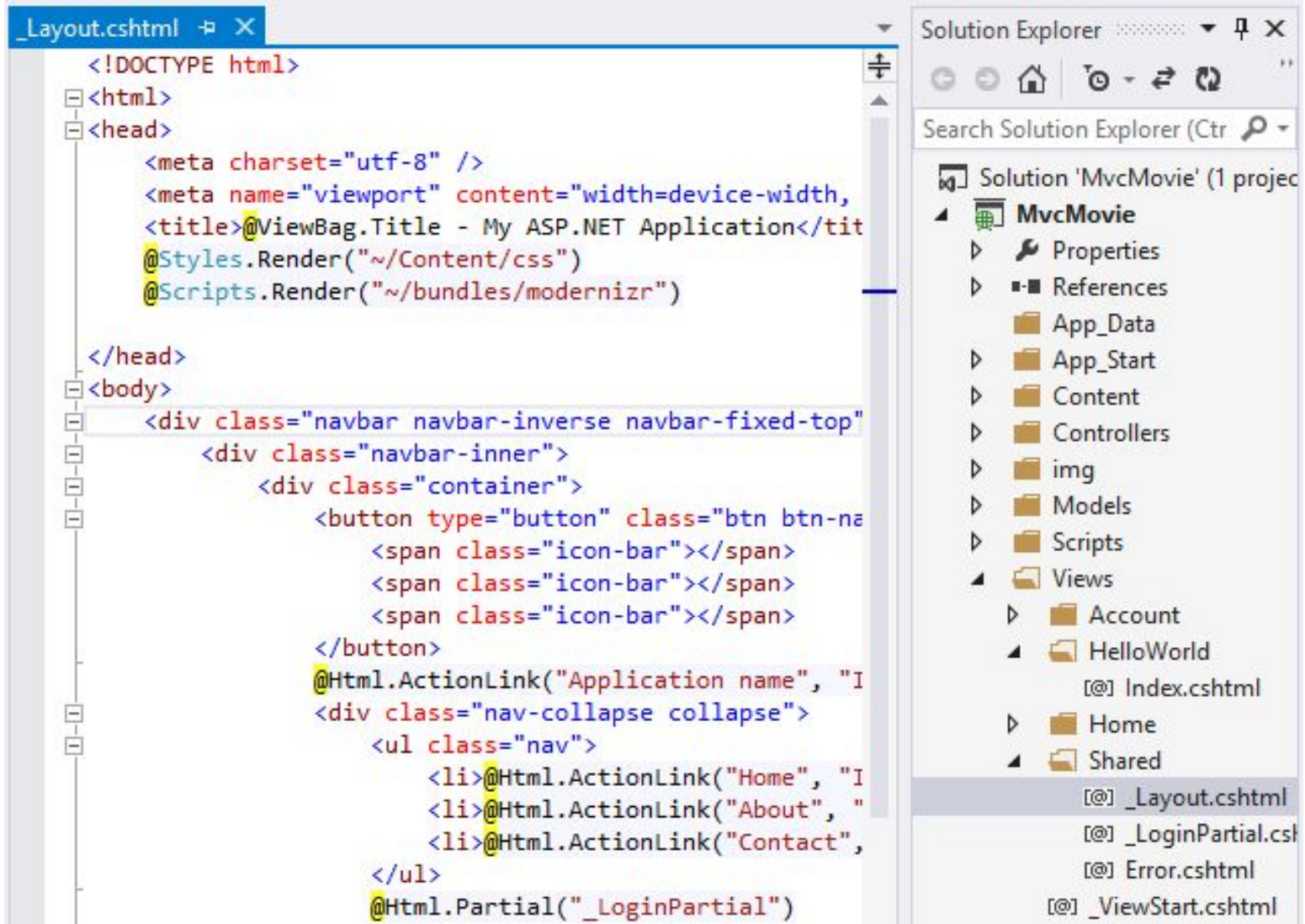


The 'Select a Layout Page' dialog box is shown. It has a title bar with a help icon and a close button. The main area is divided into two panes. The left pane, titled 'Project folders:', shows a tree view of the project structure with the 'Shared' folder under the 'Views' folder selected. The right pane, titled 'Contents of folder:', shows a list of files: '\_Layout.cshtml', '\_LoginPartial.cshtml', and 'Error.cshtml'. At the bottom right, there are two buttons: 'OK' and 'Cancel'.



```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}  
  
@{  
    ViewBag.Title = "Index";  
}  
  
<h2>Index</h2>  
  
<p>Hello from our View Template!</p>
```





The screenshot displays the Visual Studio IDE with the following components:

- Code Editor:** Shows the content of `_Layout.cshtml`. The code includes a DOCTYPE declaration, a `<html>` root element, a `<head>` section with meta tags for charset, viewport, and title, and a `<body>` section containing a navigation bar and a main content area.
- Solution Explorer:** Shows the project structure for 'MvcMovie'. The 'Views' folder is expanded, showing sub-folders like 'Account', 'HelloWorld', 'Home', and 'Shared'. The file `_Layout.cshtml` is selected and highlighted.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
  <title>@ViewBag.Title - My ASP.NET Application</tit
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <button type="button" class="btn btn-na
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("Application name", "I
        <div class="nav-collapse collapse">
          <ul class="nav">
            <li>@Html.ActionLink("Home", "I
            <li>@Html.ActionLink("About", "
            <li>@Html.ActionLink("Contact",
          </ul>
          @Html.Partial("_LoginPartial")
    </div>
  </div>
</body>
</html>
```

- The layout has access to the same properties the Razor view has, including:
  - AjaxHelper (through the Ajax property)
  - HtmlHelper (through the Html property)
  - ViewData and model
  - UrlHelper (through the Url property)
  - TempData and ViewContext
- To specify a layout inside a view, we can specify the layout to use with the Layout property:

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

- an alternative to the Web Forms view engine
- is responsible for rendering views in the Razor format (either .cshtml files or .vbhtml files)
  - The Web Form view engine is used to support the older-format Web Form views (.aspx and .ascx files)

### Web Forms view engine example:

```
<%@ Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<Product[]>" %>
<ul>
<% foreach(var product in Model) { %>
    <li><%= product.Name %></li>
<% } %>
</ul>
```

### Razor view engine example

```
@model Product[]
<ul>
@foreach(var product in Model) {
    <li>@product.Name</li>
}
</ul>
```

- '@' is the magic character that precedes code instructions in the following contexts
  - '@' For a single code line/values

```
<p>  
    Current time is: @DateTime.Now  
</p>
```

- '@{ ... }' For code blocks with multiple lines

```
@{  
    var name = "John";  
    var nameMessage = "Hello, my name is " + name + " Smith";  
}
```

- '@:' For single plain text to be rendered in the page

```
@{  
    @:The day is: @DateTime.Now.DayOfWeek. It is a <b>great</b> day!  
}
```



- HTML markup lines can be included at any part of the code:

```
@if(IsPost) {  
    <p>Hello, the time is @DateTime.Now and this  
    page is a postback!</p>  
} else {  
    <p>Hello, today is: </p> @DateTime.Now  
}
```

- **Razor uses code syntax to infer indent:**

```
// This won't work in Razor. Content has to be  
// wrapped between { }  
if( i < 1 ) int myVar=0;
```

- There are three different ways to pass data to a view:
  - by using the ViewDataDictionary,
  - by using the ViewBag,
  - by using strongly typed views.

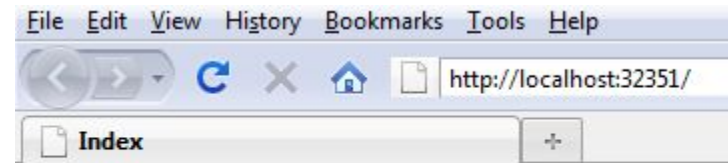
- It isn't recommended to use ViewDataDictionary
  - You have to perform type casts whenever you want to retrieve something from the dictionary.

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        int hour = DateTime.Now.Hour;
        ViewData["greeting"] = (hour < 12 ? "Good Morning" : "Good Afternoon");
        return View();
    }
}
```

```
@{
    Layout = null;
}
```

```
<!DOCTYPE html>
```

```
<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        Hello,
        @ViewData["greeting"], World (from the view)!
    </div>
</body>
</html>
```



Hello, Good Afternoon, World (from the view)!

- It isn't recommended to use ViewBag
- The ViewBag provides a way to pass data from the controller to the view
  - It makes use of the dynamic language features of C# 4
- Set properties on the dynamic ViewBag property within your controller:
- A ViewBag property is also available in the view:


```
public ActionResult About()  
{  
    ViewBag.Message = "Your app description page.";  
  
    return View();  
}
```

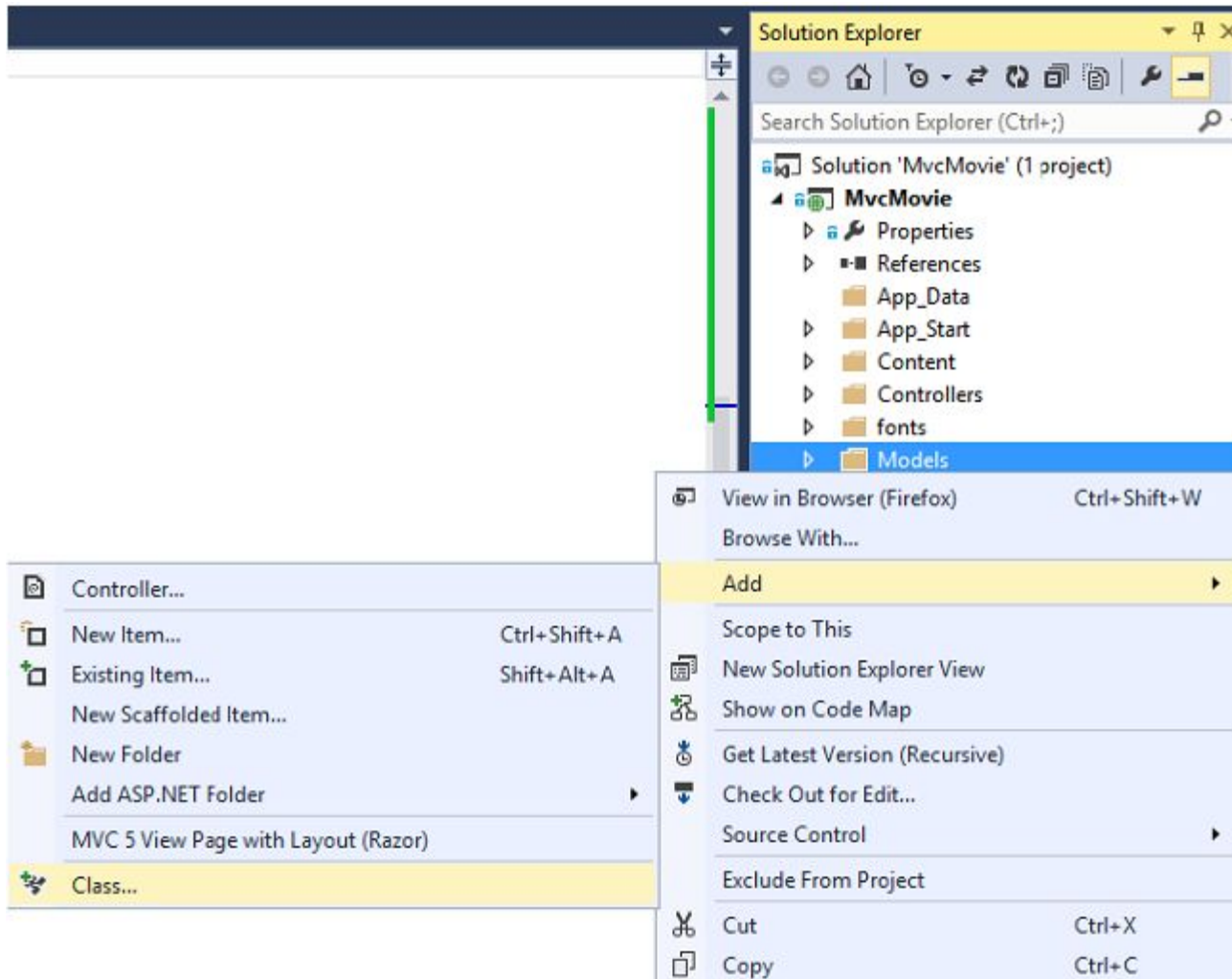
```
@{  
    ViewBag.Title = "About";  
}  
  
<hgroup class="title">  
    <h1>@ViewBag.Title.</h1>  
    <h2>@ViewBag.Message</h2>  
</hgroup>
```

- Views can inherit from two types by default:
  - System.Web.Mvc.WebViewPage or
  - System.Web.Mvc.WebViewPage<T>
- Class `WebViewPage<T>` provides a strongly typed wrapper over `ViewData.Model` through the `Model` property and provides access to strongly typed versions of the associated view helper objects - `AjaxHelper` and `HtmlHelper`

## Skeleton definition of `WebViewPage<T>`

```
public class WebViewPage<TModel> : WebViewPage
{
    public new AjaxHelper<TModel> Ajax { get; set; }
    public new HtmlHelper<TModel> Html { get; set; }
    public new TModel Model { get; }
    public new ViewDataDictionary<TModel> ViewData { get; set; }
}
```

 **1** Strongly typed view model



- By specifying the model type using the @model keyword, view will inherit from `WebViewPage<T>` instead of `WebViewPage`, and we will have a strongly typed view

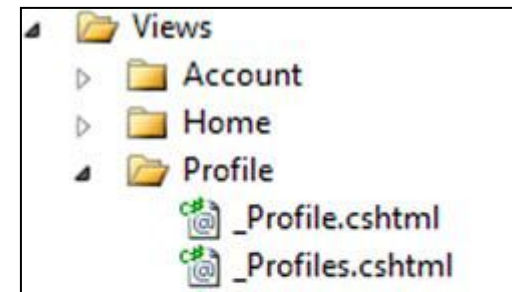
```
public ActionResult Index()
{
    //...
    SomeModel model = new SomeModel();
    return View(model);
}
```

```
<dl>
<dt>Name:</dt>
<dd>@Model.Name</dd>
<dt>Date Added:</dt>
<dd>@Model.DateAdded</dd>
<dt>Message:</dt>
<dd>@Model.Message</dd>
</dl>
```

- Partials are intended to render snippets of content
- If you find yourself copying and pasting one snippet of HTML from one view to the next, that snippet is a great candidate for a partial
- To render a partial we can use the `RenderPartial` method or the `Partial` method in a parent view

#### Rendering a partial from a parent view

```
@model IEnumerable<Profile>
<h2>Profiles</h2>
<table>
  <tr>
    <th>Username</th>
    <th>First name</th>
    <th>Last name</th>
    <th>Email</th>
  </tr>
  @foreach (var profile in Model) {
    @Html.Partial("_Profile", profile)
  }
</table>
```



#### A partial to display a row for a Profile model

```
@model AccountProfile.Models.Profile
<tr>
  <td>@Model.FirstName</td>
  <td>@Model.LastName</td>
  <td>@Model.Email</td>
</tr>
```



- The partial name is used to locate the partial markup in the locations:
  - `<Area>\<Controller>\<PartialName>.cshtml`
  - `<Area>\Shared\<PartialName>.cshtml`
  - `\<Controller>\<PartialName>.cshtml`
  - `\Shared\<PartialName>.cshtml`
- In order to prevent accidentally using a partial view from an action, we prefix the view name with an underscore
- `Html.RenderPartial(...)` renders the partial immediately to the response stream
- `Html.Partial(...)` returns a string
  - In Razor, `Html.RenderPartial` must be in a code block

