

# ASP.NET MVC



Authorized & published by Summitworks Technologies Inc

# Agenda

- Introduction to MVC
  - Understanding Model, View and Controller
  - Understanding MVC request life cycle
  - Separation of Concerns, Extensible
  - Understanding the MVC Design pattern
- Creating a MVC Application
  - First controller
  - First View
- Working with Url's and Routing
  - About Routing
  - Understanding the Routing System
  - Adding a Route Entry
  - Using Defaults
  - Using Parameters
  - Using Constraints
- HTML Helpers
  - HTML helper methods
  - Render HTML Form
  - Binding HTML helper to Model
- Working with Controllers
  - Understanding Controllers
  - Passing Data from Controller to View
  - Comparing View Bag, View Data, and Temp Data
  - Types of Action methods
  - Action Method Parameters
  - Types of Action Result
    - View Result
    - Partial View Result
    - Redirect Result
    - Redirect to Action Result
    - Redirect to Route Result
    - Json Result
    - File Result
    - Content Result

# **Introduction to Model-View-Controller (MVC)**

# What is ASP.NET MVC?

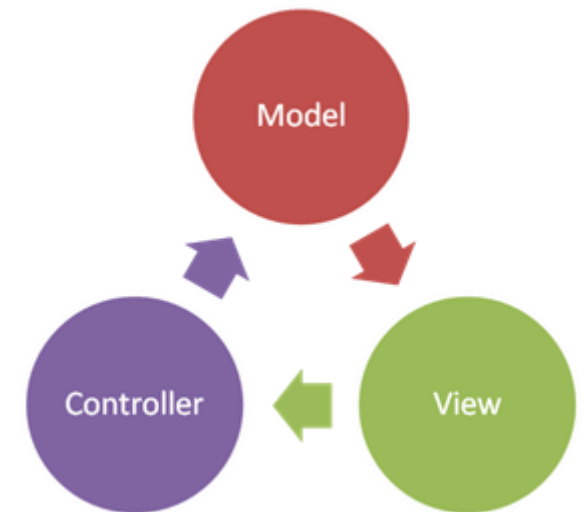
- MVC is a Web development Framework that runs on top of ASP.NET
- It is not a replacement for WebForms
- Just like WebForms, MVC can leverage the benefits of ASP.NET features like caching, modules, handlers, sessions states, query strings and etc.

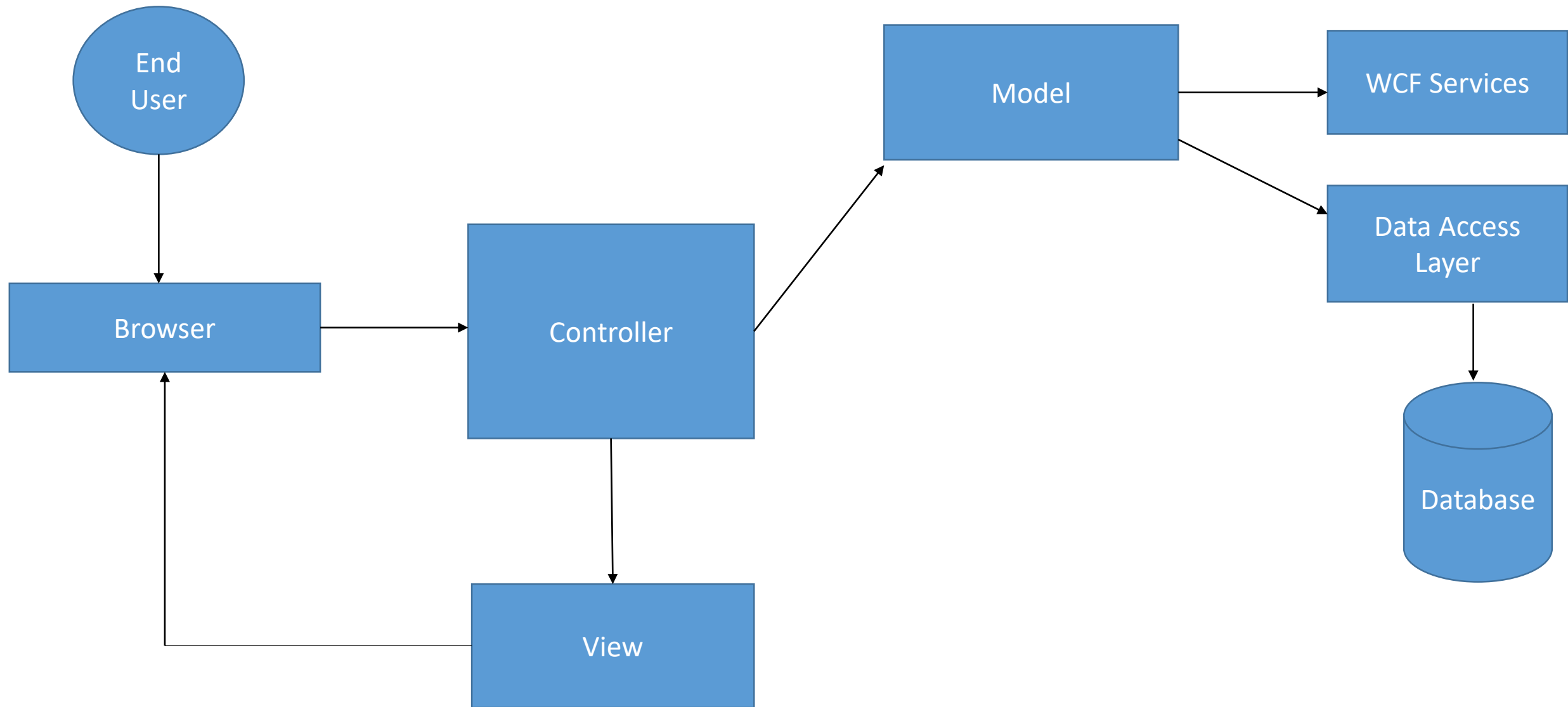
# Problems with WebForms

- View State Weight: slower response times and increasing the bandwidth demands of the server
- Page life cycle: extremely complicated
- Heavy abstraction: Every code is generated with the help of controls.
- Limited control over HTML: Server controls render themselves as HTML, but not necessarily the HTML you want.
- Low testability: The tightly coupled architecture of ASP.NET WebForms is unsuitable for unit testing

# Introduction to MVC

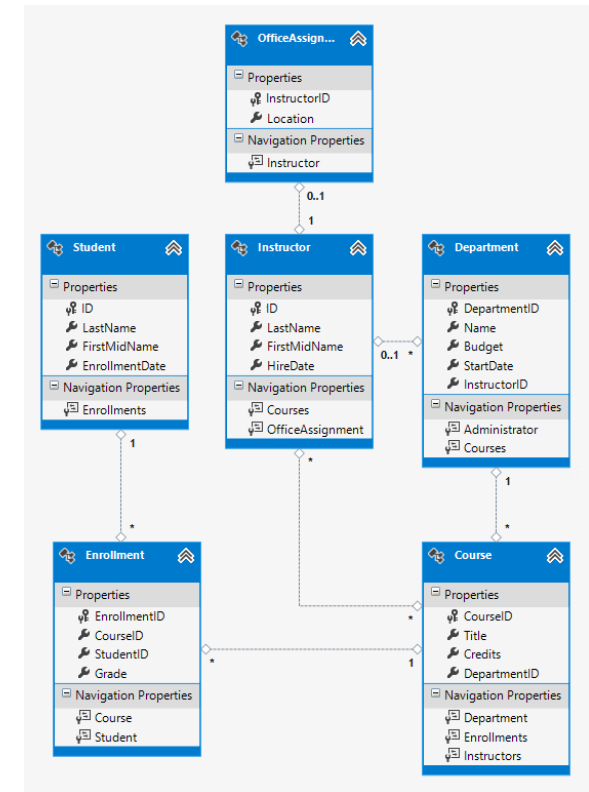
- MVC is another pattern used for web application development just like Web forms.
- MVC is an architectural pattern which separates the representation and user interaction.
- Model represents the real world object and provides data to the View.
- The view is responsible for the look and feel of the application.
- The controller is responsible for taking the end user request and loading the appropriate model and view.





# Model in MVC

- In real-life situations, data that comes from the user doesn't go straight into the database. It must often be validated, filtered, or transformed.
- The role of the model layer is usually to make sure that data arrives properly into the backend store, usually the database by performing these operations, which should not be the responsibility of the controller and not the responsibility of the database engine either.
- In other words, the Model layer is responsible for how the data should be handled.





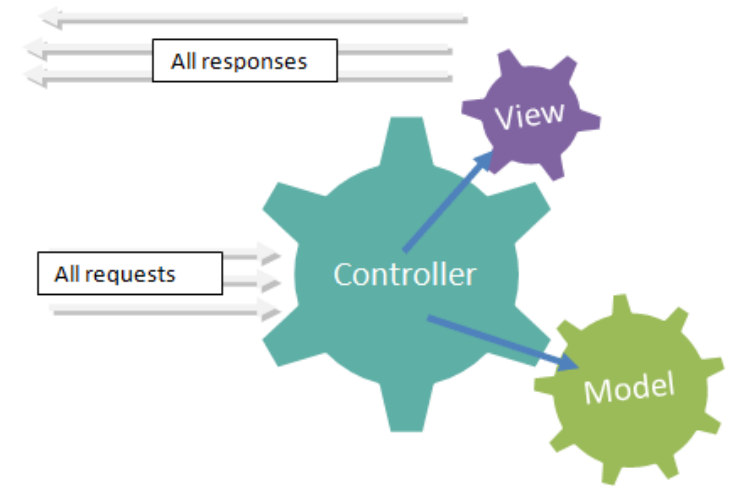
# Views in MVC

- In the Model-View-Controller (MVC) pattern, views are intended exclusively for encapsulating presentation logic.
- They should not contain any application logic or database retrieval code. All application logic should be handled by the controller.
- A view renders the appropriate UI by using the data that is passed to it from the controller. This data is passed to a view from a controller action method by returning a view.
- Views located inside this Shared folder will be available to all the controllers.



# Controllers in MVC

- A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen.
- It provides means for user output by presenting the user with menus or other means of giving commands and data.
- The controller receives user output, translates it into the appropriate messages and pass these messages on to one or more of the views.
- So basically, controllers in MVC are used to handle the business logic.

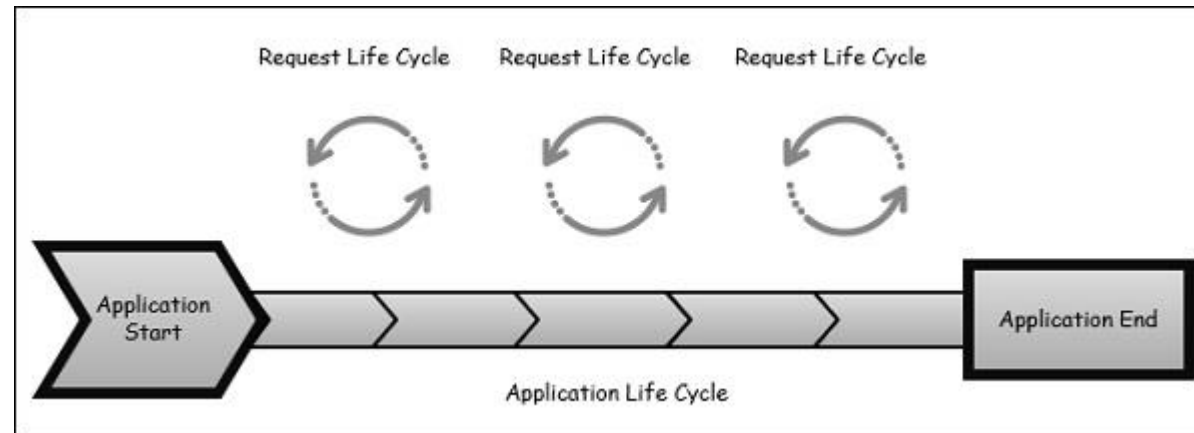


# MVC request life cycle

At a high level, a life cycle is simply a series of steps or events used to handle some type of request or to change an application state.

MVC has two life cycles:

- *The application life cycle*
- *The request life cycle*



# MVC request life cycle

## The Application Life Cycle

The application life cycle refers to the time at which the application process actually begins running IIS until the time it stops. This is marked by the application start and end events in the startup file of your application.

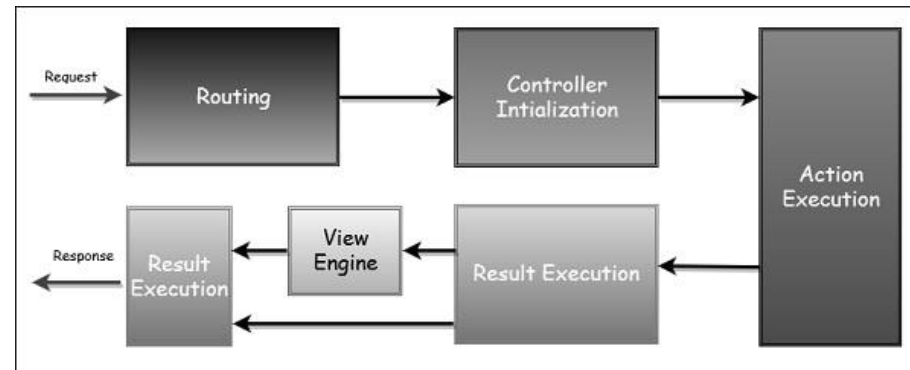
## The Request Life Cycle

It is the sequence of events that happen every time an HTTP request is handled by our application.

The entry point for every MVC application begins with routing. After the ASP.NET platform has received a request, it figures out how it should be handled through the URL Routing Module.

Modules are .NET components that can hook into the application life cycle and add functionality. The routing module is responsible for matching the incoming URL to routes that we define in our application.

All routes have an associated route handler with them and this is the entry point to the MVC framework.



# Separation of Concerns

*“The process of breaking a computer program into distinct features that overlap in functionality as little as possible”.*

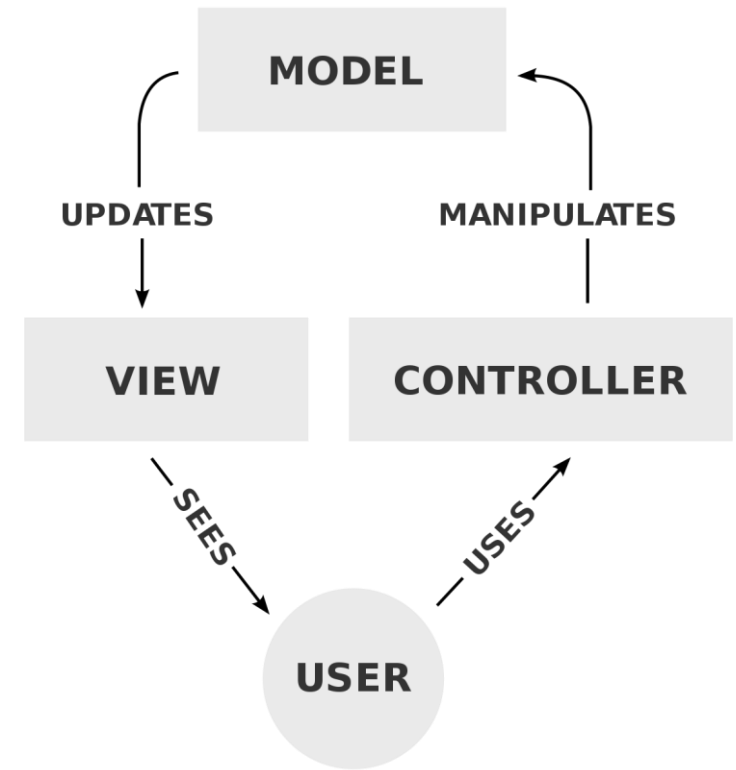
The purpose of the MVC design pattern is to separate content from presentation and data-processing from content. Theoretically well, but where do we see this in MVC? One is reasonably clear; between the data-processing (Model) and the rest of the application.

When we talk about Views and Controllers, their ownership itself explains separation. The Views are just the presentation form of an application, it does not need to know specifically about the requests coming from the Controller. The Model is independent of Views and Controllers, it only holds business entities that can be added to any View by the Controller as per the needs for exposing them to the end user. The Controller is independent of Views and Models, its sole purpose is to handle requests and then on as per the routes defined and as per the needs of the rendering Views. Thus our business entities (Model), business logic (Controllers) and presentation logic (Views) lie in logical/physical layers independent of each other.

# MVC Design Pattern

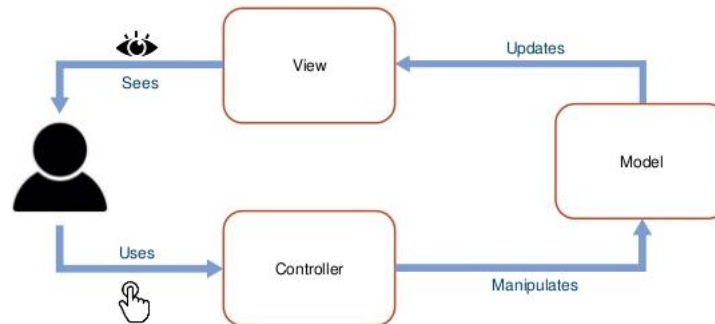
The **Model View Controller** (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application. You're still going to need business logic layer, maybe some service layer and data access layer.



# MVC Design Pattern

- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.



# MVC Design Pattern

## Advantages

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Models can have multiple views.

## Disadvantages

- The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Knowledge on multiple technologies becomes the norm. Developers using MVC need to be skilled in multiple technologies.



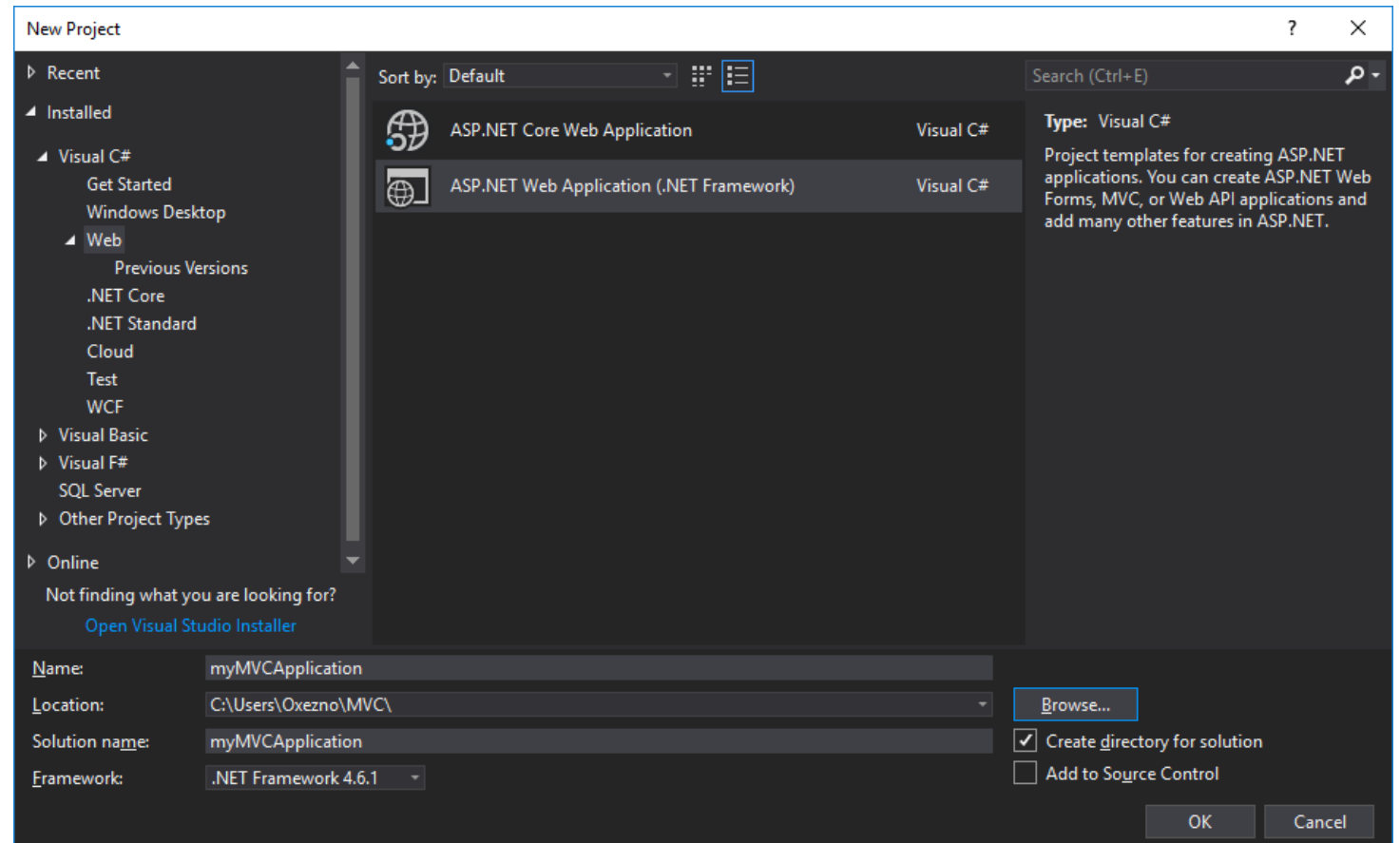
# Creating a MVC Application

# Create a MVC Application

**Step 1:** Start your Visual Studio and select:

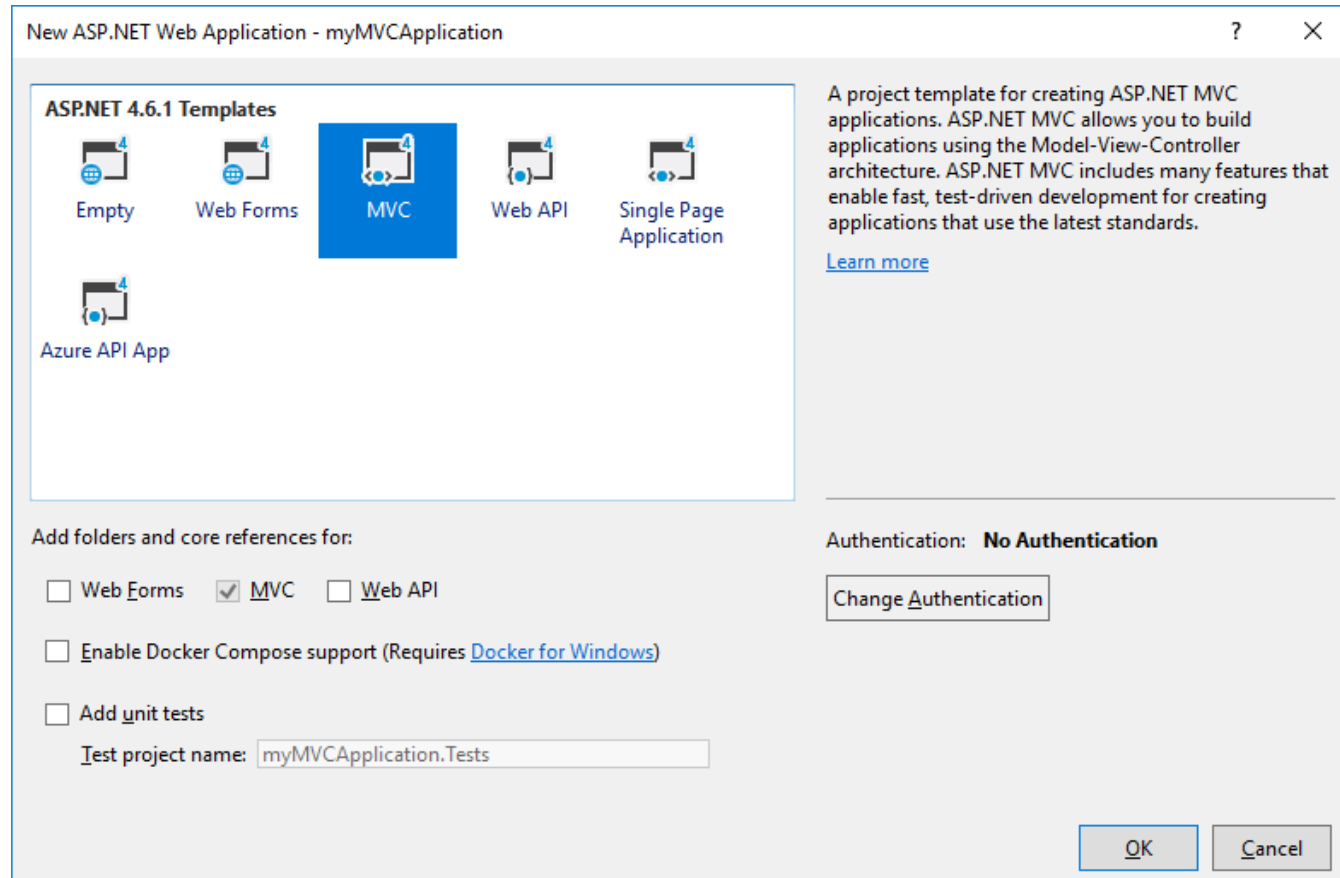
*File → New → Project. Select Web → ASP.NET Web Application (.NET Framework)*

Name this project as **myMVCAApplication**.



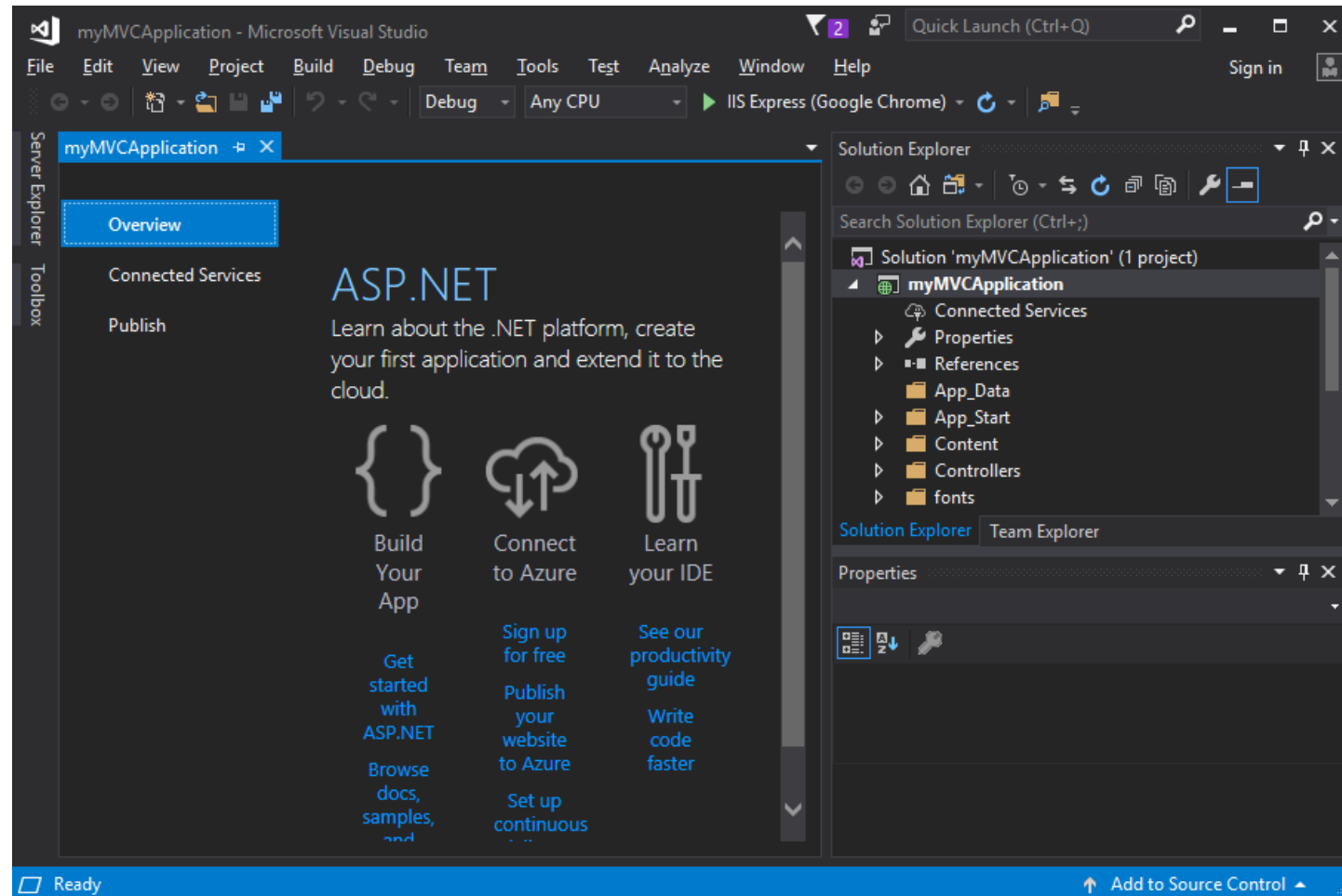
# Create a MVC Application

**Step 2:** Select MVC Template. Click OK.



# Create a MVC Application

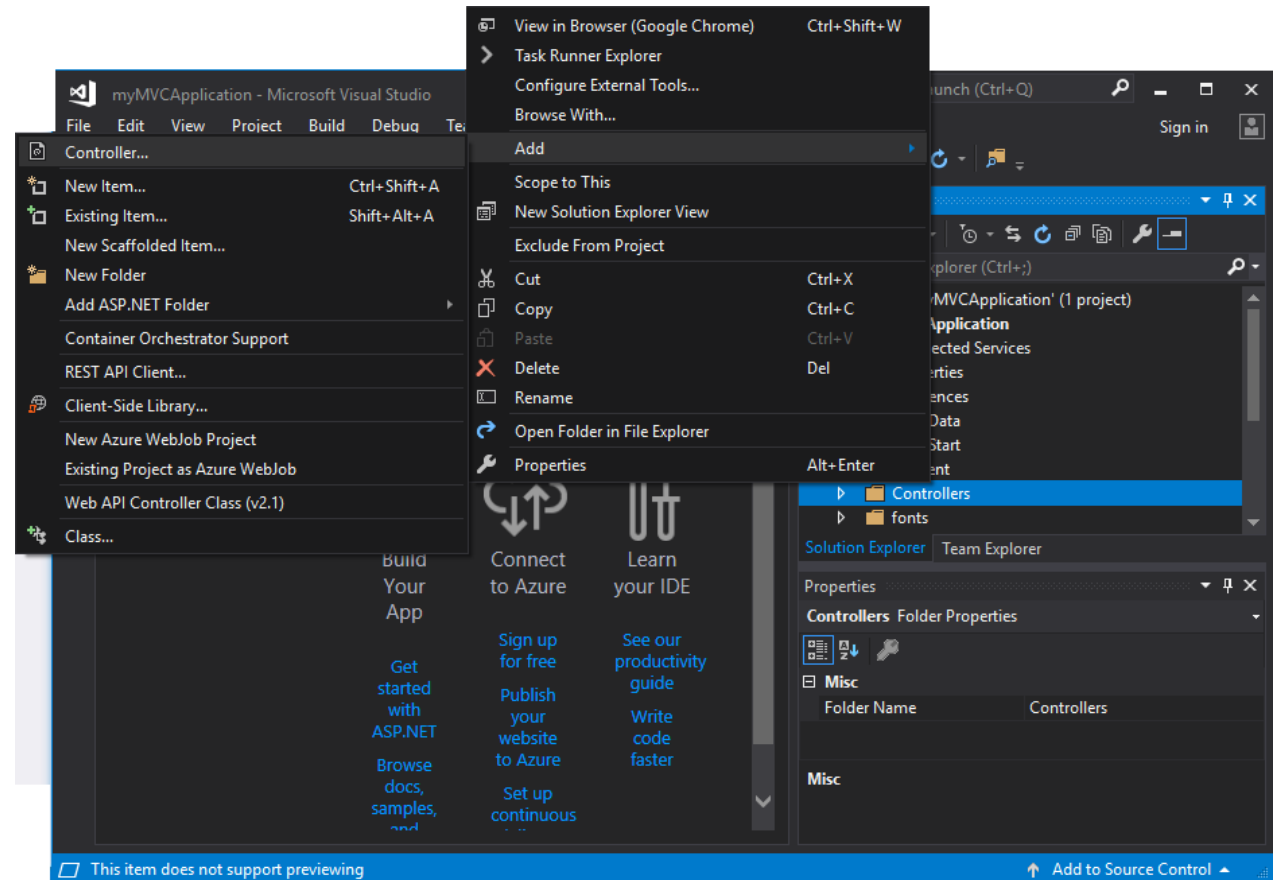
Now, Visual Studio will create our first MVC project.



# Create a MVC Application

**Step 3:** Now we will create the first Controller in our application. Controllers are just simple C# classes, which contains multiple public methods, known as action methods. To add a new Controller, right-click the Controllers folder in our project and select

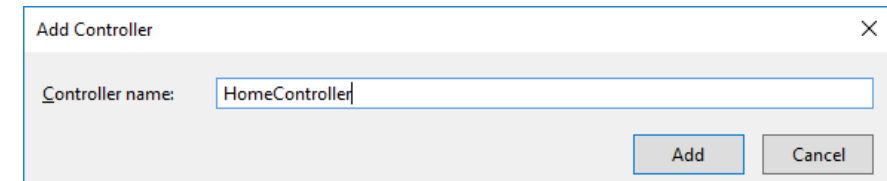
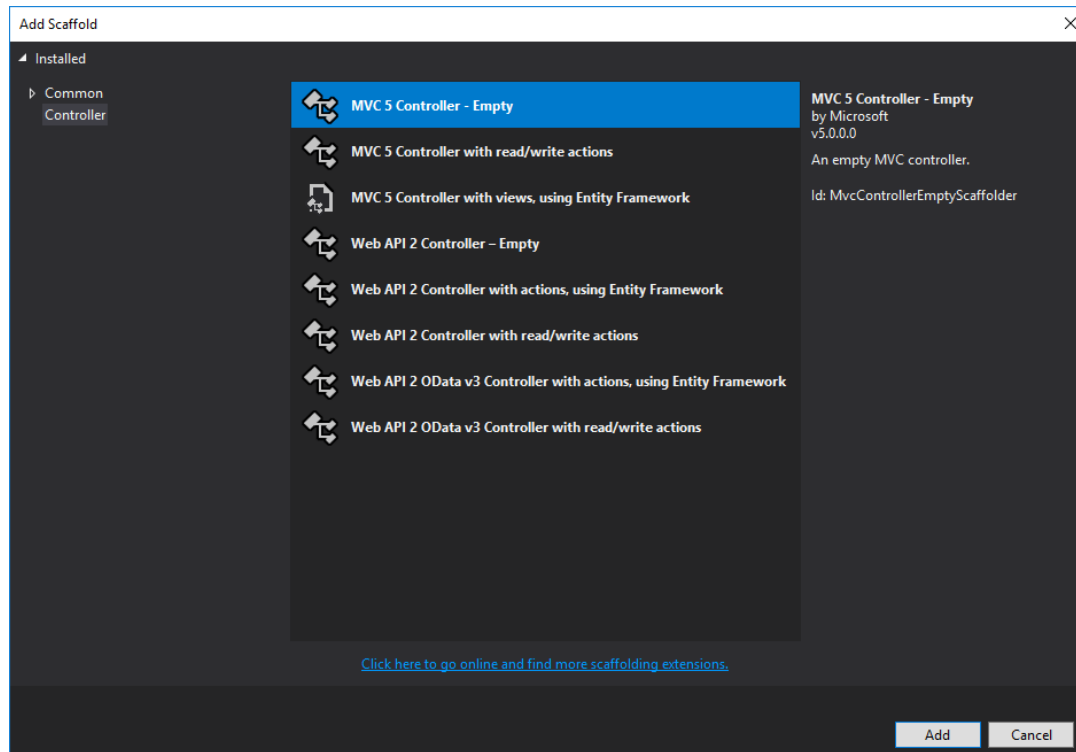
*Add → Controller.*



# Create a MVC Application

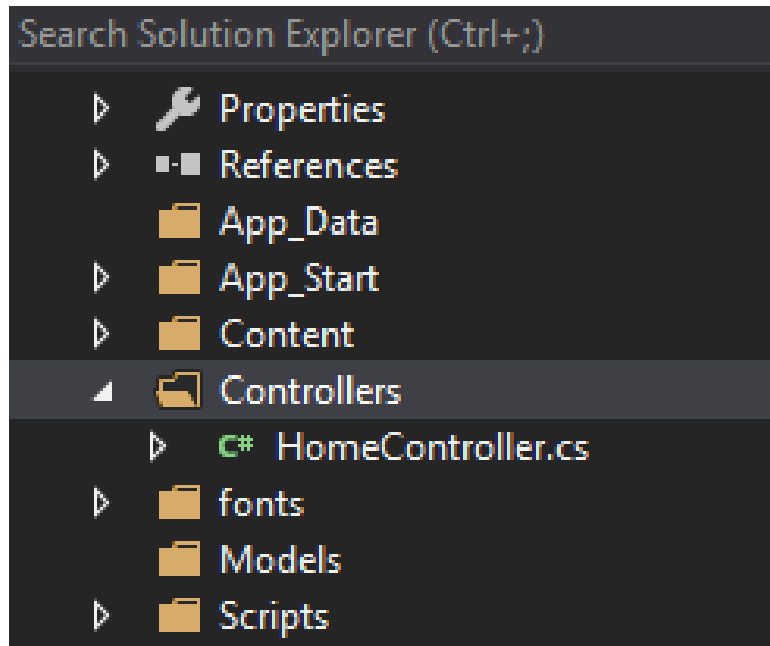
**Step 4:** Select MVC 5 Controller – Empty. Click add

**Step 5:** Name the Controller as HomeController.



# Create a MVC Application

This will create a class file **HomeController.cs** under the Controllers folder with the following default code.



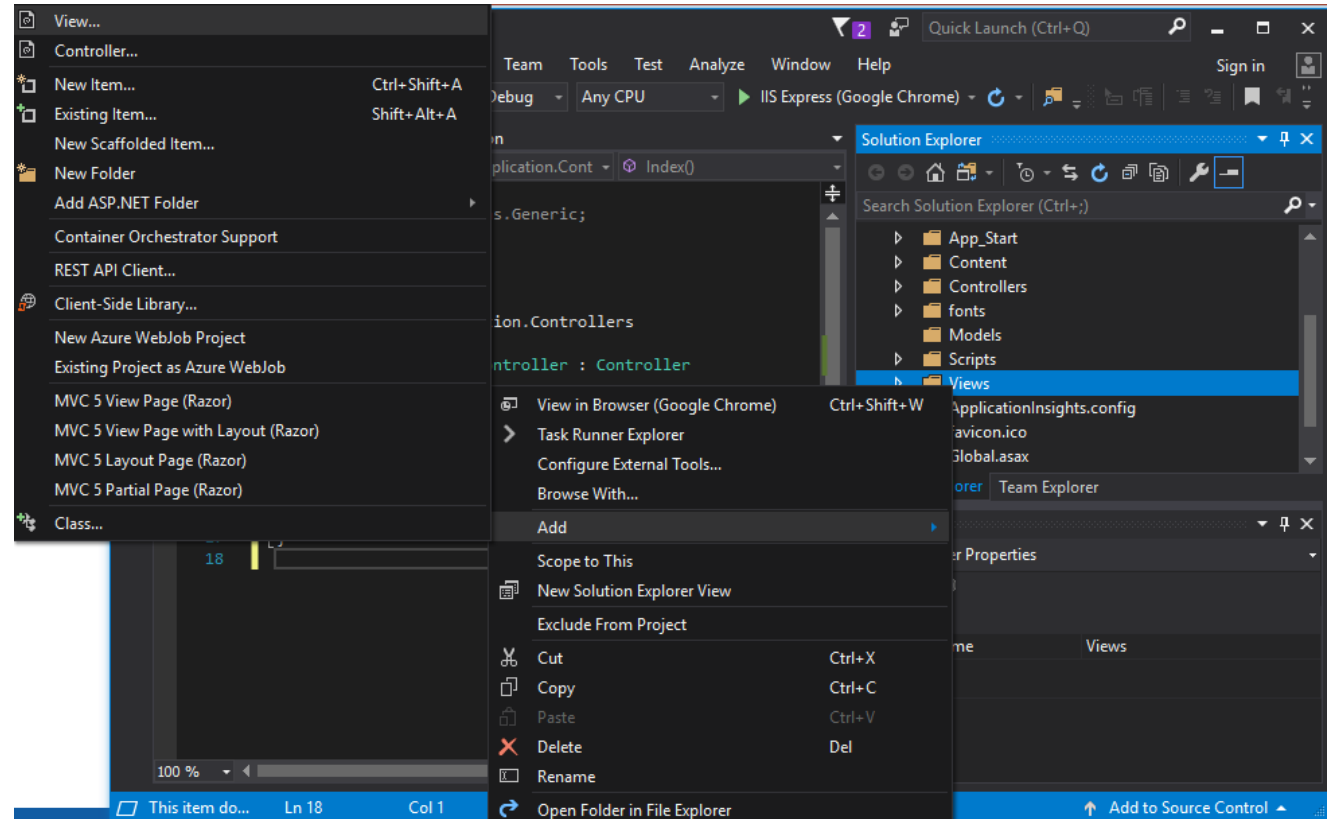
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace myMVCApplication.Controllers
{
    public class HomeController : Controller
    {
        // GET: Home
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

# Create a MVC Application

**Step 7:** Now we will add a new View to our Home Controller. To add a new View, right click view folder and click

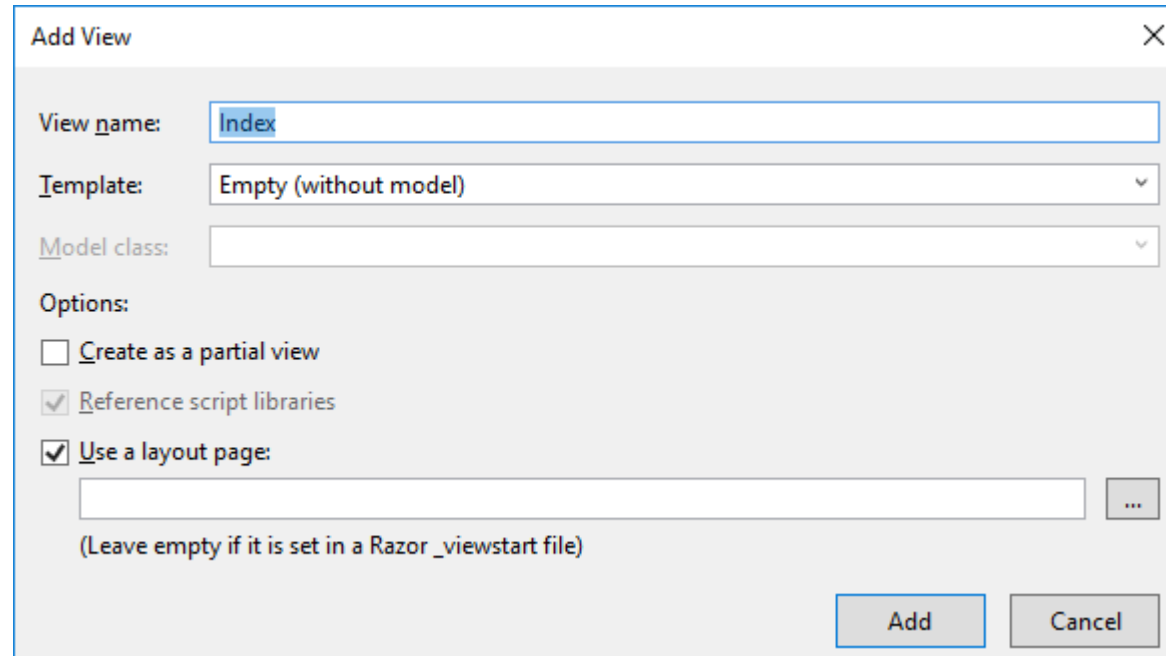
*Add → View.*





# Create a MVC Application

**Step 8:** Name the new View as Index and click Add.



The screenshot shows the 'Add View' dialog box with the following fields and options:

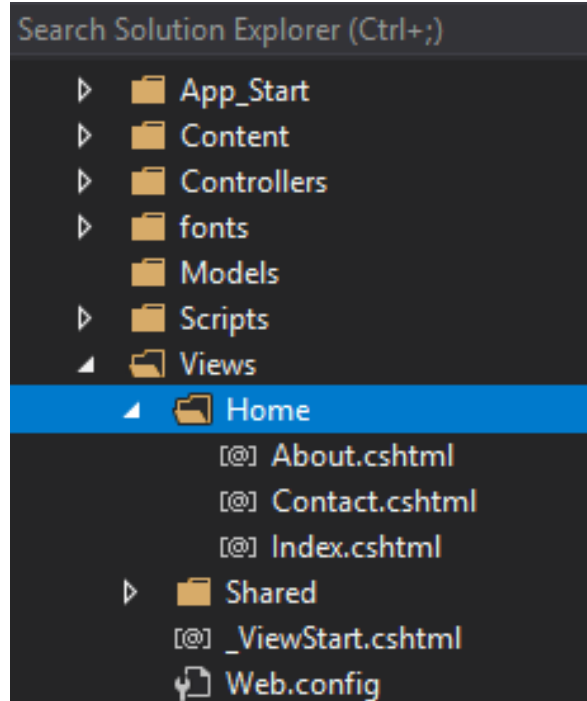
- View name:** A text box containing 'Index'.
- Template:** A dropdown menu showing 'Empty (without model)'.
- Model class:** An empty dropdown menu.
- Options:**
  - ☐ Create as a partial view
  - ☒ Reference script libraries
  - ☒ Use a layout page:
    - A text box for the layout page name, currently empty.
    - A button with three dots '...' to the right of the text box.

Below the options, there is a note: (Leave empty if it is set in a Razor \_viewstart file).

At the bottom right, there are two buttons: 'Add' and 'Cancel'.

# Create a MVC Application

This will add a new **cshtml** file inside Views/Home folder with the following code.



```
@{  
    ViewBag.Title = "Index";  
}  
  
<h2>Index</h2>
```

# Create a MVC Application

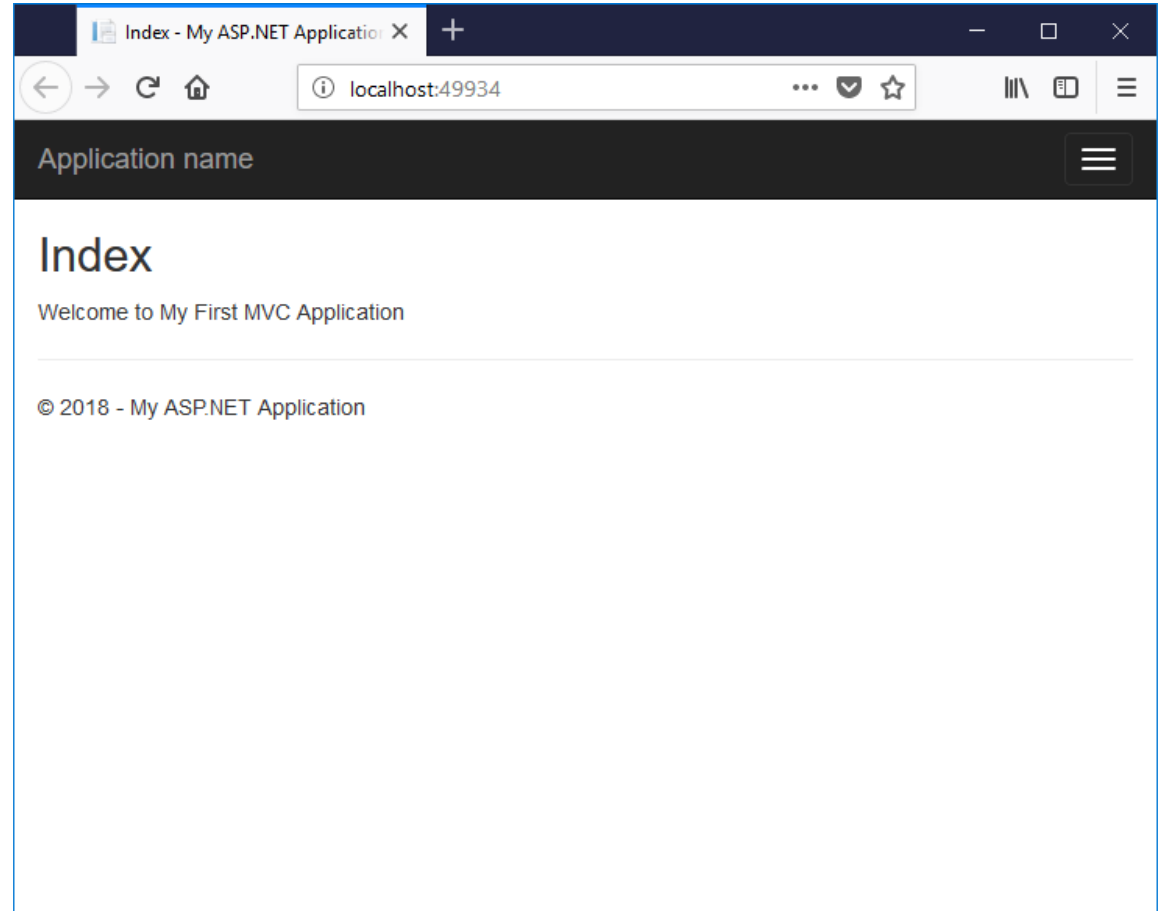
**Step 9:** Modify the above View's body content with the following code.

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<body>
    <div>
        Welcome to My First MVC Application
    </div>
</body>
```

**Step 10:** Modify the above View's body content with the following code.



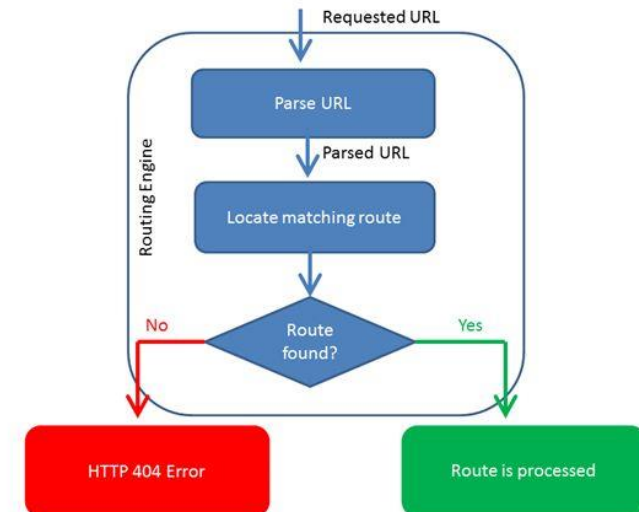
# Working with Url's and Routing

# About Routing

ASP.NET MVC Routing enables the use of URLs that are descriptive of the user actions and are more easily understood by the users. At the same time, Routing can be used to hide data which is not intended to be shown to the final user.

For example, in an application that does not use routing, the user would be shown the URL as *http://myapplication/Users.aspx?id=1* which would correspond to the file Users.aspx inside myapplication path and sending ID as 1, Generally, we would not like to show such file names to our final user.

To handle MVC URLs, ASP.NET platform uses the routing system, which lets you create any pattern of URLs you desire, and express them in a clear and concise manner. Each route in MVC contains a specific URL pattern. This URL pattern is compared to the incoming request URL and if the URL matches this pattern, it is used by the routing engine to further process the request.



# Routing System

To understand the MVC routing, consider the following URL:

*<http://servername/Products/Phones>*

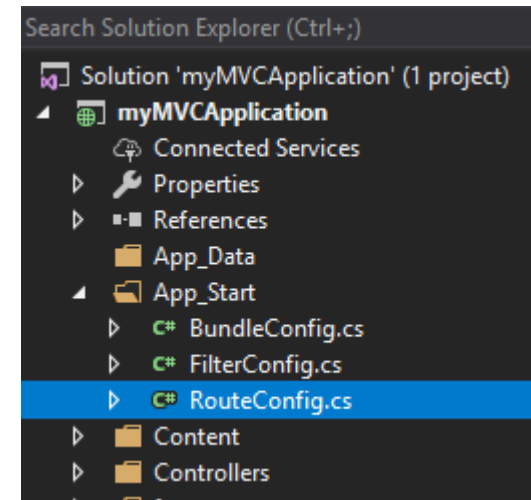
In the above URL, Products is the first segment and Phone is the second segment which can be expressed in the following format:

*[{controller}/{action}](#)*

The MVC framework automatically considers the first segment as the Controller name and the second segment as one of the actions inside that Controller.

**Note** – If the name of your Controller is ProductsController, you would only mention Products in the routing URL. The MVC framework automatically understands the Controller suffix.

Routes are defined in the RouteConfig.cs file which is present under the App\_Start project folder.



# Adding a Route Entry

In RouteConfig.cs file you will see the following code:

```
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 }
        );
    }
}
```

This RegisterRoutes method is called by the Global.asax when the application is started. The Application\_Start method under Global.ascx calls this MapRoute function which sets the default Controller and its action (method inside the Controller class).

# Adding a Route Entry

Modify the default mapping:

```
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 }
            defaults: new { controller = "Products", action = "Phones", id = UrlParameter.Optional }
        );
    }
}
```

This setting will pick the *ProductsController* and call the *Phone* method inside that. Similarly, if you have another method such as *Electronics* inside *ProductsController*, the URL for it would be:

***<http://servername/Products/Electronics>***



# Using Defaults

We call this style *conventional routing* because it establishes a *convention* for URL paths:

- The first path segment maps to the controller name
- The second maps to the action name.
- The third segment is used for an optional id used to map to a model entity

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

Using conventional routing with the default route allows you to build the application quickly without having to come up with a new URL pattern for each action you define. For an application with CRUD style actions, having consistency for the URLs across your controllers can help simplify your code and make your UI more predictable.

# Using Parameters

Create Route to accept parameters.

Following routes work for both with parameter and without parameter because parameter optional:

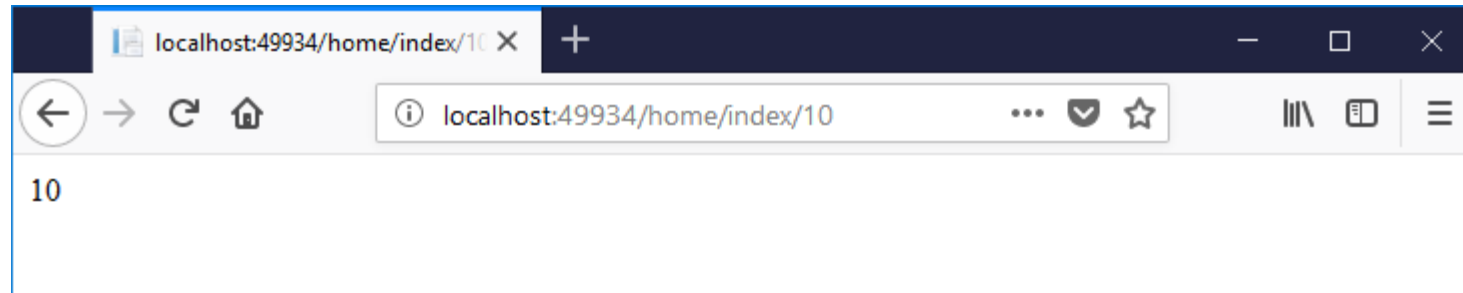
```
defaults: new { controller = "Products", action = "Phones", id = UrlParameter.Optional }
```

For getting that parameter value we modified an Index Action Method which now take ID as input parameter and Using Content to Display value on view.

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index(string id)
    {
        return Content(id);
    }
}
```

# Using Parameters

Now let's run application and check how it works.



# Using Parameters

## Create a New Route which takes Multiple Parameters

Here we are trying to send Country and State both as input parameters. For that in RouteConfig.cs we made modification by adding new routes with name **Citysearch**.

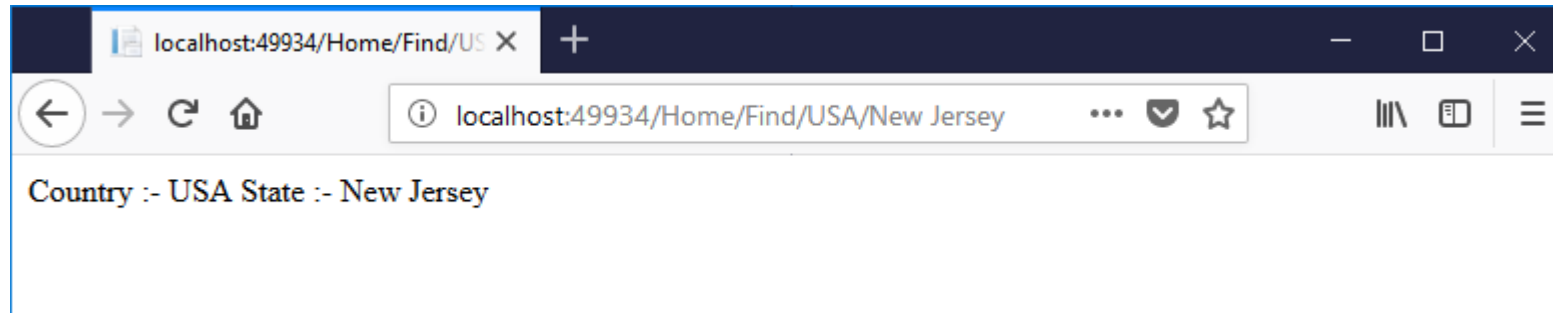
```
routes.MapRoute(  
    name: "Citysearch",  
    url: "Find/{Country}/{state}",  
    defaults: new { controller = "Home", action = "Find", Country = UrlParameter.Optional, state = UrlParameter.Optional }  
);
```

In Home Controller we need to add another action method with name **Find** that will be like as shown below

```
public ActionResult Find(string Country, string state)  
{  
    string result = "Country :- " + Country + " State :- " + state;  
    return Content(result);  
}
```

# Using Parameters

Now let's run application and check how it works.



# Using Constraints

We can restrict the type of value that we pass to actions using constraints. For example, if we expect an argument that is a number we have to restrict it to an integer type.

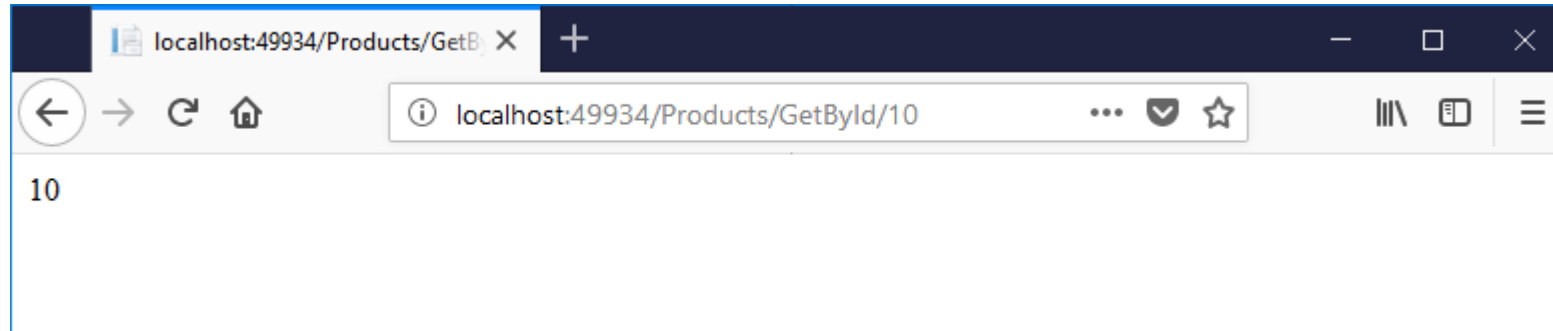
```
routes.MapRoute(  
    name: "getProductById",  
    url: "{controller}/Product/{id:int}",  
    defaults: new { controller = "Products", action = "GetById" }  
);
```

In Products Controller we need to add an action method with name **GetById** that will be like as shown below

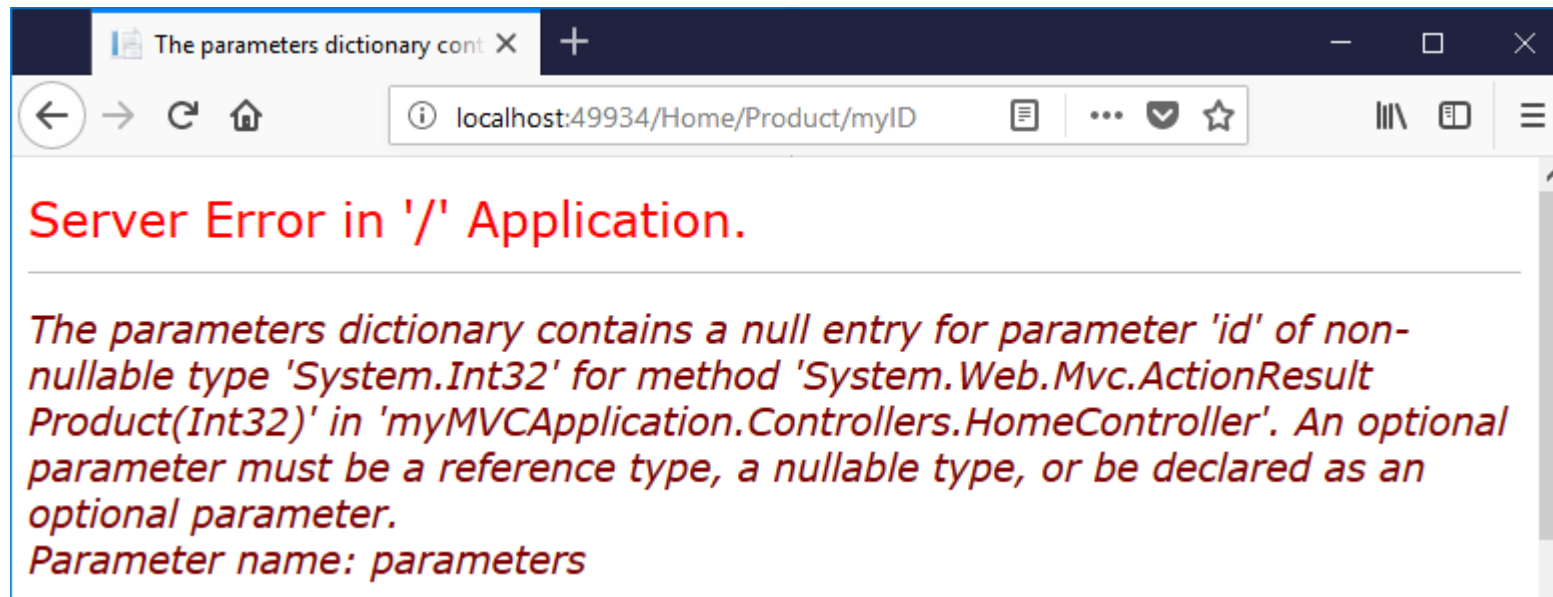
```
public class ProductsController : Controller  
{  
    // GET: Products  
    public ActionResult GetById(string id)  
    {  
        return Content(id);  
    }  
}
```

# Using Constraints

Now let's run application and check how it works.



If id is not an integer value:



# HTML Helpers



# HTML Helpers Methods

In ASP.Net web forms, developers are using the toolbox for adding controls on any particular page. However, in ASP.NET MVC application there is no toolbox available to drag and drop HTML controls on the view. In ASP.NET MVC application, if you want to create a view it should contain HTML code. So those developers who are new to MVC especially with web forms background finds this a little hard.

To overcome this problem, ASP.NET MVC provides HtmlHelper class which contains different methods that help you create HTML controls programmatically. All HtmlHelper methods generate HTML and return the result as a string. The final HTML is generated at runtime by these functions. The HtmlHelper class is designed to generate UI and it should not be used in controllers or models.

There are different types of helper methods.

- **Createinputs:** Creates inputs for text boxes and buttons.
- **Createlinks:** Creates links that are based on information from the routing tables.
- **Createforms:** Create form tags that can post back to our action, or to post back to an action on a different controller.

# Available HTML Helpers

The following list shows some of the currently available HTML helpers.

HTML Helpers	Action
ActionLink	Links to an action method
BeginForm	Marks the start of a form and links to the action method that renders the form.
CheckBox	Renders a check box.
DropDownList	Renders a drop-down list.
Hidden	Embeds information in the form that is not rendered for the user to see.
ListBox	Renders a list box.
Password	Renders a text box for entering a password.
RadioButton	Renders a radio button.
TextArea	Renders a text area (multi-line text box).
TextBox	Renders a text box.

# Render HTML Form

The BeginForm helper marks the start of an HTML form and renders as an HTML **form** element.

```
using (Html.BeginForm())  
  
@Html.Label("Enter your name") @Html.TextBox("name")  
<br /><br />  
  
@Html.Label("Select your favorite color:")  
<br />  
@Html.RadioButton("favColor", "Blue", true) @Html.Label("Blue") <br />  
@Html.RadioButton("favColor", "Blue", true) @Html.Label("Red") <br />  
@Html.RadioButton("favColor", "Blue", true) @Html.Label("Orange") <br />  
@Html.RadioButton("favColor", "Blue", true) @Html.Label("Yellow") <br />  
@Html.RadioButton("favColor", "Blue", true) @Html.Label("Brown") <br />  
@Html.RadioButton("favColor", "Blue", true) @Html.Label("Green") <br />  
  
<br />  
@Html.CheckBox("bookType")  
@Html.Label("I read more fiction than non-fiction") <br />  
<br /><br />  
@Html.Label("My favorite pet:")  
@Html.DropDownList("pets")  
<br /><br />  
<input type="submit" value="Submit" />
```

```
ViewData["Message"] = "Welcome to ASP.NET MVC!";
```

```
List<string> petList = new List<string>();  
petList.Add("Dog");  
petList.Add("Cat");  
petList.Add("Hamster");  
petList.Add("Parrot");  
petList.Add("Gold fish");  
petList.Add("Mountain lion");  
petList.Add("Elephant");
```

```
ViewData["Pets"] = new SelectList(petList);
```

```
return View();
```

Enter your name

Select your favorite color:

- ☐ Blue  
☐ Red  
☐ Orange  
☐ Yellow  
☐ Brown  
☒ Green

☐ I read more fiction than non-fiction

My favorite pet:

© 2018 - My ASP.NET Application

# Binding HTML helper to Model

```
@model myMVCApplication.Controllers.HomeController.EmployeeLoginViewModel

using (Html.BeginForm("EmployeeRegistration", "Employee", FormMethod.Post, new { @class = "form-horizontal", role = "form" }))
{
    <h4>Create a new account.</h4>
    <hr />
    <div class="form-group">
        @Html.LabelFor(m => m.Name, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Name, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

```
public class EmployeeLoginViewModel
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

Create a new account.

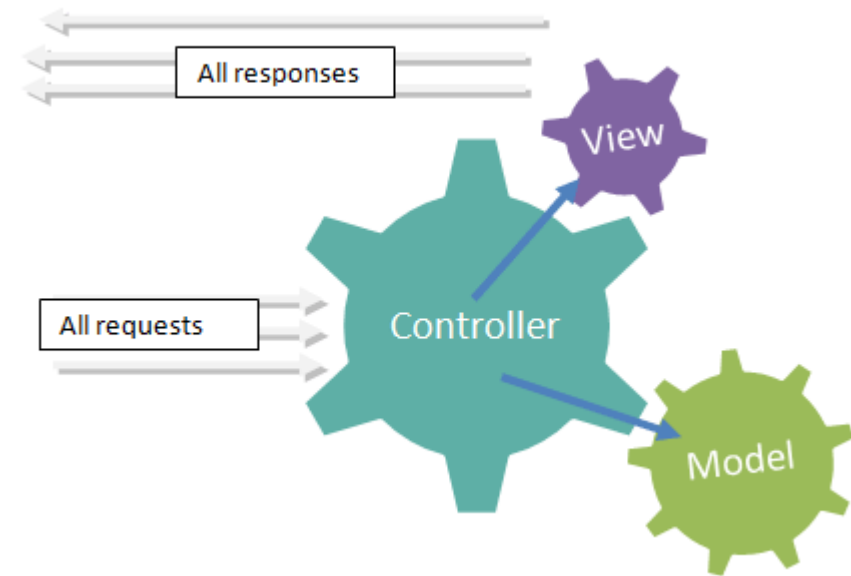
Name	<input type="text"/>
Email	<input type="text"/>
Password	<input type="password"/>
ConfirmPassword	<input type="password"/>
	<input type="button" value="Register"/>

# Working with Controllers

# Understanding Controllers

A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on to one or more of the views.

The brains of the application. The controller decides what the user's input was, how the model needs to change as a result of that input, and which resulting view should be used.



# Passing Data from Controller to View

There are various ways to pass data from a Controller to a View. Controllers can interact with Views, you can pass data from a Controller to a View to render a response back to a client.

## ViewBag

ViewBag is a very well known way to pass the data from Controller to View & even View to View. ViewBag uses the dynamic feature that was added in C# 4.0.



You can see how data flows from the "Controller" to the "View", and how it looks in the browser.

# Passing Data from Controller to View

## ViewData

ViewBag and ViewData serves the same purpose in allowing developers to pass data from controllers to views. When you put objects in either one, those objects become accessible in the view.

ViewData is typed as a dictionary containing "objects", we need to cast ViewData["Students"] to a List<string> or an IEnumerable<string> in order to use the foreach statement on it

```
public ActionResult Index()
{
    var students = new List<string>
    {
        "Andrew",
        "Thomas",
        "Susan"
    };

    ViewData["students"] = students;

    return View();
}
```



```
<h2>Friend List</h2>
<body>
    <ul>
        @foreach (var std in (List<string>)ViewData["students"])
        {
            <li>
                @std
            </li>
        }
    </ul>
</body>
```



## Friend List

- Andrew
- Thomas
- Susan

© 2018 - My ASP.NET Application



# Passing Data from Controller to View

## TempData

TempData is meant to be a very short-lived instance, and you should only use it during the current and the subsequent requests only. Since TempData works this way, you need to know for sure what the next request will be, and redirecting to another view is the only time you can guarantee this. You can use TempData to pass error messages or something similar.

```
public ActionResult Index()
{
    ViewData["VDFriend"] = "Andrew";
    ViewBag.VBFriend = "Andrew";
    TempData["TDFriend"] = "Andrew";
    return View();
}
```



```
<h2>Friend</h2>

<body>

    <p>Using ViewData: @ViewData["VDFriend"]</p>
    <p>Using ViewBag: @ViewBag.VBFriend</p>
    <p>Using TempData: @TempData["TDFriend"] </p>

</body>
```



## Friend

Using ViewData: Andrew

Using ViewBag: Andrew

Using TempData: Andrew

---

© 2018 - My ASP.NET Application

# View Bag vs View Data vs Temp Data

ViewData	ViewBag	TempData
It is Key-Value Dictionary collection	It is a type object	It is Key-Value Dictionary collection
ViewData is a dictionary object and it is property of ControllerBase class	ViewBag is Dynamic property of ControllerBase class.	TempData is a dictionary object and it is property of ControllerBase class.
ViewData is Faster than ViewBag	ViewBag is slower than ViewData	NA
ViewData is introduced in MVC 1.0 and available in MVC 1.0 and above	ViewBag is introduced in MVC 3.0 and available in MVC 3.0 and above	TempData is also introduced in MVC1.0 and available in MVC 1.0 and above.
ViewData also works with .net framework 3.5 and above	ViewBag only works with .net framework 4.0 and above	TempData also works with .net framework 3.5 and above
Type Conversion code is required while enumerating	In depth, ViewBag is used dynamic, so there is no need to type conversion while enumerating.	Type Conversion code is required while enumerating
Its value becomes null if redirection has occurred.	Same as ViewData	TempData is used to pass data between two consecutive requests.
It lies only during the current request.	Same as ViewData	TempData only works during the current and subsequent request

# View Bag vs View Data vs Temp Data

## Conclusion

We have three options:

- ***ViewData***
- ***ViewBag***
- ***TempData***

for passing data from controller to view and in next request. ViewData and ViewBag are almost similar and it helps us to transfer the data from controller to view whereas TempData also works during the current and subsequent requests.

# Action Method Parameters

Every action methods can have input parameters as normal methods. It can be primitive data type or complex type parameters as shown in the below example.

```
[HttpPost]
public ActionResult Edit(Student std)
{
    // update student to the database
    return RedirectToAction("Index");
}

[HttpDelete]
public ActionResult Delete(int id)
{
    // delete student from the database whose id matches with specified id
    return RedirectToAction("Index");
}
```

# Types of Action Results

Name	Framework Behavior	Producing Method
ContentResult	Writes a string value directly into the response.	Content
EmptyResult	Blank HTTP response.	
FileContentResult	Takes the contents of a file (represented as an array of bytes) and writes the contents into the HTTP response.	File
FilePathResult	Takes the contents of a file at the given location and writes the contents into the HTTP response.	File
FileStreamResult	Takes a file stream produced by the controller and writes the stream into the HTTP response.	File
HttpUnauthorizedResult	A special result used by authorization filters when authorization checks fail.	
JavaScriptResult	Responds to the client with a script for the client to execute.	JavaScript
JsonResult	Responds to the client with data in JavaScript Object Notation (JSON).	JSON
RedirectResult	Redirects the client to a new URL.	Redirect
RedirectToRouteResult	Renders the specified view to respond with an HTML fragment (typically used in AJAX scenarios).	RedirectToRoute / RedirectToAction
PartialViewResult	Renders the specified view to respond with an HTML fragment (typically used in AJAX scenarios).	PartialView
ViewResult	Renders the specified view and responds to the client with HTML.	View

# Types of Action Result

## View Result

View result is a basic view result. It returns basic results to view page. View result can return data to view page through which class is defined in the model. View page is a simple HTML page. Here view page has “.cshtml” extension.

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

# Types of Action Result

## Partial View Result

Partial View Result is returning the result to Partial view page. Partial view is one of the views that we can call inside Normal view page.

```
public PartialViewResult Index()  
{  
    return PartialView("_PartialView");  
}
```

We should create a Partial view inside shared folder, otherwise we cannot access the Partial view. The diagram is shown above the Partial view page and Layout page because layout page is a Partial view. Partial View Result class is also derived from Action Result.

# Types of Action Result

## Redirect Result

Redirect result is returning the result to specific URL. It is rendered to the page by URL. If it gives wrong URL, it will show 404 page errors.

```
public ActionResult Index()  
{  
    return Redirect("Home/Contact");  
}
```



# Types of Action Result

## Redirect to Action Result

Redirect to Action result is returning the result to a specified controller and action method. Controller name is optional in Redirect to Action method. If not mentioned, Controller name redirects to a mentioned action method in current Controller. Suppose action name is not available but mentioned in the current controller, then it will show 404 page error.

```
public ActionResult Index()
{
    return RedirectToAction("Login", "Account");
}
```

# Types of Action Result

## Json Result

Json result is a significant Action Result in MVC. It will return simple text file format and key value pairs. If we call action method, using Ajax, it should return Json result.

```
public ActionResult Index()
{
    var persons = new List<Person1>
    {
        new Person1{Id=1, FirstName="Harry", LastName="Potter"},
        new Person1{Id=2, FirstName="James", LastName="Smith"}
    };

    return Json(persons, JsonRequestBehavior.AllowGet);
}
```

# Types of Action Result

## File Result

File Result returns different file format view page when we implement file download concept in MVC using file result. Simple examples for file result are shown below:

```
public ActionResult Index()  
{  
    return File("Web.Config", "text");  
}
```

# Types of Action Result

## Redirect to Route Result

To redirect user to another route url from action method of the controller, we can use RedirectToRoute method by passing route name defined in the App\_Start/RouteConfig.cs file.

```
routes.MapRoute(  
    name: "MyCustomRoute",  
    url: "MyEmployees/{action}/{id}", \  
    defaults: new { controller = "PersonalDetail",  
                    action = "Create", id = UrlParameter.Optional }  
);  
  
public ActionResult OutputToAction()  
{  
    return RedirectToRoute("MyCustomRoute", new { Id = 5 }); \  
}
```