

ASP.NET MVC is the architectural pattern based on ASP.NET framework which provides a clean and elegant way of developing Web application. Before we can understand why MVC is in vogue we need to understand some limitation of Web forms first.

## A Brief Discussion on Web Forms

let us discuss some limitation of web forms which are seen as drawbacks. I am sure many veteran developers will agree that these drawbacks can be put in check by following best practices and guidelines but none the less its a good idea to discuss them before looking at the ASP.NET MVC architecture.

- **ViewState:** This is the heart of ASP.NET when it comes to the ability of ASP.NET to provide stateful abstraction over stateless HTTP protocol. But with great power comes great responsibility. If the ViewState is not managed properly then this would increase the size of the rendered web page a lot and this causing extra load.
- **Generated HTML:** Server side controls are another strong selling points of web forms. They facilitate rapid application development without worrying too much about the client side HTML code. But since the use of client side technologies are increasing (javascript, jQuery etc.), the less control over generated markup is becoming bit of a problem.
- **Urls:** In ASP.NET web forms the request URL points to physical files and the structure of the URL is not SEO friendly. Also, now is the time when URL structure is very important and clean URLs and control over URLs is very much desired.
- **Code Behind:** Code behind model, in my opinion, is the best part of ASP.NET. It provides clear separation of concern. From a developers perspective having the code behind file provides a great way to write the server side logic without having to write code between the HTML markup.
- **Page Life Cycle:** Although code behind model provides a great way for providing separation of concern, the page life cycle is little complex and should be fully understood. If a developer fails to understand the page life cycle then there might be some unforeseen and undesired issues in the final application.
- **Test Driven Development support:** Now is the time of being agile. When we follow agile methodology (scrum specifically), it is very important to define the "Done" in scrum. TDD helps in this. If all our test cases are passed then we can be sure that the "Done" is done. ASP.NET does not provide native support for TDD because it is kind of difficult to imitate the HTTP request and HTTP context.

Having said that(all the above points), I would also like to point that question of limitations of web forms is becoming blurred with every new release of ASP.NET web forms. A lot of these limitations are being curbed in Web forms. **ASP.NET 4.0** added URL Routing, reduced **ViewState**, and a much better control of the HTML mark-up. A lot of other issues related to leaky abstractions in web forms can be solved by following best practices and design guidelines. And the TDD can be performed with the help of some tool.

So it is not a question of which is better. ASP.NET Web Forms and MVC are two different architectural styles. Forms focusing on rapid application development (and now getting a lot better

with every new release). and MVC is for designing Web applications without the baggage that comes with forms and following good patterns.

A lot of developers think that MVC is the new way of developing web application and Web forms is dead but strictly in my personal opinion, they both are two different architectural styles and no one supersedes other. We should choose the one based on our requirements and resources available. In fact the possibility of being able to mix both the styles is the best thing. We can use both these styles in a single application and get the best of both worlds.

## A look at MVC

MVC framework embraces the fact that the HTTP is stateless and thus no stateful abstraction will be provided by the framework. It is up to the developers to take care of managing the state in a MVC application.

In MVC architecture there are three major player. **Model**, **View** and **Controller**. we need to work on these creating these components to get our web application to work. Now before looking at these three in detail let us try to think how we can put server side contents in a HTML page. We will take a rather reverse approach to understand this.

We need to put some server side data into an HTML markup before rendering it. Now this can easily be done in Web forms too where we used to put C# code in aspx markup. The only prerequisite for that is that we should have some object containing the value on the server side from which we can extract the data. That is what views does in MVC. they simply run and before rendering the markup they use some server side object to extract the data and create the complete HTML markup and render on client browser.

Now lets take a step back again, Now we need an object on server side. In MVC world this is model. We need an instantiated model and pass it to the view so that views can extract the data from this object and create the final HTML markup.

Now the question would be, who would instantiate these Models. These models will be instantiated in the controllers. Controller will instantiate the models and then pass the model to the view so that the view can use them to generate the markup.

But now the bigger question, How would the controller be invoked The controller will be invoked on the user request. All the user requests will be intercepted by some module and then the request URL will be parsed and based on the URL structure an appropriate controller will be invoked.

So now that we know the basic flow let us try to define the Model, View and Controller formally.

- **Model**: These are the classes that contain data. They can practically be any class that can be instantiated and can provide some data. These can be entity framework generated entities, collection, generics or even generic collections too.

- **Controllers:** These are the classes that will be invoked on user requests. The main tasks of these are to generate the model class object, pass them to some view and tell the view to generate markup and render it on user browser.
- **Views:** These are simple pages containing HTML and C# code that will use the server side object i.e. the model to extract the data, tailor the HTML markup and then render it to the client browser.

## A Look at MVC Request Response process

Now we already saw the MVC request life cycle but quite in the reverse direction. Let us try to see this formally.

1. User sends the request in form of URL.
2. **UrlRoutingModule** intercepts the request and starts parsing it.
3. The appropriate Controller and handler will be identified from the URL by looking at the **RouteTable** collection. Also any data coming along with the request is kept in **RouteData**.
4. The appropriate action in the identified controller will be executed.
5. This action will create the Model class based on data.
6. The action will then pass on this created model class to some view and tell the view to proceed.
7. Now the view will then execute and create the markup based on the logic and Model's data and then push the HTML back to the browser.

This life cycle above is defined for explanation and has omitted some technical details which involved a few more objects (like controller base class). Once the idea of MVC is understood I would recommend to dig deeper into the Request life cycle.

## A Sneak Peak at Routing

Now we have said that the controller and action will be decided by the URL. So it is perhaps a good idea to look at a very simple example of this **RouteTable** entry so that we know what pattern of URL will invoke which controller and which action.

Here is the default entry in the **RouteTable** which is made in **global.asax** application\_start event.

 [Collapse](#) | [Copy Code](#)

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

Here in this above route, the name for the route is specified as Default. The second argument specifies the pattern of the URL that should lead to this route usage i.e. Any URL with pattern **"SiteAddress/{controller}/{action}/{id}"** will use this route. The final argument specifies that if this route will be used then a controller named Blog and an action Index will be invoked. id is optional meaning it could be present or not.

So the following URLs will invoke the respective actions and controllers.

**Url:** www.siteaddress.com

**Controller:** Blog

**Action:** Index

**Url:** www.siteaddress.com/Blog

**Controller:** Blog

**Action:** Index

**Url:** www.siteaddress.com/Blog/Create

**Controller:** Blog

**Action:** Create

**Url:** www.siteaddress.com/Blog/Delete/1

**Controller:** Blog

**Action:** Delete

**Id:** 1

**Url:** www.siteaddress.com/Category/Edit/1

**Controller:** Category

**Action:** Edit

**Id:** 1

So the above examples would perhaps make it clear that what URL will invoke which controller and which action. We can also define custom routes too. I am not talking about creating custom routes in this article but once routing is understood creating custom routes is fairly straight forward.

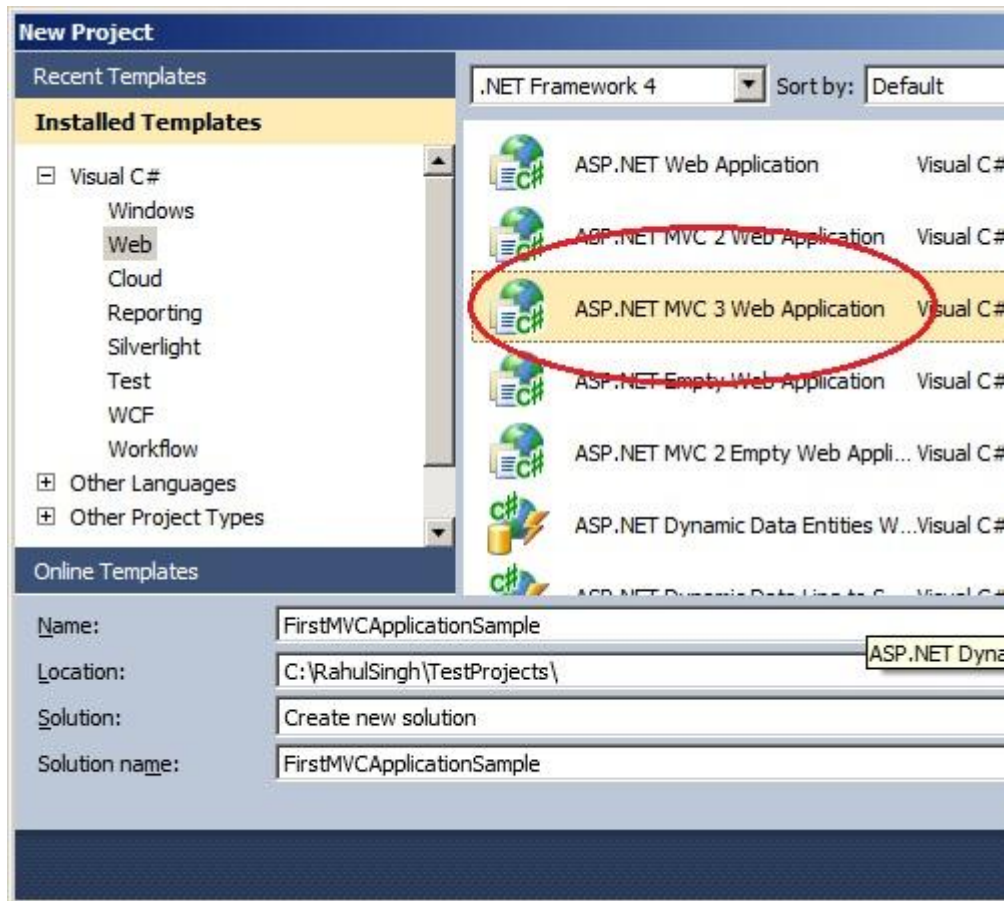
## Using the code

Now we have seen some theory behind MVC architecture. Let us now create a simple application that will be invoked on a specific URL. The controller will be invoked and a Model class will be created. Then this Model will be passed to the view and the resulting HTML will be displayed on the browser.

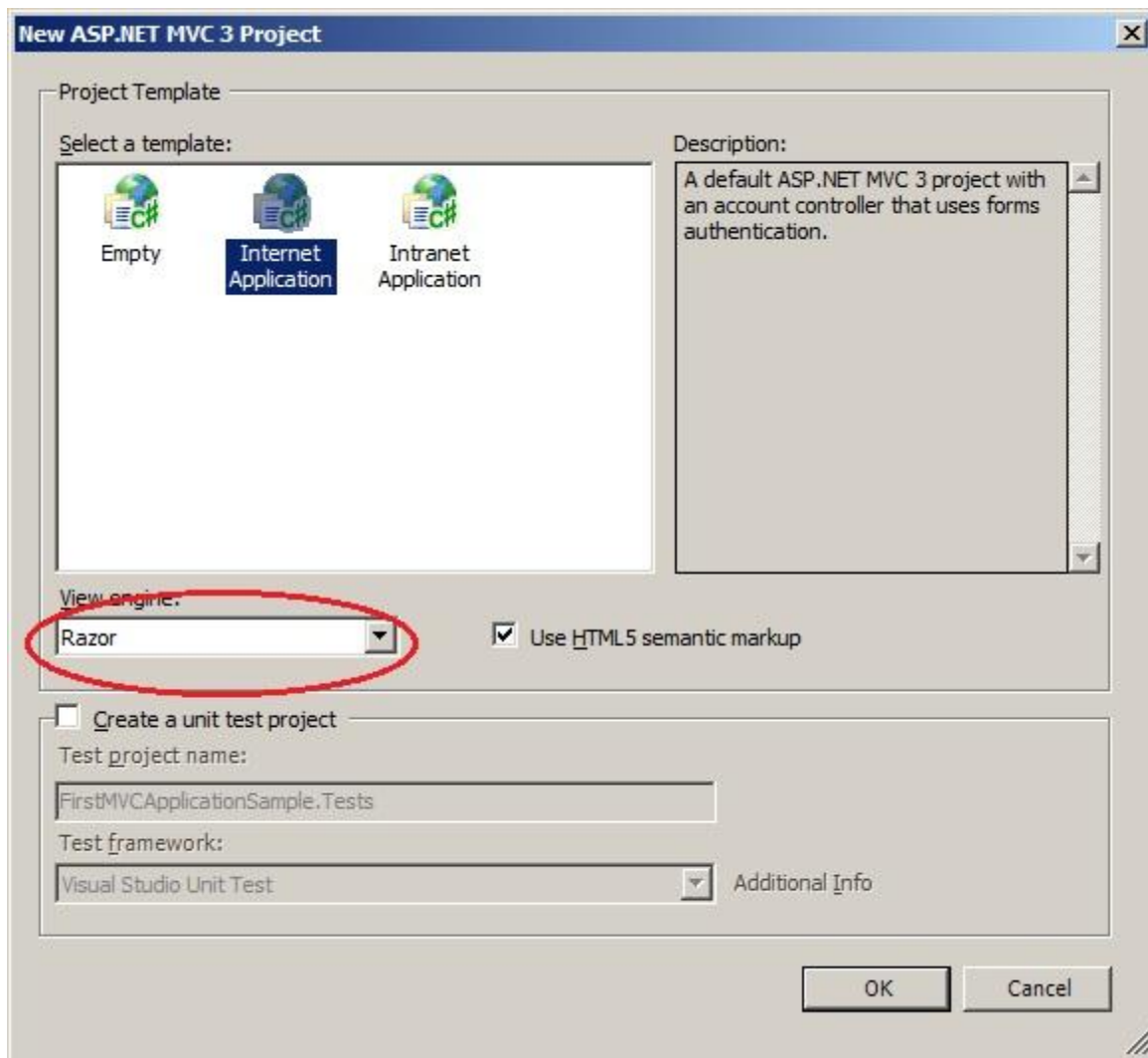
Now the creation of **Controllers**, **Models** and **Views** need some code to be written by the developer. Visual studio also provides some standard templates that provides several scaffolds out of the box. Now since this tutorial is mainly to look at the overall architecture of MVC and understanding how we can create our first MVC application we will use these templates and scaffolding. But it is highly recommended that some good book on MVC should be referred to get the full understanding of these.

**Note:** We will be creating an MVC 3.0 application and using Razor view engine for this exercise.

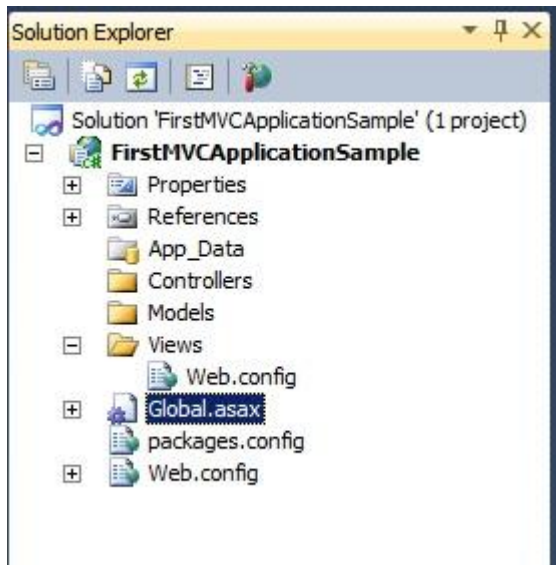
Let us go ahead and create a new MVC 3 Web Application.



Upon asked, let us select the Razor View Engine.



Now this template comes with a lot of predefined Controllers, models and view. Let us delete all this and just have three Folders named Models, Views And Controllers in this application.



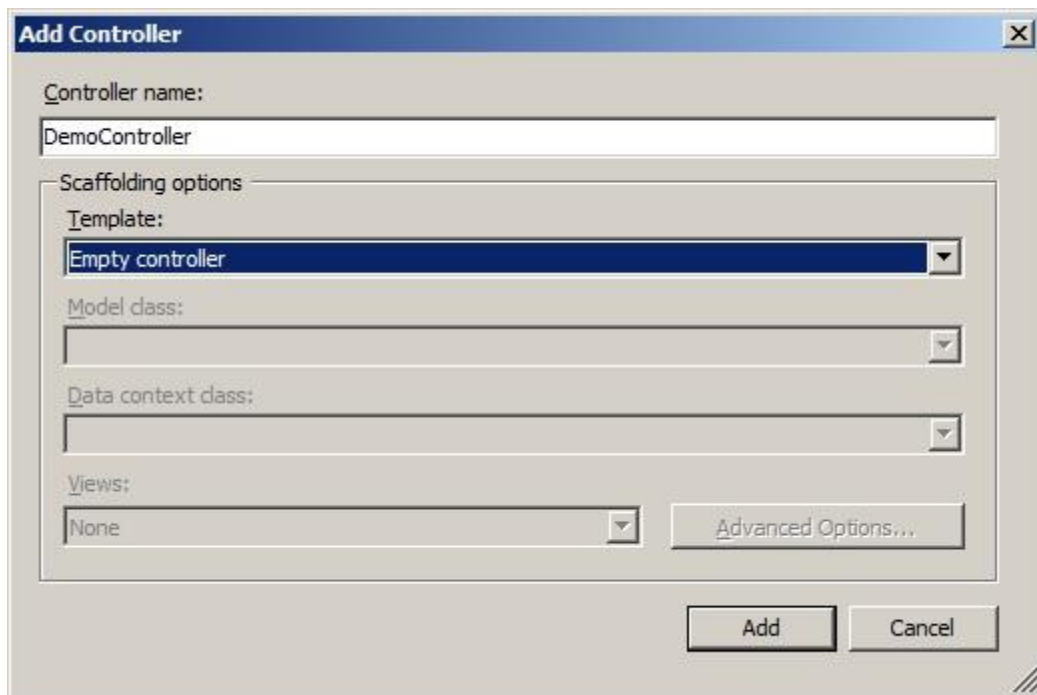
Now let us start by defining our own route in the **global.asax**. Now the Controller that we want to access by default will be **DemoController**. We want the Index action of this controller to execute by default. We will also pass an ID which, if present will also be displayed on the page. So the route defined for this in the **global.asax** will look like:

[Collapse](#) | [Copy Code](#)

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

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

And now Let us add a Controller named Demo to this application. This can be done by right clicking the controller's folder and selecting Add Controller. Name this Controller as **DemoController**.



Now this controller comes with a default action Index. This action will be called if no action is specified for this controller. Let us show a simple message if this controller is invoked. We will use **ViewBag** for this. **Viewbag** is an object that is accessible in the View page and it should be populated in the Controller.

[Collapse](#) | [Copy Code](#)

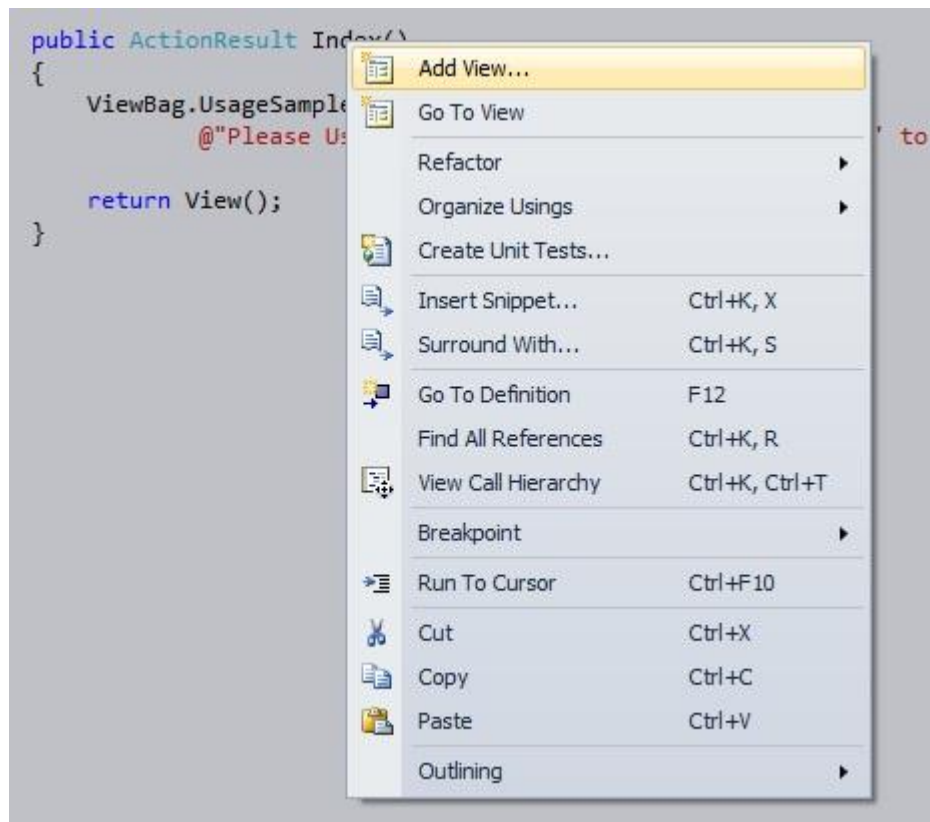
```
public ActionResult Index()
{
    ViewBag.UsageSample =
        @"Please Use the Url 'Demo/Test' or 'Demo/Test/1' to see the Book Details";

    return View();
}
```

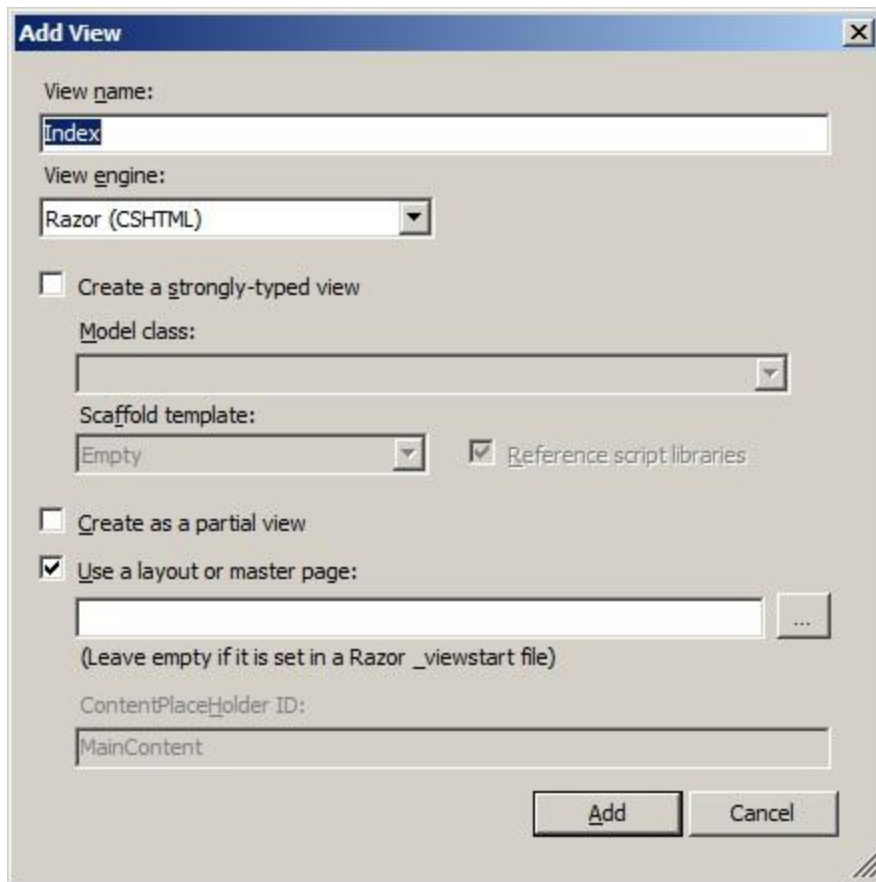
**Note:** We are not taking about the data transfer techniques like **ViewBag**, **ViewData** and **TempData** in this article. that topic requires some detailed discussion. We will perhaps discuss them later.

Now this code will simply put a string in the view bag and call the View. The return statement will expect a view named index present in a folder named Demo. We don't have that right now so let us create this view by right clicking the method name as:





This will then ask for more options for the view we need to add. For now lets not select any other option and just create the view.

The 'Add View' dialog box is shown with the following settings: View name: 'Index'; View engine: 'Razor (CSHTML)'; 'Create a strongly-typed view' is unchecked; Model class: empty; Scaffold template: 'Empty'; 'Reference script libraries' is checked; 'Create as a partial view' is unchecked; 'Use a layout or master page' is checked; ContentPlaceHolder ID: 'MainContent'. The 'Add' button is highlighted.

**Add View**

View name:  
Index

View engine:  
Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:  
Empty ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:  
...  
(Leave empty if it is set in a Razor \_viewstart file)

ContentPlaceHolder ID:  
MainContent

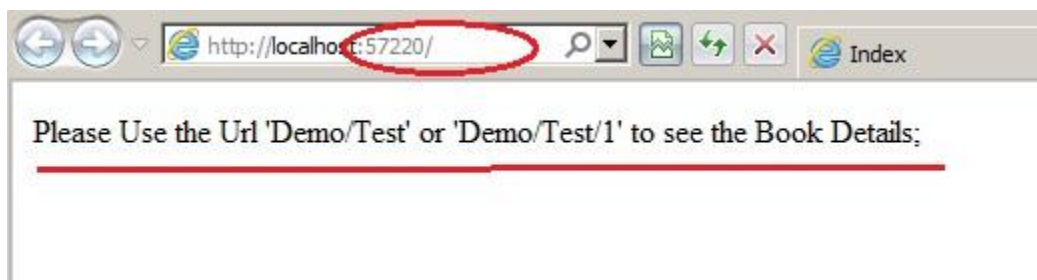
Add Cancel

Now in the view we will write the logic of extracting the data from **ViewBag** and showing it on the page.

[Collapse](#) | [Copy Code](#)

```
<div>  
    @ViewBag.UsageSample;  
</div>
```

And now if we try to run this application:



And so we have seen how the controller gets invoked from the the user request and then controller passes some data to the view and the view renders it after processing. But we have not yet added any model to it. Let us add a simple Model in this application. Let us define a simple model for a **Book** class in the models folder. We will then see how we can use instantiate this model and use this in our View.

 [Collapse](#) | [Copy Code](#)

```
public class Book
{
    public int ID { get; set; }
    public string BookName { get; set; }
    public string AuthorName { get; set; }
    public string ISBN { get; set; }
}
```

Now let us add an action called Test in this **DemoController** and create an object of this **Book** Model. Once the object is created we need to pass this object in the view as:

 [Collapse](#) | [Copy Code](#)

```
public ActionResult Test()
{
    Book book = new Book
    {
        ID = 1,
        BookName = "Design Pattern by GoF",
        AuthorName = "GoF",
        ISBN = "NA"
    };

    return View(book);
}
```

the above code will create the model and pass it to the view. Now let's create a view that is capable of extracting the data from this model. This can be done by creating a strongly typed view as:

**Add View**

View name:  
Test

View engine:  
Razor (CSHTML)

☒ Create a strongly-typed view

Model class:  
Book (FirstMVCApplicationSample.Models)

Scaffold template:  
Details

☒ Reference script libraries

☐ Create as a partial view

☐ Use a layout or master page:  
(Leave empty if it is set in a Razor \_viewstart file)

ContentPlaceHolder ID:  
MainContent

Add Cancel

I have also chosen the scaffold template as "**Details**" so that all the boilerplate code in the view page to display these details is already generated.

```

@model FirstMVCApplicationSample.Models.Book

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Test</title>
</head>
<body>
    <fieldset>
        <legend>Book</legend>

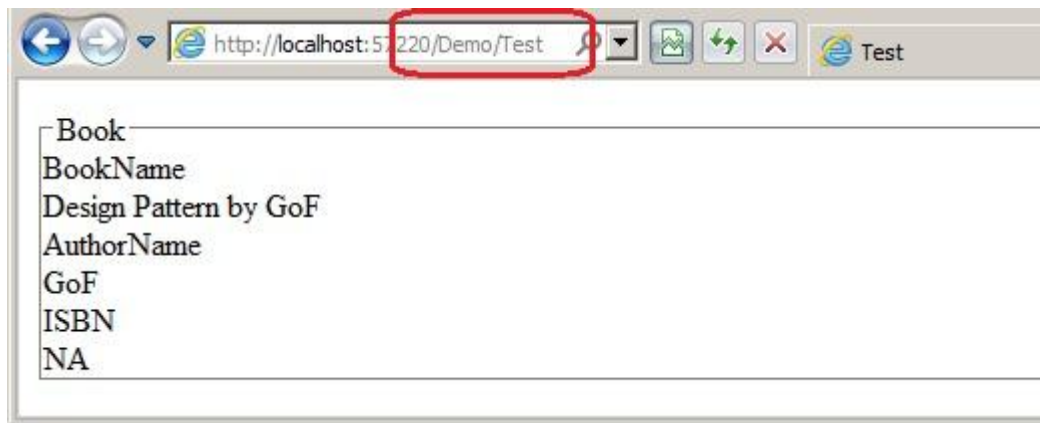
        <div class="display-label">BookName</div>
        <div class="display-field">
            @Html.DisplayFor(model => model.BookName)
        </div>

        <div class="display-label">AuthorName</div>
        <div class="display-field">
            @Html.DisplayFor(model => model.AuthorName)
        </div>

        <div class="display-label">ISBN</div>
        <div class="display-field">
            @Html.DisplayFor(model => model.ISBN)
        </div>
    </fieldset>
    <p>
        @Html.ActionLink("Edit", "Edit", new { id=Model.ID }) |
        @Html.ActionLink("Back to List", "Index")
    </p>
</body>
</html>

```

And now we run the code we can see the details of the book which were extracted from the Model which was created in the controller, which was executed on parsing user's request URL.



And thus we have our very first MVC 3.0 application with razor view engine working.