

**An E-Book on**

# **Razor View Engine in MVC 3**

**Abhimanyu Kumar Vatsa**

Founder of <http://www.itorian.com> and <http://www.microsoftskill.in>

*Dedicated to  
Mom (Mummy) & Dad (Papa)  
for their blessings, understanding and support*

[www.itorian.com](http://www.itorian.com) and [www.microsoftskill.in](http://www.microsoftskill.in)

### About the Author



**Abhimanyu** is a student of M.Sc.(IT) final semester. In January 2011 he has completed his **Bachelor in Computer Applications**. In addition with BCA he also holds certificates in **Advanced Diploma in Computer Applications, Diploma in Web Designing and Developments, Certificate in ASP.NET, Certification in Ethical Hacker** etc. **Vatsa** is administrator of about 10 featured websites of local cities. He has developed all this using ASP.NET, VB.NET, C#.NET, Ajax, JavaScript, JQuery, Silverlight, PHP, ColdFusion, SQL Server, MySQL and many more.

**Vatsa** has more than 4 years experience in building websites. He loves to work with Internet scale projects that have potential to change every day use of technology.

**Vatsa** is founder of [www.itorian.com](http://www.itorian.com) and [www.microsoftskill.in](http://www.microsoftskill.in) where he use to post his blogs, articles, tips, fixes, news or simply say he shares everything he learn. You can find over 350 technology posts on [www.itorian.com](http://www.itorian.com), he is getting huge comments, feedbacks on his posts. He also hold MVP award from Mindcracker community.

**Thanks**

### Why I'm writing this e-book?

I had the chance to learn more after joining Compu-ZIT ([www.compuzit.com](http://www.compuzit.com)) as an IT faculty. I used to gather the ideas of students while teaching to make the study interesting and purposeful. But over the time I felt the need for some concise materials so that any beginner (web) can be able to learn the concepts involved within the shortest possible time.

Many of my students even after finishing the courses lacks the fundamental knowledge involved in programming languages. Hence, I thought of to provide with materials and resources so that it will benefit not only the learners but also for the users all over the world. Since, the web is most reachable and cheapest medium to find information to almost all the users; I started [www.itorian.com](http://www.itorian.com) and writing e-books, posting articles, blogs, answering in forums etc.

### About this e-book

In this e-book you will learn all about Razor View Engine introduced in MVC 3. I will walk through the simple steps and even I will keep my ideas simple so that you can understand the Razor View Engine better. My aim through this e-book is to teach Razor so I am going to play a little loose with rest all.

### Disclaimer

My personal expertise and opinions expressed using this e-book. For any accuracy I recommend to visit official websites. For any damage arising from this e-book content, author or distributor will not be responsible.

### Assumptions I have made

I have made few assumptions about you. I have assumed that you have installed Visual Studio 2010 and configured for MVC 3 developments. Although you can read this e-book without them, you will have a harder time fully understanding what is being presented.

I have assumed that you are a beginning level programmer. If you are not, you will still gain a lot from this e-book; however, you might find that in some areas you will progress slower than you would like.

### Web Support

No one is perfect – especially me. Combine this with a programming language that is relatively new and that faces future changes. You can expect problems to crop up. It is very first edition of this e-book, editorial, technical and development reviews of the e-book have been done by me only. Even with all the reviews, errors still happen.

I have created a site specifically for the support of this e-book: <http://www.itorian.com/articles/mvc/post/283/>. I will post errata (errors) at this location and requesting the same to readers. Even I will love to hear your voice there; you welcome to post comments there.

### What is Razor?

Razor is the name of the new view engine introduced by Microsoft with the release of MVC 3. The ASP.NET view engines processes web pages, looking for special elements that contain server-side instructions. As we already know the standard ASPX view engine relies on the <% and %> elements, which is familiar to all ASP.NET developers. But with Razor View Engine, the MVC development team has introduced a new set of syntax elements, centered on the @ symbol.

If you are familiar with the <% %> syntax, you won't have too many problems with Razor, although there are a few new rules.

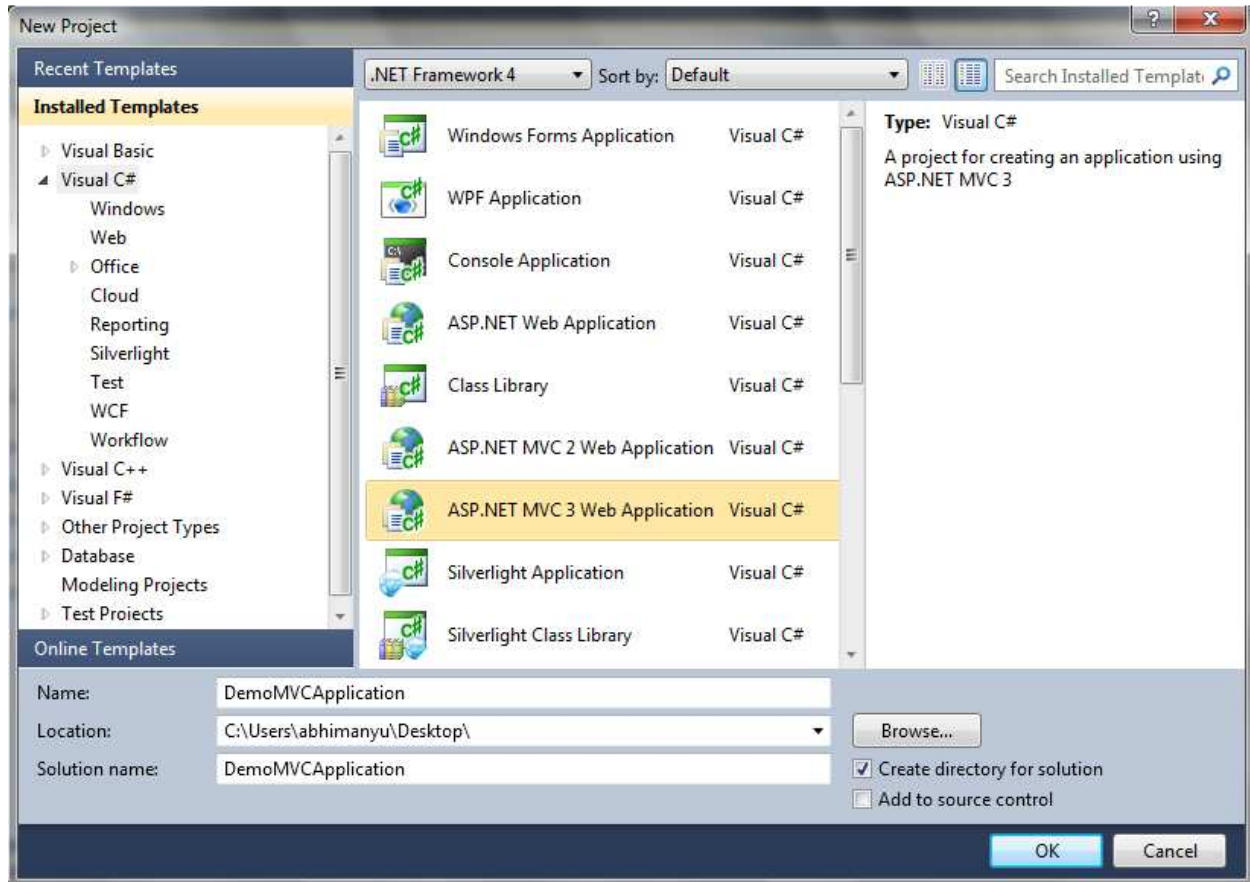
Razor views have a file extension of .cshtml, as opposed to the .aspx extension used in previous MVC releases and in ASP.NET Web Forms, we can still use the ASPX view engine in an MVC 3 project but try to learn Razor engine, because it seems to be a strong area of focus.

### Developing Sample Application in MVC 3 using Razor

Open the Visual Studio and create the new MVC 3 Project for the further learning in this e-book.

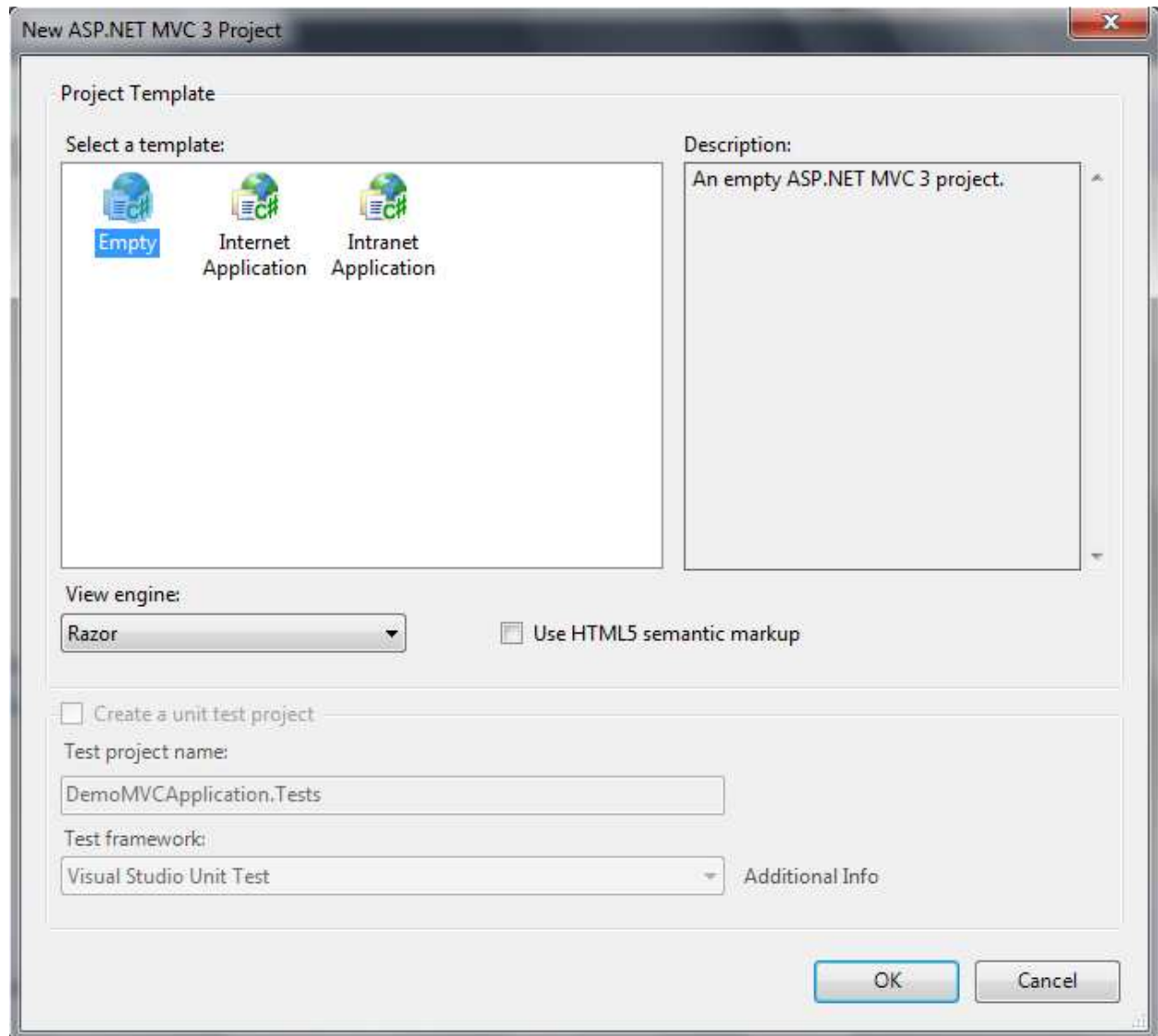
Follow the steps given below, remember to install MVC 3 extension if you don't have installed from here <http://www.asp.net/mvc/mvc3>.

Click on File > New > Project and click to open the following project selection window.



In above image I'm selecting "ASP.NET MVC 3 Web Application" using Visual C# as my primary language and also typing the appropriate project name and location.

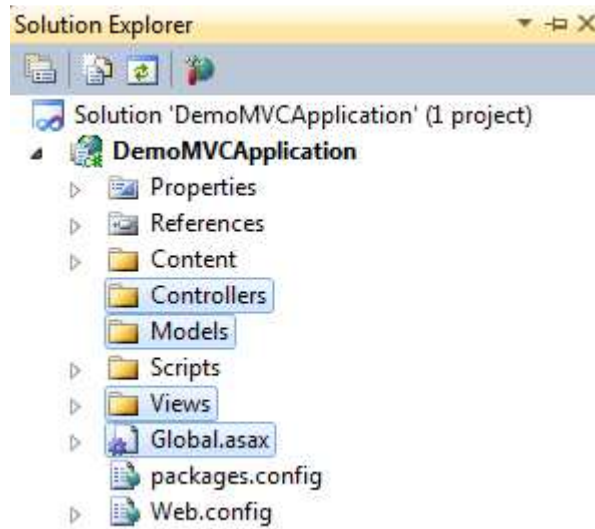
If you click on OK above, you will get following window.



In above image, I'm selecting project template as Empty and View engine as Razor. Without going to deeper in HTML5 semantic markup, I'll click on OK button.

It will open a nice pre-created folder hierarchy having some files that we'll need. Here is the screen.





Above image have some files and folders in a nice hierarchy that is created by Visual Studio by default.

For to learn this e-book we need to take a quick look over selected items in image namely Controllers, Models, Views, Global.asax.

### **Models (M)**

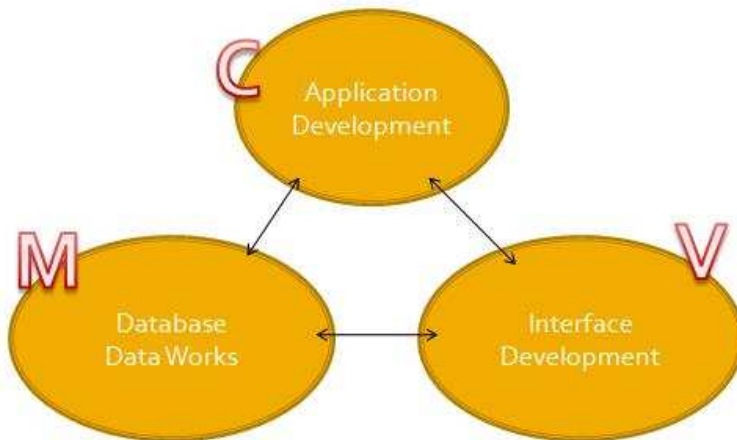
The model contains the core information for an application. This includes the data and validation rules as well as data access and aggregation logic.

### **Views (V)**

The view encapsulates the presentation of the application, and in ASP.NET this is typically the HTML markup.

### **Controllers (C)**

The controller contains the control-flow logic. It interacts with the Model and Views to control the flow of information and execution of the application.



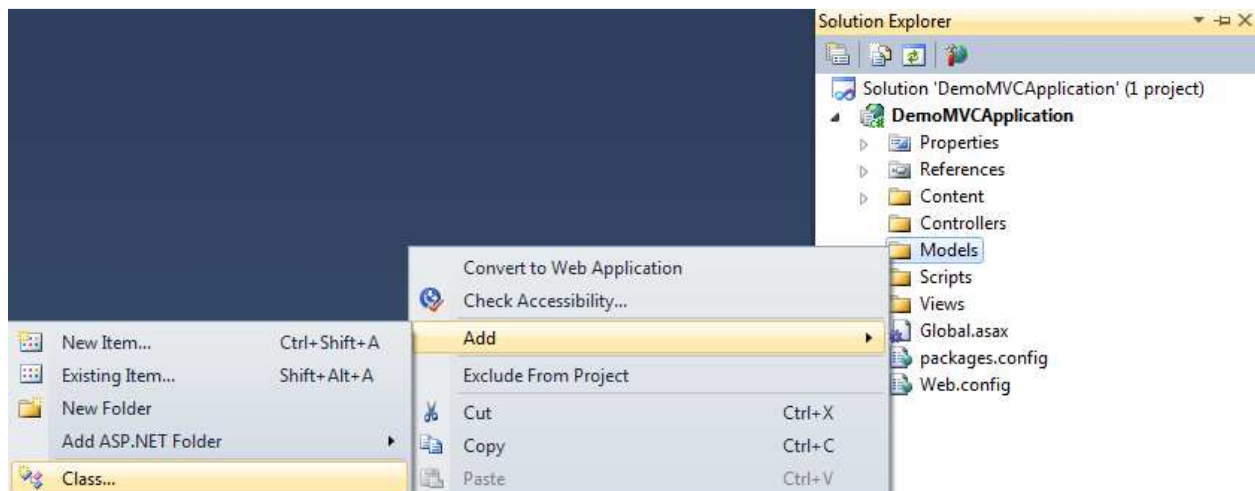
### Global.asax

This file is also known as application file which is used to perform the operations like RegisterRoutes, Application\_Start etc. In this e-book we will take a quick look over RegisterRoutes later.

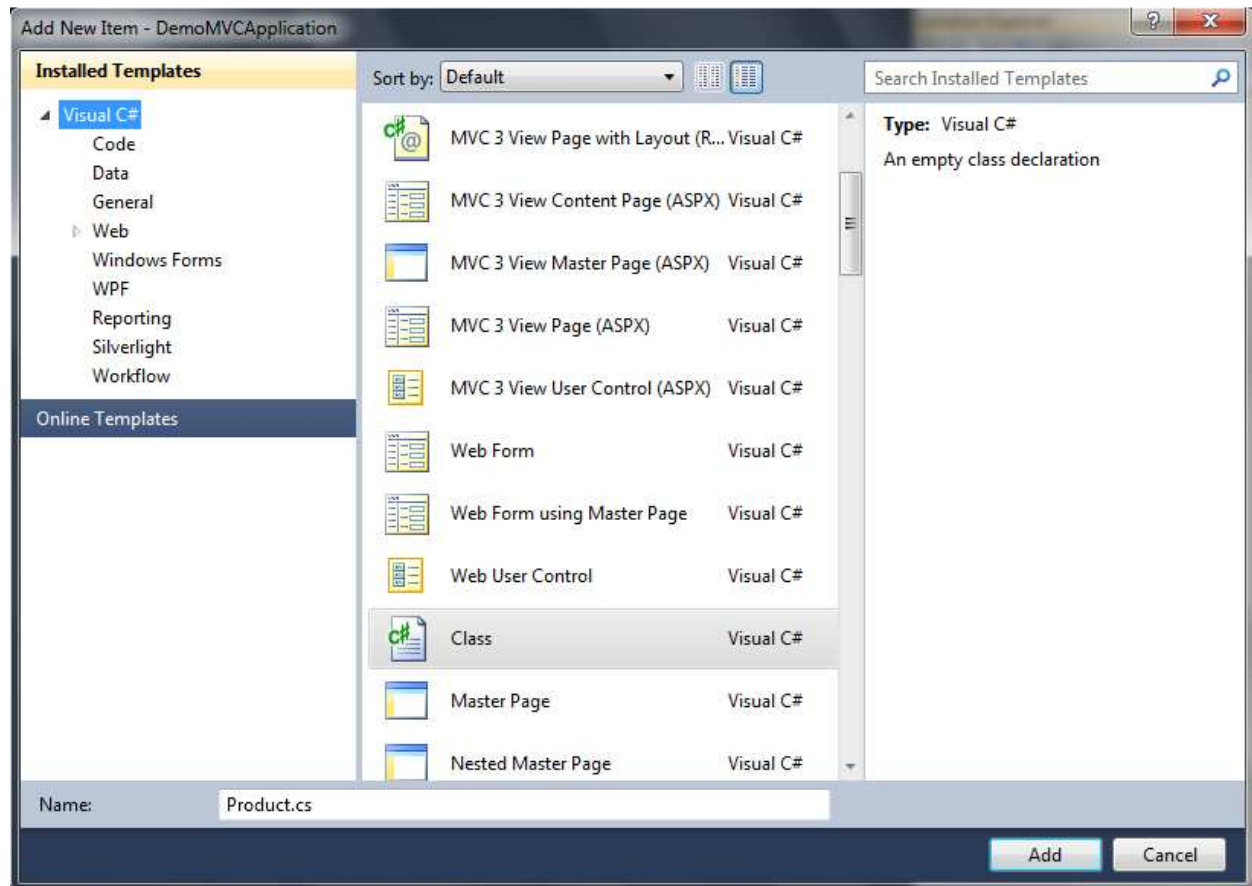
So finally, you have basic idea about above stuffs, let's proceed next.

### Adding Model to Project

To add the model in project, right click on Models folder and navigate to Add and then click on Class.

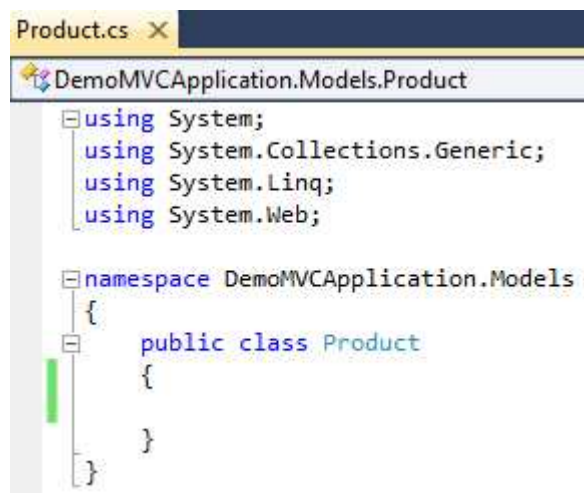


If you click on Class, this will bring windows to enter the class name, here is the screen.



In above screen I'm using class name as Product.cs, I'll recommend using the same if you are very beginner, at the end click on Add button.

This will add a file inside the Models folder and the code will look something like, as given below.



Now let's add our Model code in product class as given below.

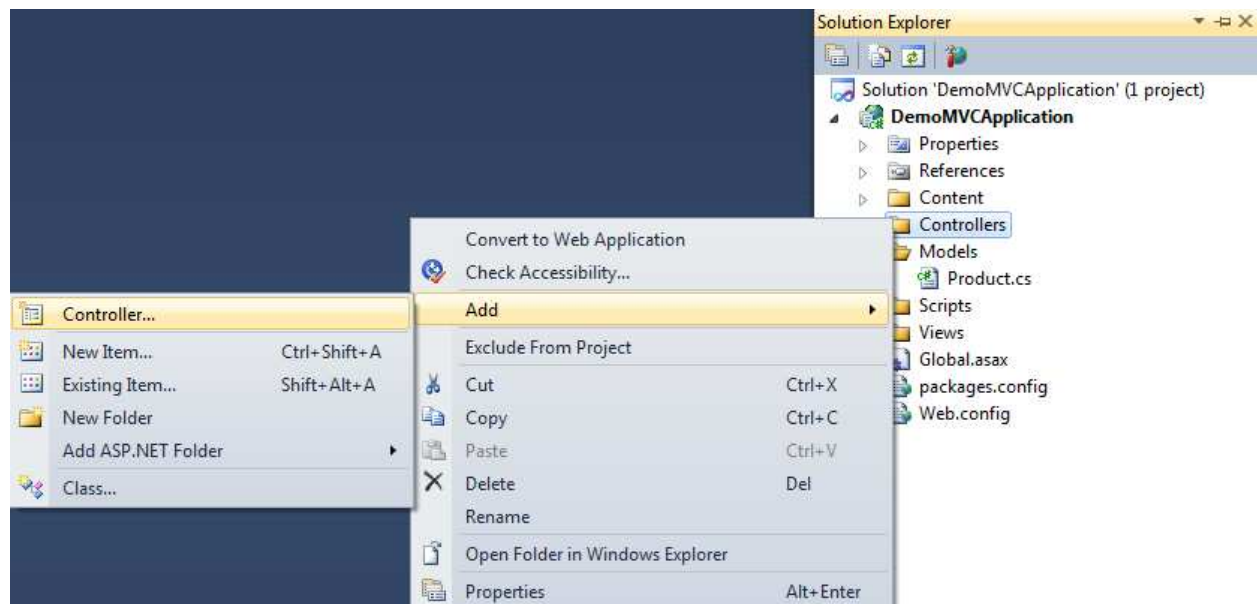
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace DemoMVCApplication.Models
{
    public class Product
    {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

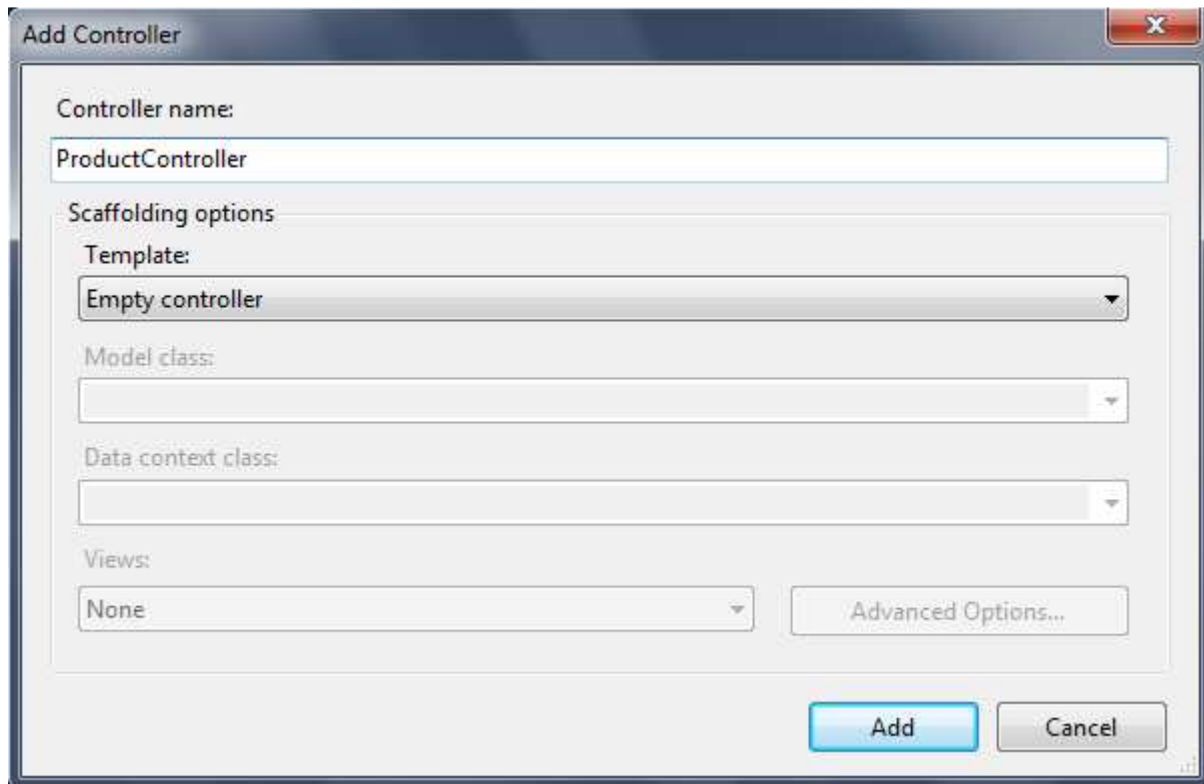
In above code, I'm using Automatically Implemented Properties that will be consumed by controller.

### Adding Controller to Project

Let's write the controller, follow the screen.



In above screen, I right clicked on Controller > Add > Controller, it will open following window.



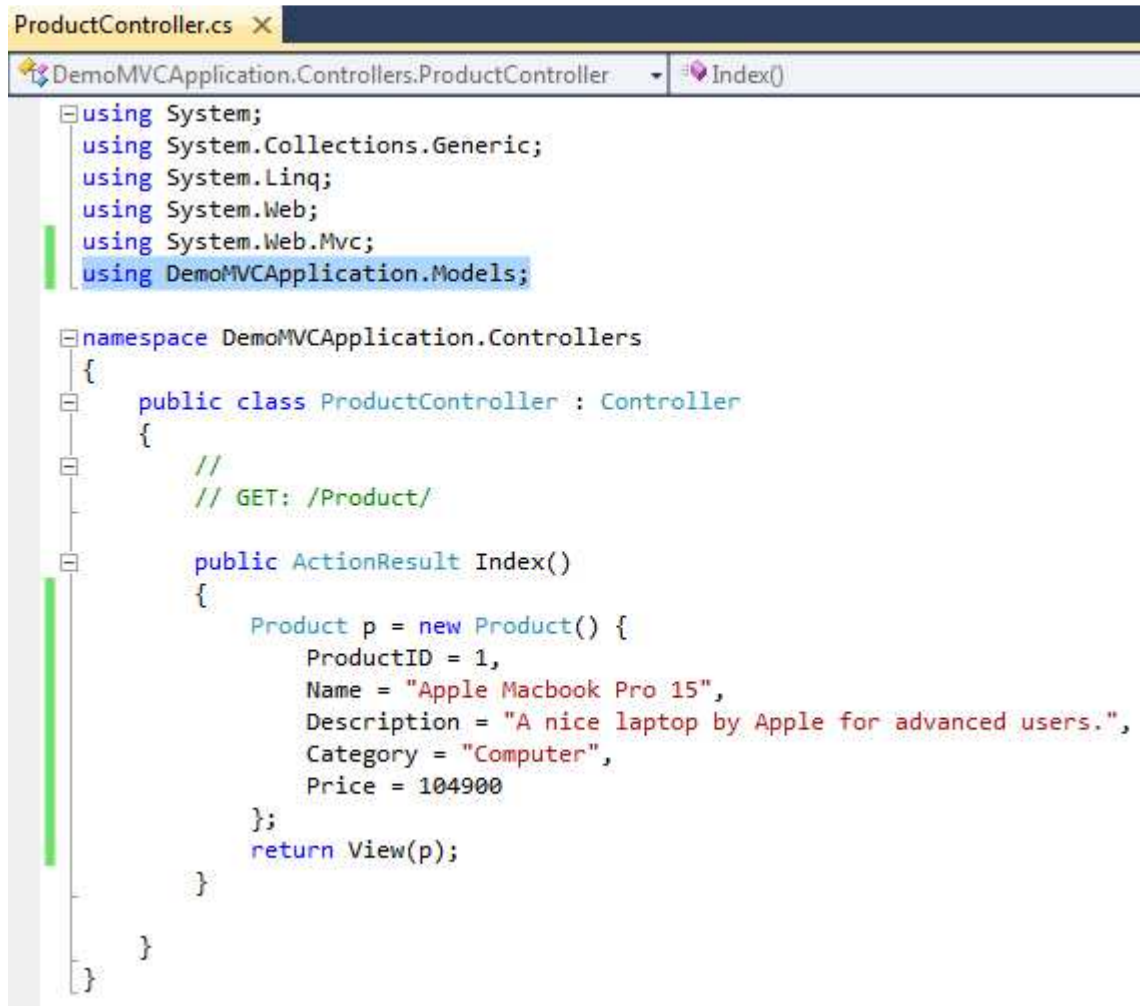
In this window type the Controller name as "ProductController" and rest all leave it default. At the end, click on Add button. It will add "ProductController.cs" controller class in the application and by default it will bring following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace DemoMVCApplication.Controllers
{
    public class ProductController : Controller
    {
        //
        // GET: /Product/

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

Now, we need create the object of above model class named Product.cs and add assign some values to the properties, as given in screen below.



```
ProductController.cs X
DemoMVCApplication.Controllers.ProductController Index()
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using DemoMVCApplication.Models;

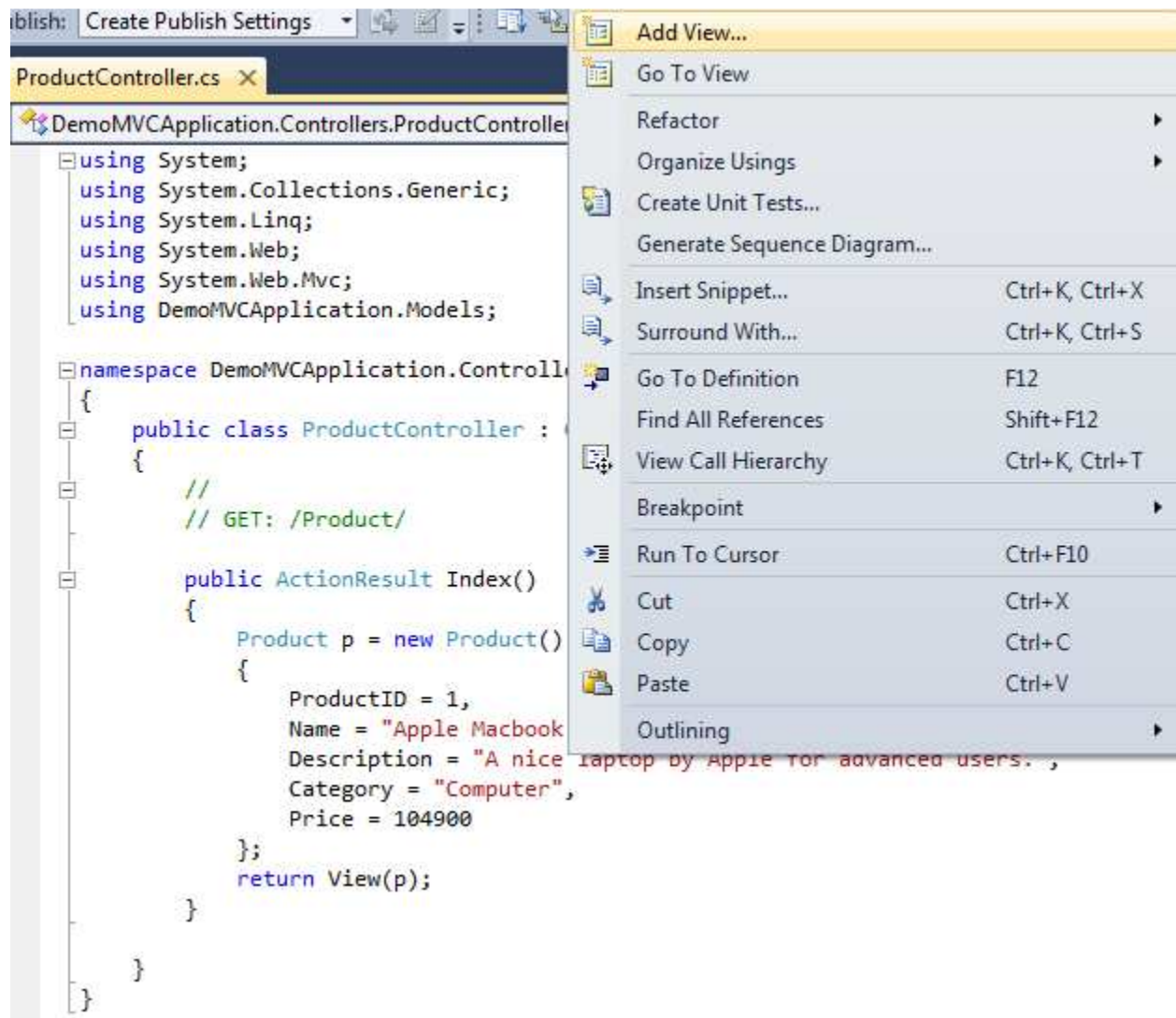
namespace DemoMVCApplication.Controllers
{
    public class ProductController : Controller
    {
        //
        // GET: /Product/

        public ActionResult Index()
        {
            Product p = new Product() {
                ProductID = 1,
                Name = "Apple Macbook Pro 15",
                Description = "A nice laptop by Apple for advanced users.",
                Category = "Computer",
                Price = 104900
            };
            return View(p);
        }
    }
}
```

In above code, I have assigned values for my all properties defined in model class above.

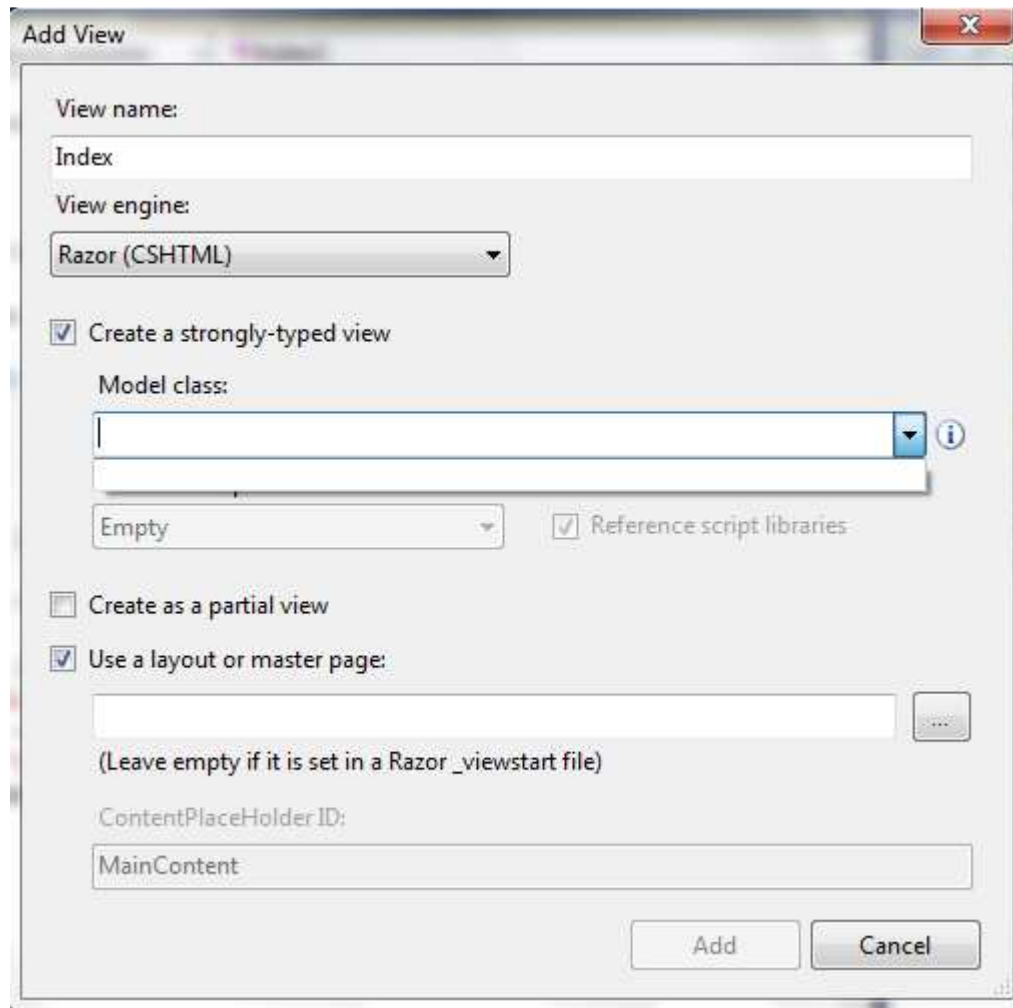
### Adding View to Project

Now we need to add the view for this Controller class. To do this, right click on above code and navigate to Add View option.



When you click on Add View option above, it will bring a window which will ask to enter the view name.





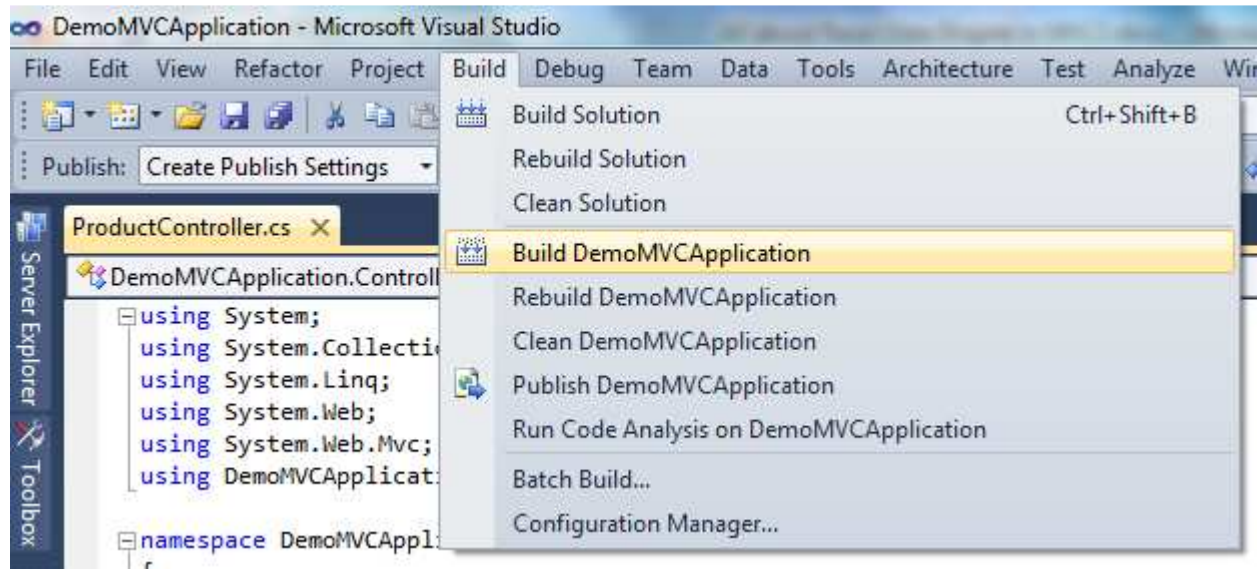
Leave the view as it is. Notice the view engine above that is Razor, that's the thing we are going to look-on throughout this e-book.

I'll recommend, checking the "Create a strongly-typed view" option and selecting the Model class name from the drop down menu. When you try to select the model class name it will not bring up in list, reason your project is not compiled. To compile the project follow the screen given below.

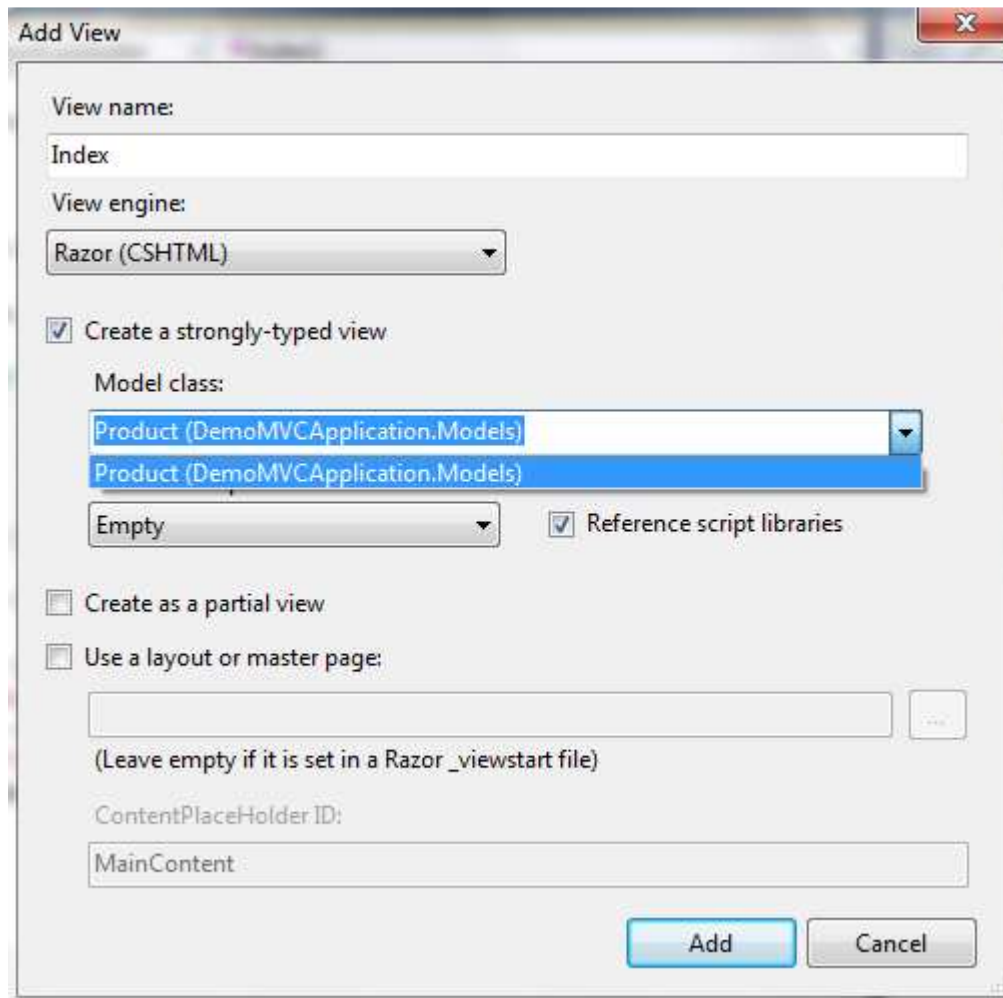
### Strongly Typed View

A strongly typed view is intended to render a specific domain type, and if we specify the type we want to work with, MVC can create some helpful shortcuts to make it easier.





After you success to compile the project, you again need to add the view for the controller class. Now, your model class name will appear in drop down list box. Please note to uncheck the "Use a layout or master page" option, because we are going little loose with rest options. When you finish with above, click on Add button as given in screen below.



When you click on Add button, it opens a view page which has .htmlcs extension. By default you will get the following codes on your view page. I have added a "Sample Message" in <div> tag in extra.

```
@model DemoMVCApplication.Models.Product
```

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

```
<!DOCTYPE html>
```

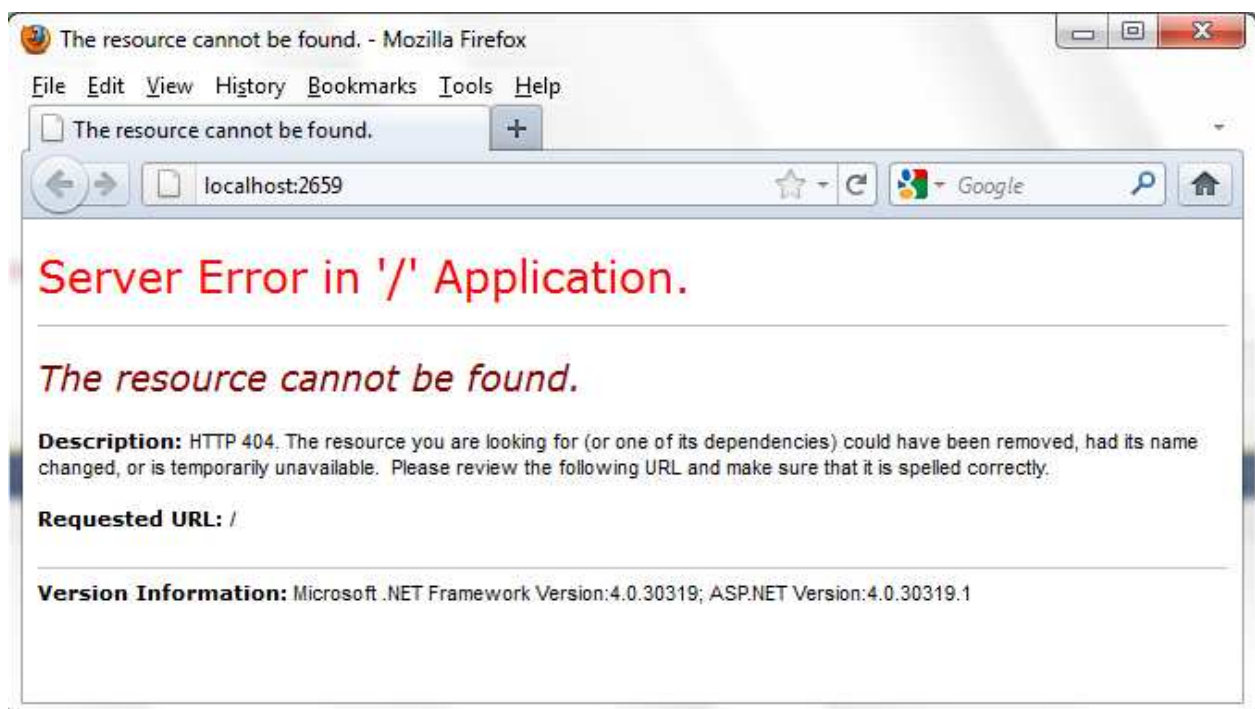
```
<html>  
<head>  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        Sample Message  
    </div>
```

```
</body>  
</html>
```

Now our page is ready to test the Razor View Engine coding. Let's start learning Razor View Engine now.

### Razor View Engine Coding

Till now we just have created a stage to run the Razor codes. For now, if you run (Start Debugging) this application, it will bring the error page.



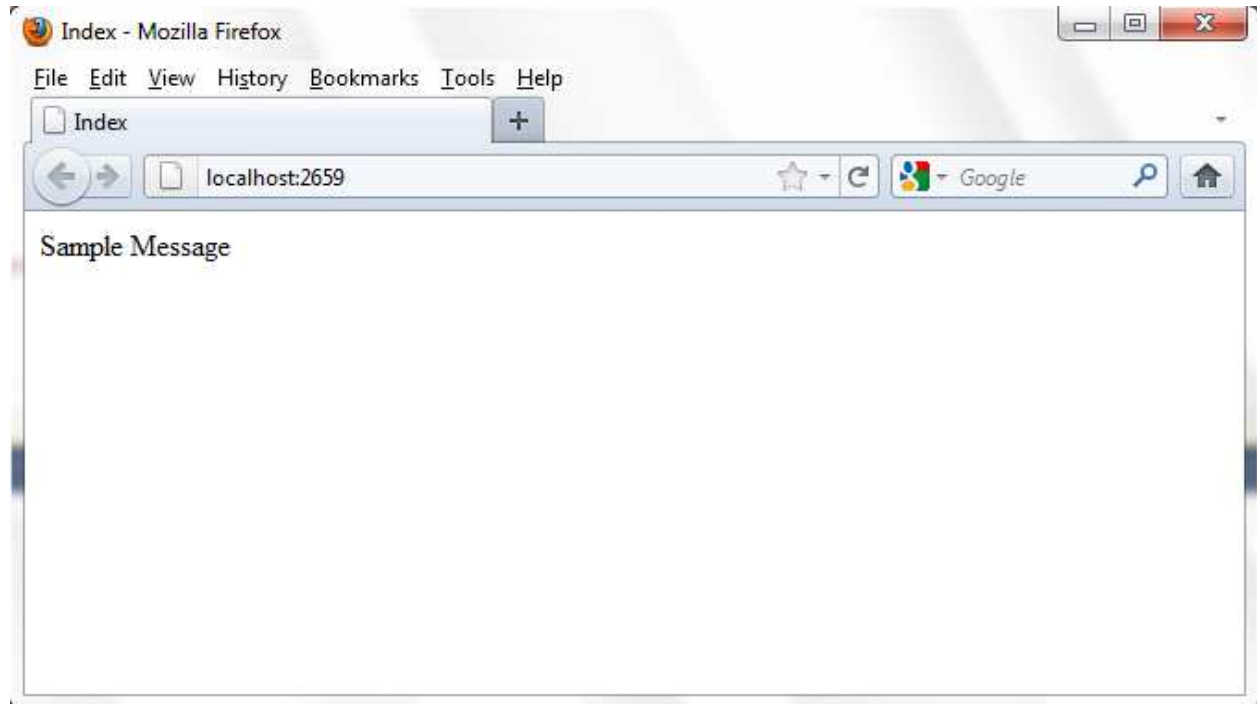
The reason, we need to set the Default Route for the application in Global.asax file (we are already bit aware about Global.asax file). Let's set up the Default Route for this application.

Open the Global.asax file and edit the code as given below.

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

    routes.MapRoute(
        "Default", // Route name
        "{controller}/{action}/{id}", // URL with parameters
        new { controller = "Product", action = "Index", id = UrlParameter.Optional } // Parameter defaults
    );
}
```

In above code, I have changed the Home Route to Product Route, where our target page is placed by the action name Index. Now, run the application you will see the output as given in screen below.



Now, let's display the model objects using Razor View Engine.

### **Displaying Model Objects using Razor**

#### ***View Data Feature***

Open the view page named Index.htmlcs and type following coding.



```
ProductController.cs  Index.cshtml X
Client Objects & Events
@model DemoMVCApplication.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

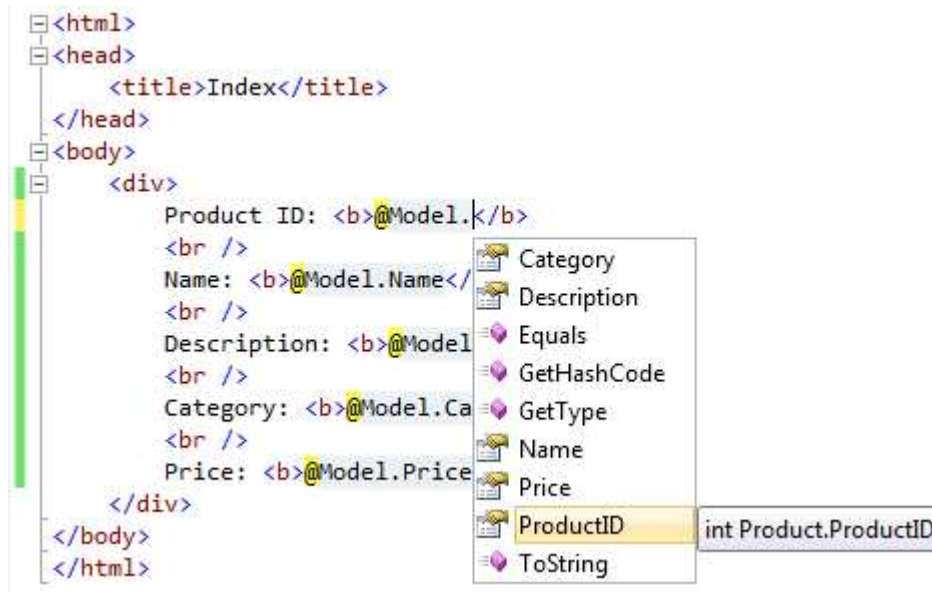
<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        Product ID: <b>@Model.ProductID</b>
        <br />
        Name: <b>@Model.Name</b>
        <br />
        Description: <b>@Model.Description</b>
        <br />
        Category: <b>@Model.Category</b>
        <br />
        Price: <b>@Model.Price</b>
    </div>
</body>
</html>
```

In above image, everything is normal like standard HTML tags excepting @ sign. Let's have introduction over them one by one.

### @model DemoMVCApplication.Models.Product

Razor statements start with the @ character. In this case, we are defining the model view that we can refer to in the rest of the view, using @model. As we already have quick discussion on strongly typed view that it let us create some shortcuts. Let's test this.

When we type @Model. then it pops to select appropriate option, here is the screen.



This is because we have created our view using strongly typed view option. So, finally we can say `@model DemoMVCApplication.Models.Product` is enabling the system to pop above option. Let's move to next line.

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

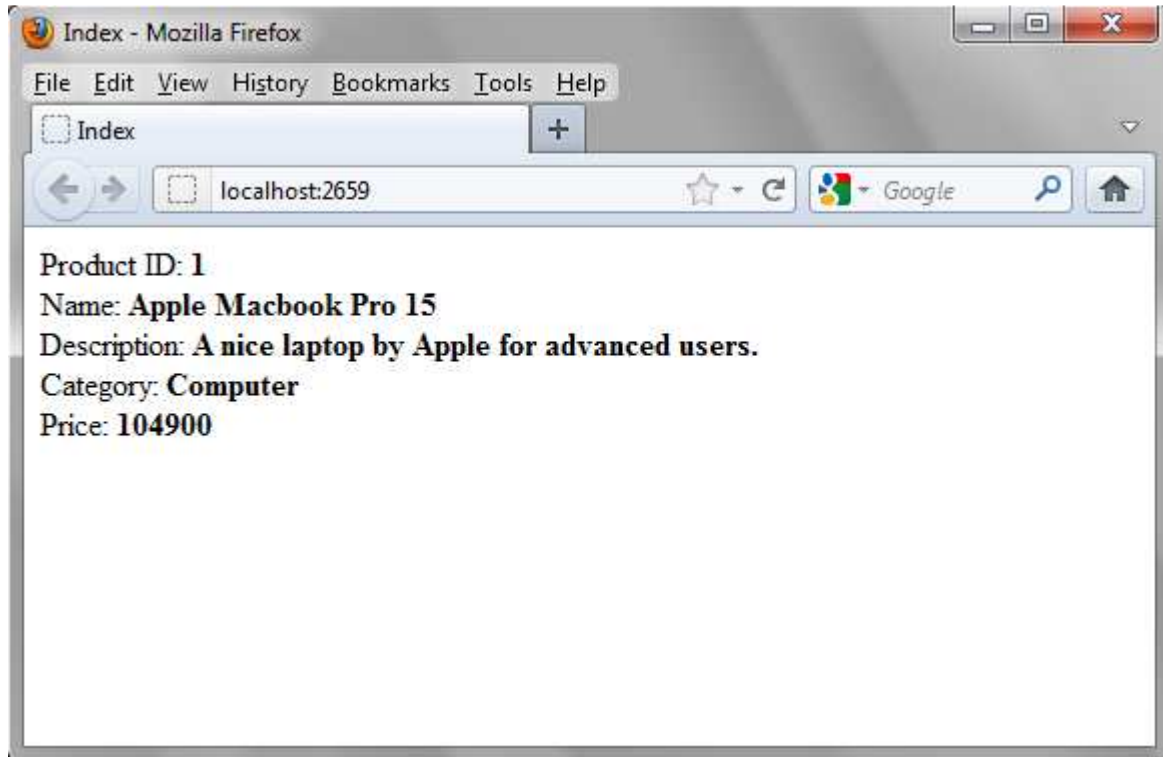
Here we are leaving this null, as we don't want to go beyond the title.

**@Model.ProductID**

`@Model.<propertyname>`

Notice that when we specify the type of the model, we use `@model` (lowercase m), but when we refer to the model object, we use `@Model` (uppercase M).

If you run the above application by selecting Start Debugging from the Visual Studio Debug menu, you see the result shown in screen below.



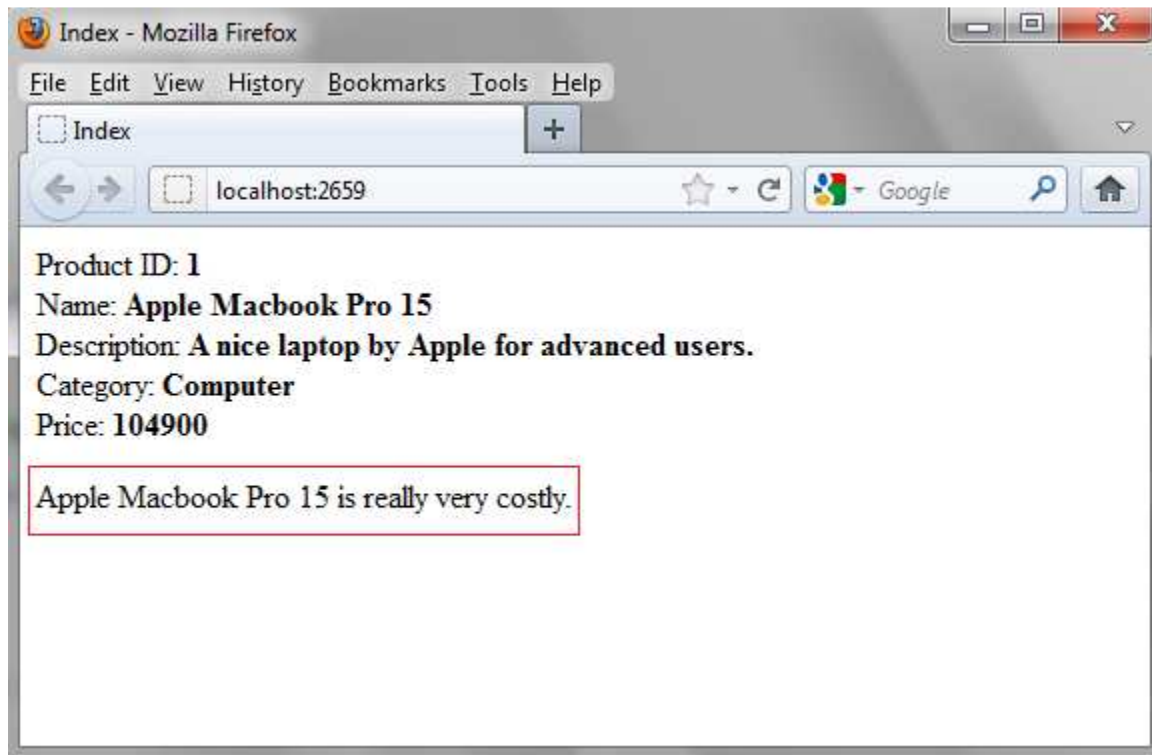
Now, let's dive in Razor conditional statements. Look at the following code.

```
Index.cshtml X
Client Objects & Events
@model DemoMVCApplication.Models.Product
@{
    Layout = null;
}
<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        Product ID: <b>@Model.ProductID</b>
        <br />
        Name: <b>@Model.Name</b>
        <br />
        Description: <b>@Model.Description</b>
        <br />
        Category: <b>@Model.Category</b>
        <br />
        Price: <b>@Model.Price</b>
        <br />
        @if (Model.Price >= 50000)
        {
            <p>@Model.Name is really very costly.</p>
        }
    </div>
</body>
</html>
```

In above code, included an if statement that inserts additional content into the page when the Category property of the Product item is Watersports. Razor is smart enough to recognize that the statement in the if body starts with an HTML tag, and so treats it as markup to be emitted. It also looks for further @ tags, processes them, and puts the results into the web page, here is the output screen.





Now, assume I want to add some words before the outlined text above so that our output looks like.

Laptop Model: Apple Macbook Pro 15 is really very costly.

Then the code given below will not work for you.

```
@if (Model.Price >= 50000)
{
    Laptop Model: <p>@Model.Name is really very costly.</p>
}
```

Above code will not work for you because we can't add our text inside the {} brackets directly. So, to fix this we have couple of ideas in mind, for now let's look at this.

```
<p>Laptop Model: @Model.Name is really very costly.</p>
```

Let's add some complexity in existing conditional statement. Let's look at the code, where I will be using multiple conditions.

```
@model DemoMVCApplication.Models.Product
@{
    Layout = null;
}
<!DOCTYPE html>
```

```
<html>
<head>
  <title>Index</title>
</head>
<body>
  <div>
    Product ID: <b>@Model.ProductID</b>
    <br />
    Name: <b>@Model.Name</b>
    <br />
    Description: <b>@Model.Description</b>
    <br />
    Category: <b>@Model.Category</b>
    <br />
    Price: <b>@Model.Price</b>
    <br />
    @{
      if (Model.Price >= 50000)
      {
        <p>Laptop Model: @Model.Name is really very costly.</p>
      }
      if (Model.Category == "Computer")
      {
        <b>One day I will buy this.</b>
      }
    }
  </div>
</body>
</html>
```

Look at the output screen of the above code.

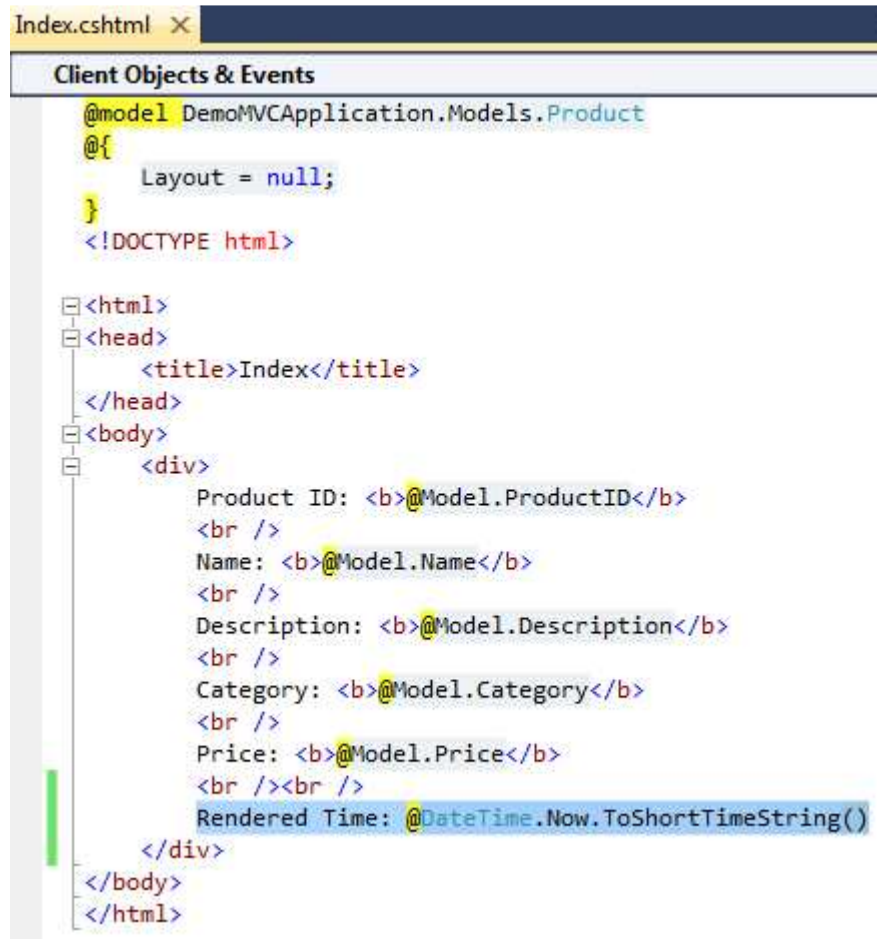


That's all about the Displaying Model Objects using Razor. Now, let's talk on ViewBag.

### ***ViewBag Feature***

Assume, you want to display the rendered time on the view, there is two ways to do this.

First Way

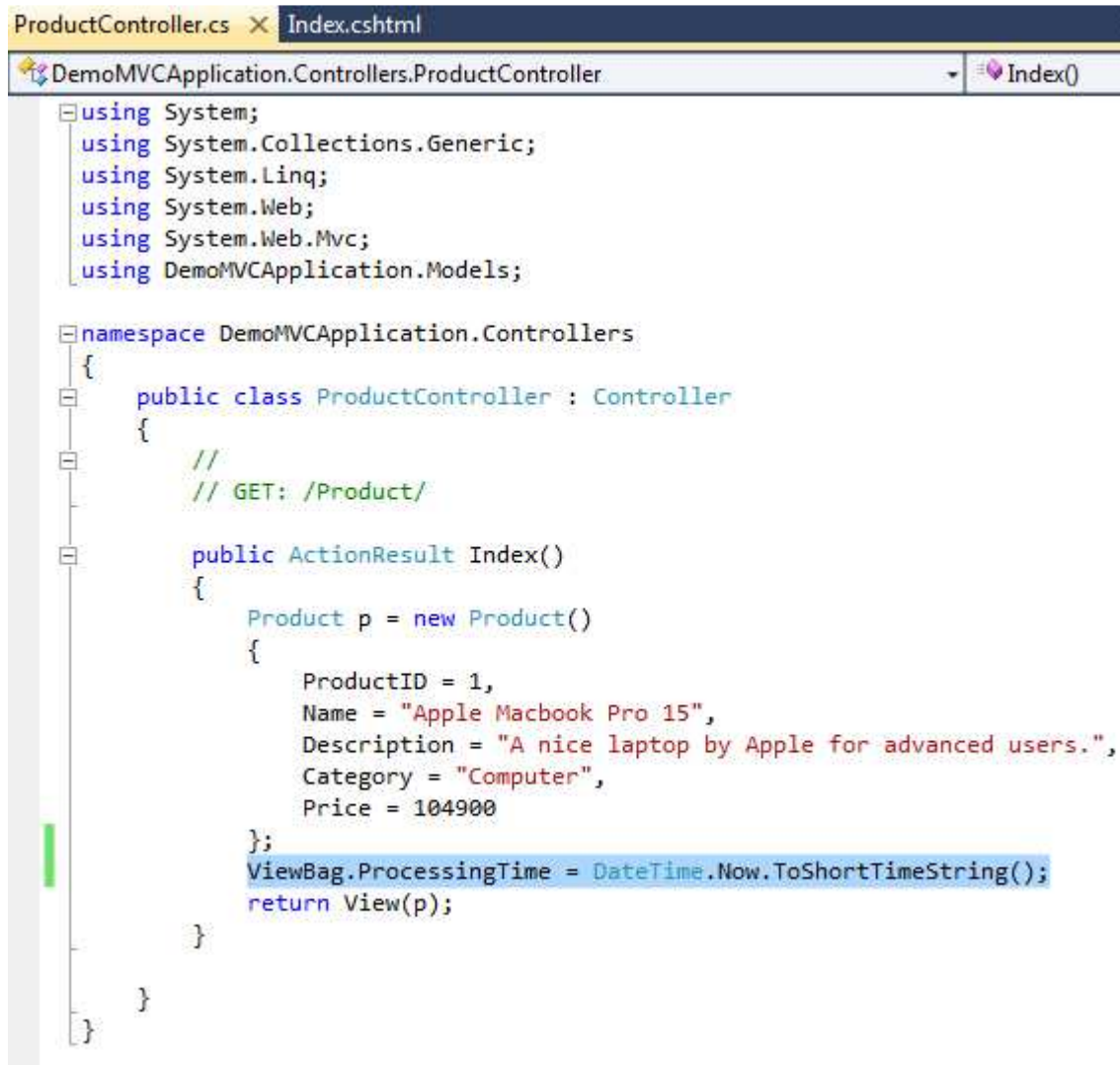


```
Index.cshtml X
Client Objects & Events
@model DemoMVCApplication.Models.Product
@{
    Layout = null;
}
<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        Product ID: <b>@Model.ProductID</b>
        <br />
        Name: <b>@Model.Name</b>
        <br />
        Description: <b>@Model.Description</b>
        <br />
        Category: <b>@Model.Category</b>
        <br />
        Price: <b>@Model.Price</b>
        <br /><br />
        Rendered Time: @DateTime.Now.ToShortTimeString()
    </div>
</body>
</html>
```

This is very simple yet but the same can be done using ViewBag too. For this place the ViewBag in controller and call it from page view.

Find the controller code given below.



```
ProductController.cs X Index.cshtml
DemoMVCApplication.Controllers.ProductController Index()

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using DemoMVCApplication.Models;

namespace DemoMVCApplication.Controllers
{
    public class ProductController : Controller
    {
        //
        // GET: /Product/

        public ActionResult Index()
        {
            Product p = new Product()
            {
                ProductID = 1,
                Name = "Apple Macbook Pro 15",
                Description = "A nice laptop by Apple for advanced users.",
                Category = "Computer",
                Price = 104900
            };
            ViewBag.ProcessingTime = DateTime.Now.ToShortTimeString();
            return View(p);
        }
    }
}
```

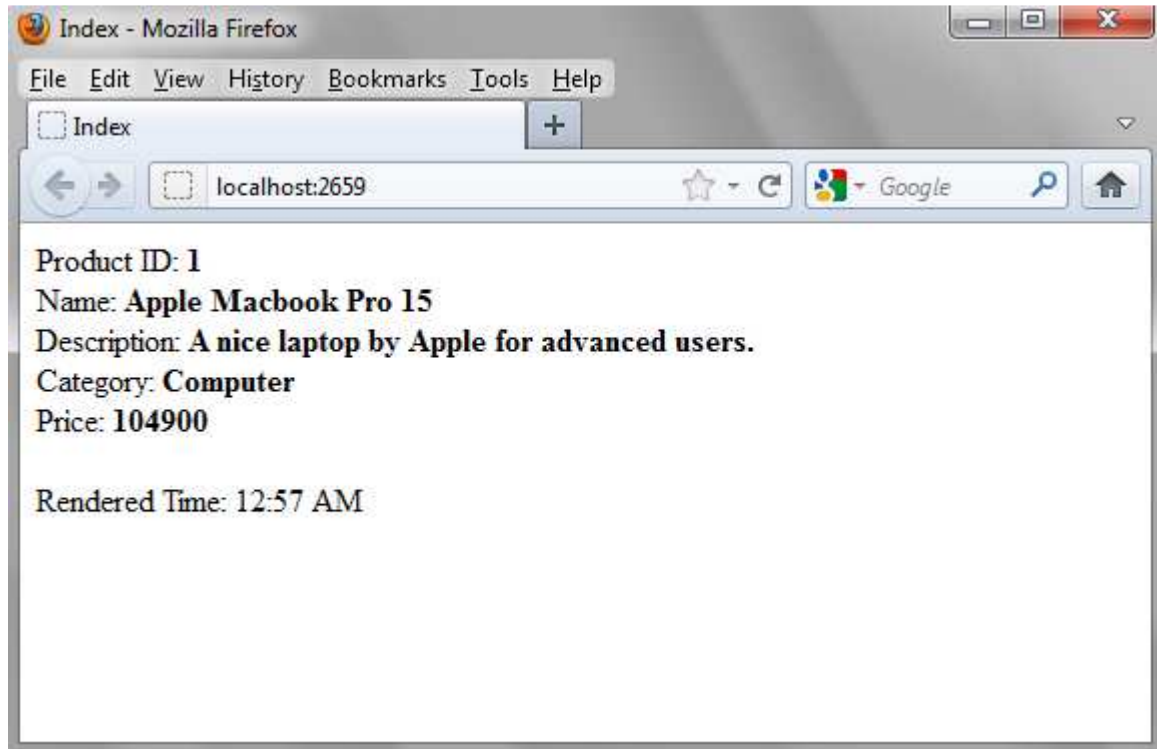
And now to call this ViewBag on View Page, we need to write following code.



```
Index.cshtml X
Client Objects & Events
@model DemoMVCApplication.Models.Product
@{
    Layout = null;
}
<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        Product ID: <b>@Model.ProductID</b>
        <br />
        Name: <b>@Model.Name</b>
        <br />
        Description: <b>@Model.Description</b>
        <br />
        Category: <b>@Model.Category</b>
        <br />
        Price: <b>@Model.Price</b>
        <br /><br />
        Rendered Time: @ViewBag.ProcessingTime
    </div>
</body>
</html>
```

Now, if you run the above View page, it will show the following screen.



There is no any significant advantage in using ViewBag over ViewData, perhaps a few fewer key strokes, but nothing more.

Thanks for reading this book. Please send your valuable comments here.

<http://www.itorian.com/articles/mvc/post/283/>

Cheers!!

**Abhimanyu Kumar Vatsa**