Model trong ASP.Net MVC

Nội dung

- ♦ Define and describe models
- ♦ Explain how to create a model
- ♦ Describe how to pass model data from controllers to views
- ♦ Explain how to create strongly typed models
- ♦ Explain the role of the model binder
- ♦ Explain how to use scaffolding in Visual Studio.NET

Giới thiệu về Model

- In an ASP.NET MVC application, a model:
  - Is a class containing properties that represents data of an application.
  - Represents data associated with the application.
- ASP.NET MVC Framework is based on the MVC pattern.
- The MVC pattern defines the following three types of models, where each model has specific purpose:
  - **Data model**: Represent classes that interact with a database. Data models are set of classes that can either follow the database-first approach or code-first approach.
  - **Business model**: Represent classes that implement a functionality that represents business logic of an application.
  - **View model**: Represent classes that provide information passed between controllers and views.

Tạo Model

- Tạo Model trong dự án ASP.Net MVC:
  - Tạo một public class với tên tùy chọn.
  - Định nghĩa các thuộc tính public cho lass đó.
- Sau đây là đoạn mã cho Model tên là User:

```
public class User
{
        public long Id { get; set; }
        public string name { get; set; }
        public string address { get; set; }
        public string email { get; set; }
}
```

Thao tác với Model trong Controller

- In an ASP.NET MVC application when a user request for some information, the request is received by an action method.
- The action method is used to access the model storing the data.
- To access the model, you need to create an object of the model class and either retrieve or set the property values of the object.
- Following code shows the creating an object of the model class in the Index()action method:

```
public ActionResult Index()
{
      var user = new MVCModelDemo.Models.User();
      user.name = "John Smith";
      user.address = "Park Street";
      user.email = "john@mvcexample.com";
      return View();
}
```

Truyền Model cho View

- Once you have accessed the model within a controller, you need to make the model data accessible to a view so that the view can display the data to the user.
- To do this, you need to pass the model object to the view while invoking the view.
- You can model the object as follows:
- A single object
- A collection of model objects

Truyền Model cho View

- In an action method, you can create a model object and then pass the object to a view by using the ViewBag object.
- Following code shows passing the User model data from an action method to a view by using a ViewBag object:

```
public ActionResult Index()
{
var user = new MVCModelDemo.Models.User();
user.name = "John Smith";
user.address = "Park Street";
user.email = "john@mvcexample.com";
ViewBag.user = user;
return View();
}
```

- In this code, an object of the User model class is created and initialized with values. The object is then, passed to the view by using a ViewBag object.

Truyền Model cho View

- ♦ You can access the data of the model object stored in the ViewBagobject from within the view.
- ♦ Following code snippet shows accessing the data of the model object stored in the ViewBagobject:
- ♦

```
<!DOCTYPE html>
<html>
<body>
        <p> User Name: @ViewBag.user.name</p>
        <p> Address: @ViewBag.user.address</p>
        <p> Email: @ViewBag.user.email</p>
</body>
</html>
```

- ♦ In this code, the view accesses and displays the name, address, and emailproperties of the Usermodel object stored in the ViewBagobject.

Truyền Model cho View

- Following code shows passing a collection of model objects to a view:

```
public ActionResult Index(){
        var user = new List<User>();
        var user1 = new User();
        user1.name = "Mark Smith";
        user1.address = "Park Street";
        user1.email = "Mark@mvcexample.com";
        var user2 = new User();
        user2.name = "John Parker";
        user2.address = "New Park";
        user2.email = "John@mvcexample.com";
        var user3 = new User();
        user3.name = "Steave Edward ";
        user3.address = "Melbourne Street";
        user3.email = "steave@mvcexample.com";
        user.Add(user1); user.Add(user2);
        user.Add(user3);
        ViewBag.user = user; return View(); }
```

Truyền Model cho View

- The preceding code:
- Creates and initializes three objects of the model class, named User.
- Then, a List<User>collection is created and the model objects are added to it.
- Finally, the collection is passed to the view by using a ViewBag object.
- Once you pass a collection of model objects to a view using a ViewBag object:
- You can retrieve the collection stored in the ViewBag object from within the view.
- You can iterate through the collection to retrieve each model object.
- You can access their properties.

Truyền Model cho View

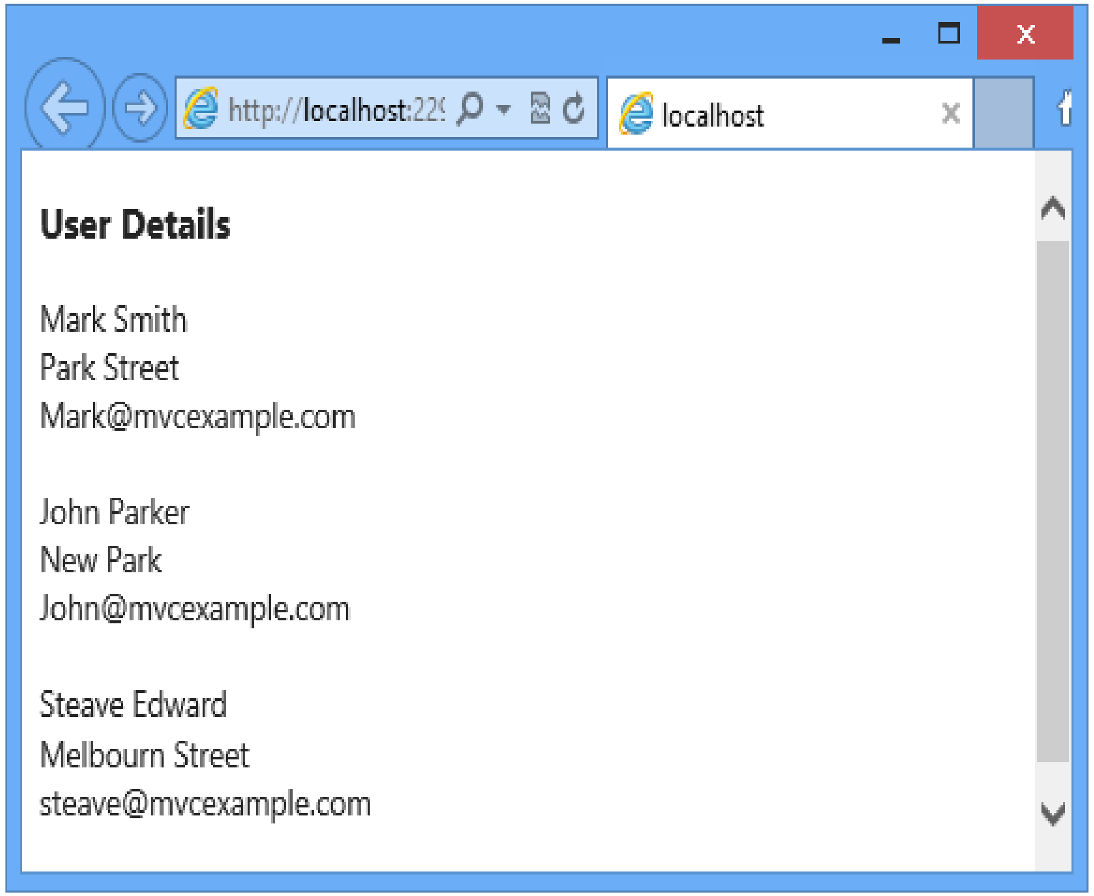
♦ Following code snippet shows retrieving model objects from a collection and displaying their properties:

```
<!DOCTYPE html>
<html> <body>
<h3>User Details</h3>
@{ var user = ViewBag.user;}
@foreach (var p in user)
       {
       @p.name<br />
       @p.address<br />
       @p.email<br />
       <br /> }
</body>
 </html>
```

♦ This code uses a foreach loop to iterate through the collection of model object stored in the ViewBag object and the name, address, and email properties are rendered as response

Tạo View cho một Action

- Following figure shows the output of retrieving model objects:
- You can use another approach to pass a collection of model objects from an action method to a view is to pass the collection directly as a parameter to the View()method.
- This code uses a foreach loop to iterate through the collection of model object stored in the ViewBag object and the name, address, and email properties are rendered as response.



User Details

Mark Smith
Park Street
Mark@mvcexample.com

John Parker
New Park
John@mvcexample.com

Steave Edward
Melbourn Street
steave@mvcexample.com

Truyền Model cho View

- ♦ Following code snippet shows passing a collection of model objects to a view as a parameter to the View() method:
- ♦ The preceding code will create and initializes three objects of the model class, named User.
    - ♦ Then, a List<User>collection is created and the model objects are added to it.
    - ♦ Finally, the collection is passed to the view as a parameter to the View() method.

```
ActionResult Index() {
        var user = new List<User>();
        var user1 = new User();
        user1.name = "Mark Smith";
        user1.address = "Park Street";
        user1.email = "Mark@mvcexample.com";
        var user2 = new User();
        user2.name = "John Parker";
        user2.address = "New Park";
        user2.email = "John@mvcexample.com";
        var user3 = new User();
        user3.name = "Steave Edward ";
        user3.address = "Melbourn Street";
        user3.email = "steave@mvcexample.com";
        user.Add(user1);
        user.Add(user2);
        user.Add(user3);
        return View(user);
}
```

Truyền Model cho View

- Following code snippet shows retrieving the user information in the view:

```
<!DOCTYPE html>
<html> <body>
<h3>User Details</h3>
@{ var user = Model }
@foreach(var p in user)
{
        @p.name <br />
        @p.address<br />
        @p.email<br />
        <br/>
}
</body>
</html>
```

- This code shows how to retrieve the user information that has been passed to a view by passing a collection of objects as a parameter.

Định kiểu cho Model trên View

- Khi truyền model từ Controller sang View, có thể View không biết chính xác kiểu dữ liệu trong Model.
- Giải pháp đặt ra là chuyển kiểu

```
<html> <body>
<h3>User Details</h3>
@{
var user = Model as MVCModelDemo.Models.User;
}
@user.name <br/>
@user.address<br/>
@user.email<br/>
</body> </html>
```

- Trong ví dụ trên, Model được chuyển sang kiểu MVCModelDemo.Models.User.

Định kiểu cho Model trên View

- Ta có thể không cần chuyển kiểu cho Model một cách tường minh như ví dụ trước mà khai báo "Định kiểu cho Model trên View" bằng việc sử dụng từ khóa @model theo cú pháp sau:

```
@model <model_name>
```

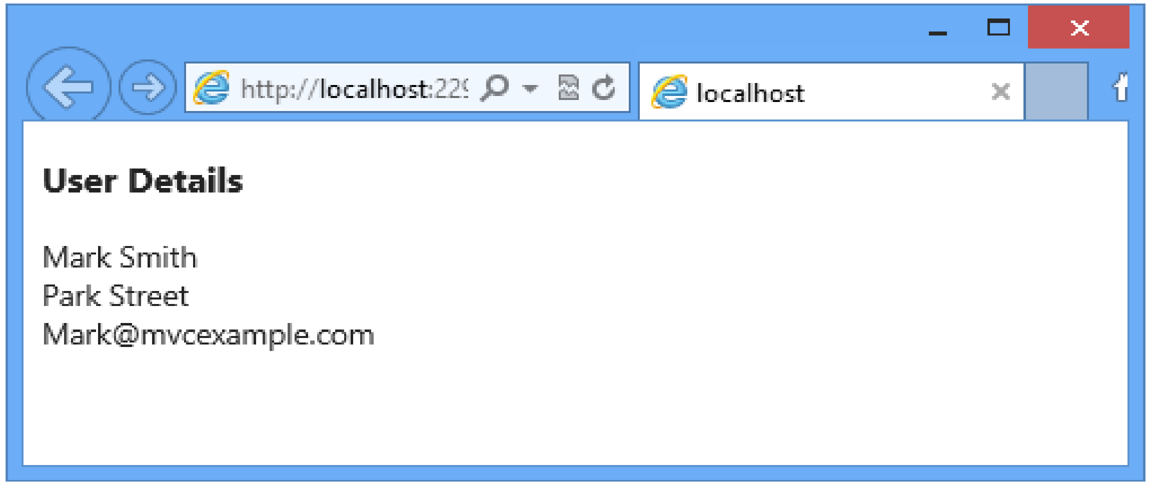
- Trong đó
  - model_name: là tên đầy đủ của Model đã được định nghĩa
  - Khi sử dụng từ khóa @model keyword, ta có thể truy nhập các thuộc tính của đối tượng Model trên View.

Định kiểu cho Model trên View

- Đoạn mã sau mô tả truy nhập các thuộc tính của Model bằng từ khóa @model

```
@model MVCModelDemo.Models.User
<html><body>
<h3>User Details</h3>
      @Model.name <br/>
      @Model.address<br/>
      @Model.email<br/>
</body> </html>
```

- Kết quả các thuộc tính của Model được hiển thị như sau:



User Details

Mark Smith
Park Street
Mark@mvcexample.com

Định kiểu cho Model trên View

- Sometime, you may need to pass a collection of objects to a view.
- In such situation, you can use the **@model**keyword.
- Following code snippet shows using the **@model** keyword to pass a collection of Model object:

```
@model IEnumerable<MVCModelDemo.Models.User>
```

- This code uses the **@model** keyword to indicate that it expects a collection of the Usermodel objects.
- Once you pass a collection of the model objects, you can access it in a view.

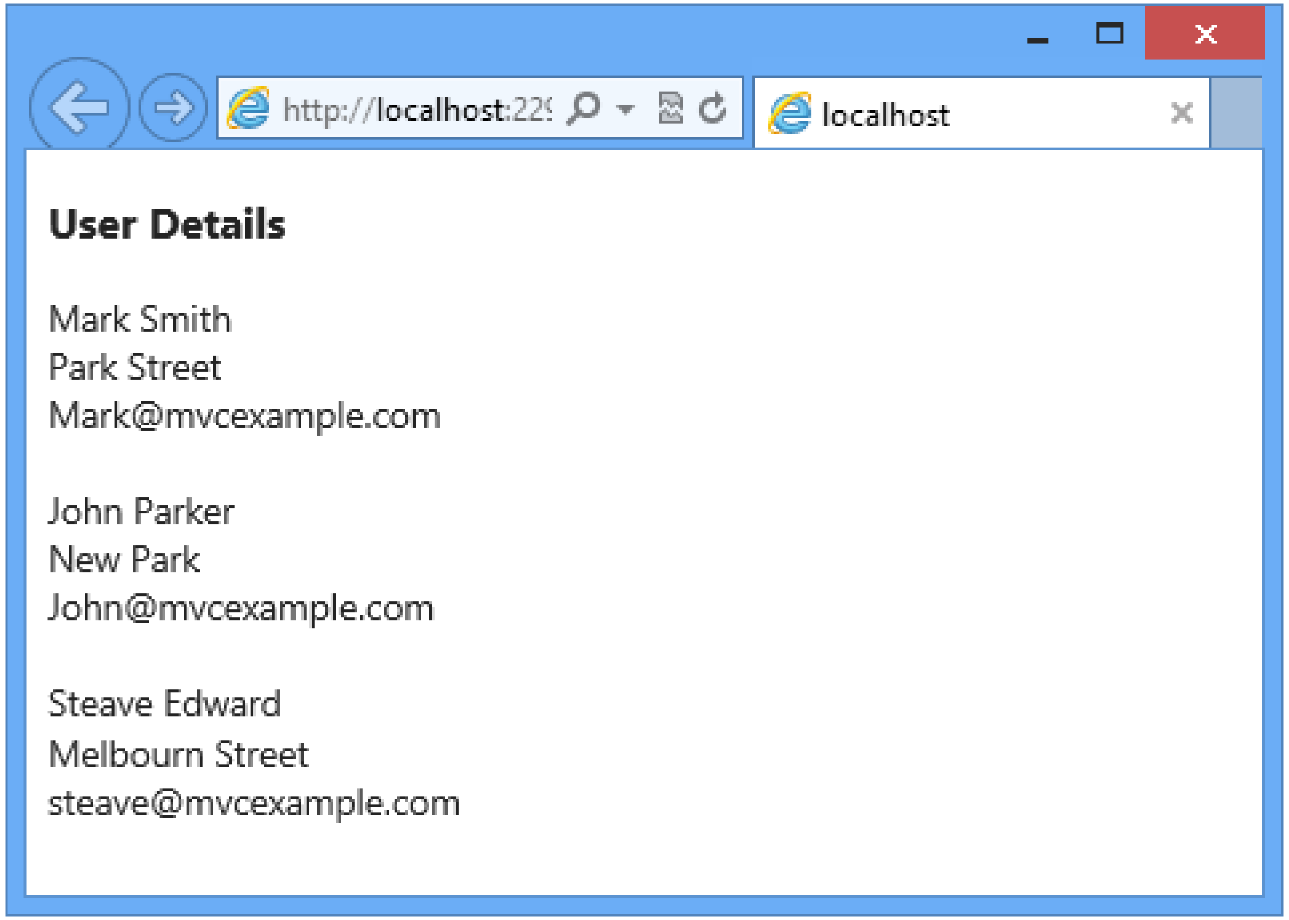
Định kiểu cho Model trên View

- Following code snippet shows accessing the collection of the Usermodel in a view:

```
@model IEnumerable<MVCModelDemo.Models.User>
<html>
<body>
<h3>User Details</h3>
@{
        var user = Model;
}
@foreach(var u in user)
{
        @u.name <br/>
        @u.address<br/>
        @u.email<br/>
        <br/>
}
</body>
</html>
```

Định kiểu cho Model trên View

- Following figure shows the output of accessing collection of the Usermodel:



```
User Details

Mark Smith
Park Street
Mark@mvcexample.com

John Parker
New Park
John@mvcexample.com

Steave Edward
Melbourn Street
steave@mvcexample.com
```

Sử dụng HTML Helper

- The MVC Framework:
  - Enables these helper methods to directly associate with model properties in a strongly types views.
  - Provides helper methods that you can use only in strongly typed views.
- Following table lists the helper methods that you can use only in strongly typed views:

Helper Method	Description
Html.LabelFor()	Is the strongly typed version of the Html.Label()helper method that uses a lambda expression as its parameter, which provides compile time checking.
Html.DisplayNameFor()	Is used to display the names of model properties.
Html.DisplayFor()	Is used to display the values of the model properties.

Sử dụng HTML Helper

Helper Method	Description
Html.TextBoxFor()	Is the strongly typed version of the Html.TextBox()helper method.
Html.TextAreaFor()	IIs the strongly typed version of the Html.TextArea()helper method that generates the same markup as that of the Html.TextArea()helper method.
Html.EditorFor()	Is used to display an editor for the specified model property.
Html.PasswordFor()	Is the strongly typed version of the Html.Password()helper method.
Html.DropDownListFor()	Is the strongly typed version of the Html.DropDownList()helper method that allows selection of a single item.

Sử dụng HTML Helper

♦ Following code snippet shows the HTML helper methods in a strongly typed view:

```
@model MVCModelDemo.Models.User
@{
ViewBag.Title = "User Form";
}
<h2>User Form</h2>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <div>@Html.LabelFor(model => model.name)</div>
    <div>@Html.EditorFor(model => model.name)</div>
    <div>@Html.LabelFor(model => model.address)</div>
    <div>@Html.EditorFor(model => model.address)</div>
    <div>@Html.LabelFor(model => model.email)</div>
    <div>@Html.EditorFor(model => model.email)</div>
}
```

Sử dụng HTML Helper

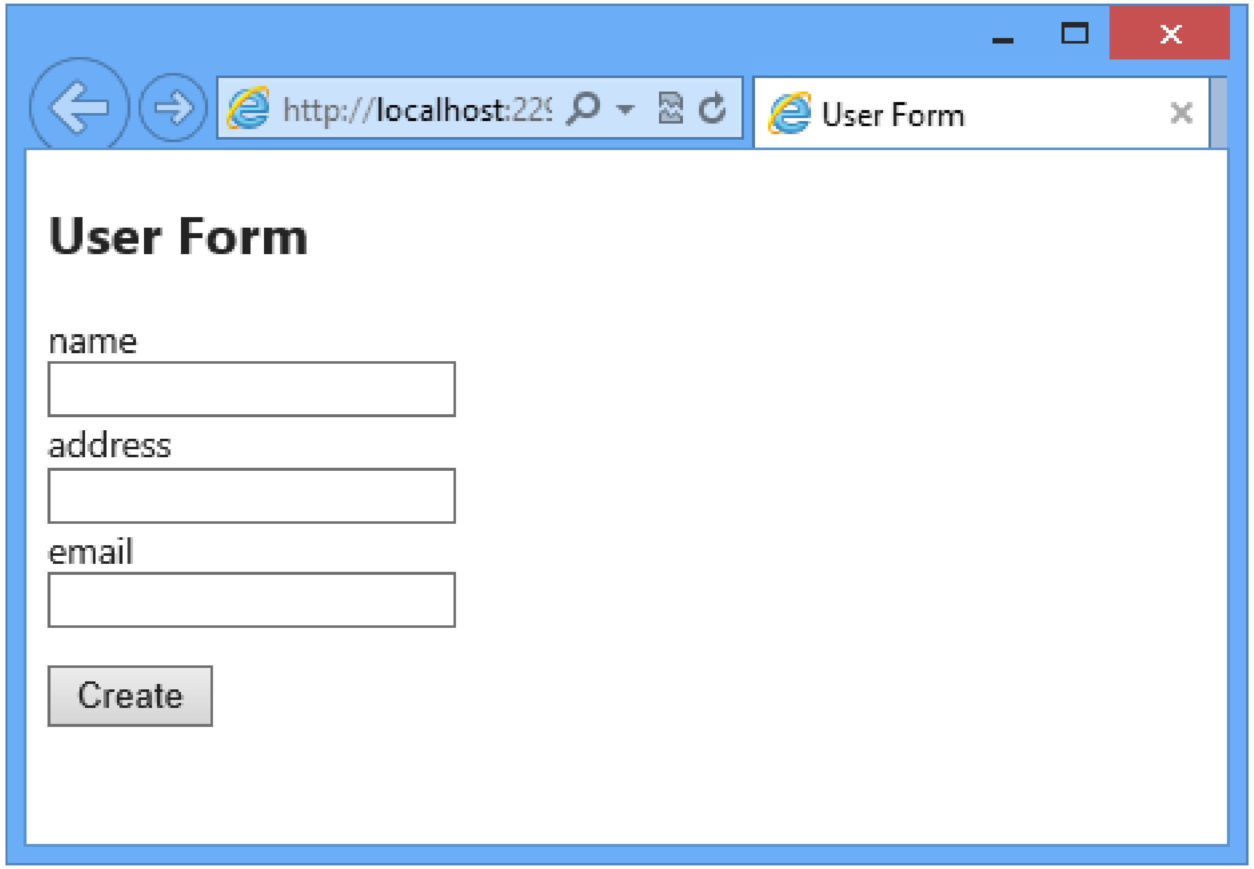
- Following code snippet shows the HTML helper methods in a strongly typed view:

```
@model MVCModelDemo.Models.User
@{ViewBag.Title = "User Form";}
<h2>User Form</h2>
@using (Html.BeginForm()) {
      @Html.ValidationSummary(true)
      <div>@Html.LabelFor(model => model.name)</div>
      <div>@Html.EditorFor(model => model.name)</div>
      <div>@Html.LabelFor(model =>model.address)</div>
      <div>@Html.EditorFor(model =>model.address)</div>
      <div>@Html.LabelFor(model =>model.email) </div>
      <div>@Html.EditorFor(model =>model.email)</div>
      <p><input type="submit" value="Create" /></p>
}
```

- In this code:
  - The Html.LabelFor() method is used to display labels based on the property names of the model.
  - The Html.EditorFor() method is used to display editable fields for the properties of the model.

Sử dụng HTML Helper

- Following figure shows the output of using HTML helper methods:



User Form

name

address

email

Binding Model

- When a user submits information in a form within a strongly typed view, ASP.NET MVC automatically examines the HttpRequest object and maps the information sent to fields in the model object.
- The process of mapping the information in the HttpRequest object to the model object is known as model binding.
- Some of the advantages of model binding are as follows:
- Automatically extract the data from the HttpRequest object.
- Automatically converts data type.
- Makes data validation easy.

Binding Model

- The MVC Framework provides a model binder that performs model binding in application.
- The Default Model Binder class implements the model binder on the MVC Framework.
- The two most important roles of the model binder are as follows:
  - Bind request to primitive values
  - Bind request to objects

Binding Primitive values

- To understand how the model binder binds request to primitive values, consider a scenario where you are creating a log in form that accepts log in details from user.
- For this, first you need to create a Login model in your application.
- Following code snippet shows the Login model class:

```
public class Login
{
        public string userName { get; set; }
        [DataType(DataType.Password)]
        public string password { get; set; }
}
```

- This code creates two properties named userName and password in the Login model.
- After creating the model class, you need to create an Index.cshtml view to display the login form.

Binding Primitive values

- Following code shows the content of the Index.cshtml file:

```
@model ModelDemo.Models.Login
@{ ViewBag.Title = "Index";}
<h2>User Details</h2>
@using (Html.BeginForm()) {
      @Html.ValidationSummary(true)
      <div>@Html.LabelFor(model =>model.userName)</div>
      <div>@Html.EditorFor(model =>model.userName)</div>
      <div>@Html.LabelFor(model =>model.password)</div>
      <div>@Html.EditorFor(model =>model.password) </div>
      <div><input type="submit" value="Submit" /></div>
}
```

Binding Primitive values

- Once, you have created the view, you need to create a controller class that contains the Index() action method to display the view.
- Following code shows the HomeController

```
public class HomeController : Controller {
        public ActionResult Index() {
                return View();
        }
        [HttpPost]
        public ActionResult Index(string userName, string password) {
                if (userName == "Peter" && password == "pass@123") {
                                string msg = "Welcome " + userName;
                                return Content(msg);
                }
                else {
                                return View();
                }
        }
}
```

Binding Primitive values

- In the preceding code:
  - The first Index() method returns the Index.cshtml view that displays a login form.
  - The second Index() method is marked with the HttpPost attribute. This method accepts two primitive as parameters. The Index()method compares the parameters with predefined values and returns a message if the comparison returns true. Else, the Index()method returns back the Index.cshtml view.
- When a user submits the login data, the default model binder maps the values of the username and password fields to the primitive type parameters of the Index()action method.
- In the Index()action method, you can perform the required authentication and return a result.

Binding object

- To understand how the model binder binds requests to objects, consider the same scenario where you are creating a login form.
- For this, you have already created the Login model and the Index.cshtml view.
- To bind request to object, you need to update the controller class so that it accepts a Login object as a parameter, instead of an HttpRequest object.

Binding object

- Following code shows the updated controller class:

```
public class HomeController : Controller {
public ActionResult Index() {
        return View();
}
[HttpPost]
public ActionResult Index(Login login) {
        if (login.userName == "Peter" &&login.password == "pass@123") {
                string msg = "Welcome " + login.userName;
                return Content(msg); }
        else {
                return View(); } }
}
```

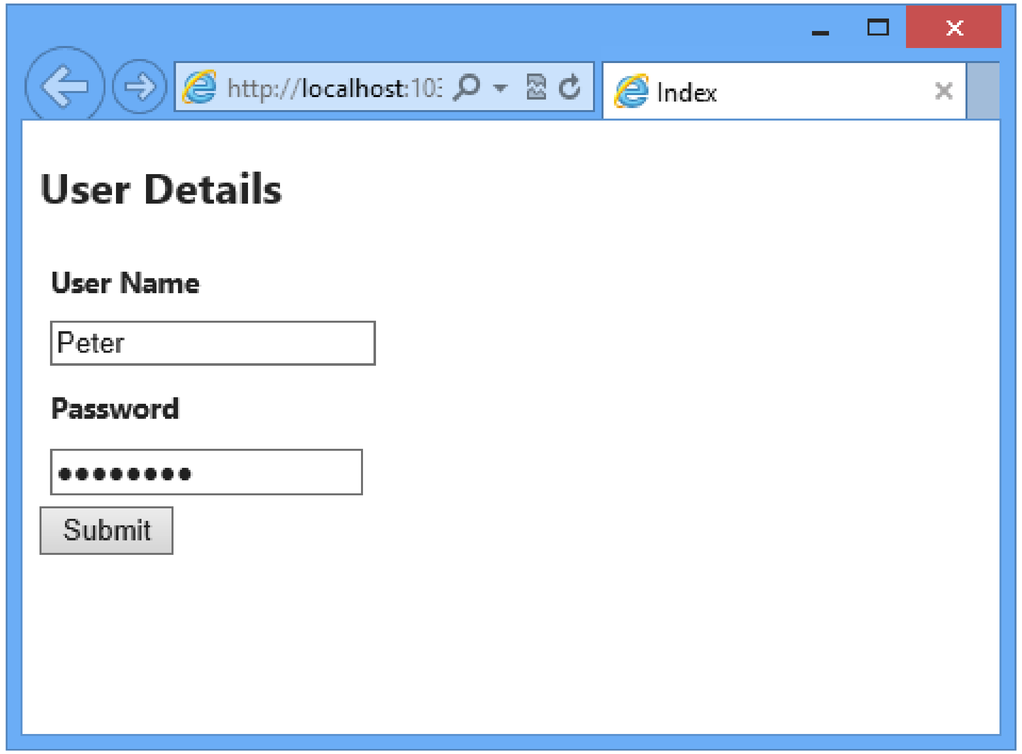
- In this code:
  - The first Index()method returns the Index.cshtml view that displays a login form.
  - The second Index()method automatically removes the data from the HttpRequest object and put into the Login object.

Binding object

- When a user submits the login data, the Index()method validates the username and password passed in the Login object.
- When the validation is successful, the view displays a welcome message.
- When you access the application from the browser the Index.cshtml view displays the login form.
- 1. Type Peter in the **UserName** text field and pass@123 in the **Password** field of the login form.

Binding object

- Following figure shows the login form with data specified in the **User Name** and **Password** fields:



http://localhost:10

Index

User Details

User Name

Peter

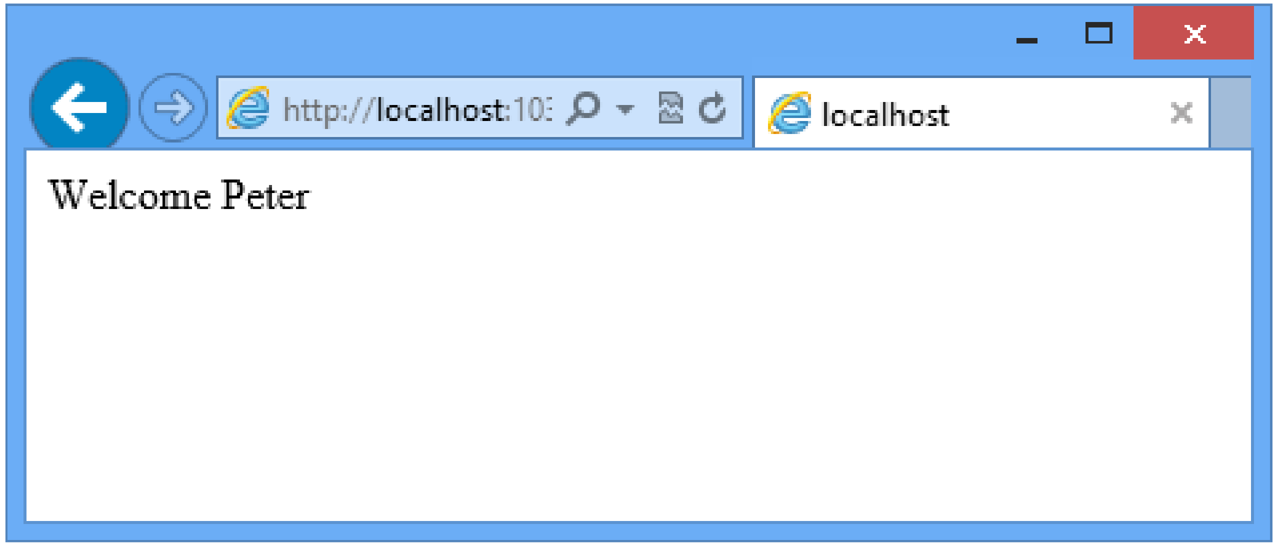
Password

●●●●●●●●●

- 2. Click **Submit**. The login form displays a ‘WelcomePeter’ message.

Binding object

- Following figure shows the output of the Index.cshtml view after submitting the user name and password:



Welcome Peter

Visual Studio.NET Scaffolding

- The ASP.NET MVC Framework provides a feature called scaffolding that allows you to generate views automatically.
- By convention, scaffolding uses specific name for views. After creating a view it stores the auto-generated code in respective places for the application to work.
- Following are the five types of template that provides to create views:
  - **List**: Generates markup to display the list of model objects.
  - **Create**: Generates markup to add a new object to the list.
  - **Edit**: Generates markup to edit an existing model object.
  - **Details**: Generates markup to show the information of an existing model object.
  - **Delete**: Generates markup to delete an existing model object.
- In this code, ViewData is used to display the values of the Message and CurrentTime keys.

List template

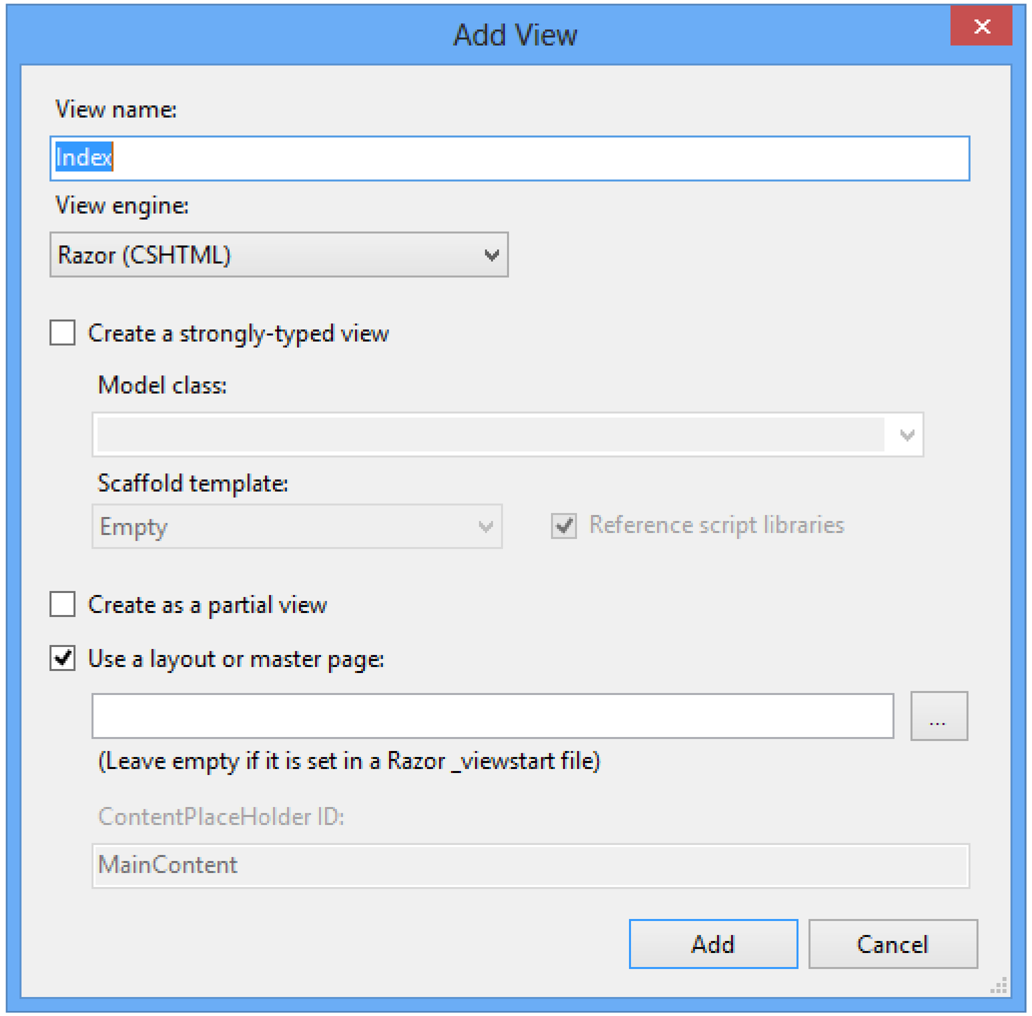
- ♦ List template allows you to create a view that displays a list of model objects.
- ♦ Following code shows an Index() action method that returns an ActionResult object through a call to the View() method of the controller class:

```
public ActionResult Index()
{
        var user = new List<User>();
        //Code to populate the user collection
        return View(user);
}
```

- ♦ This code shows the Index()action method of a controller that returns the result of a call to the View()method.
- ♦ The result of the View()method is an ActionResultobject that renders a view.
- ♦ In this code, ViewData is used to display the values of the Message and CurrentTime keys.

List template

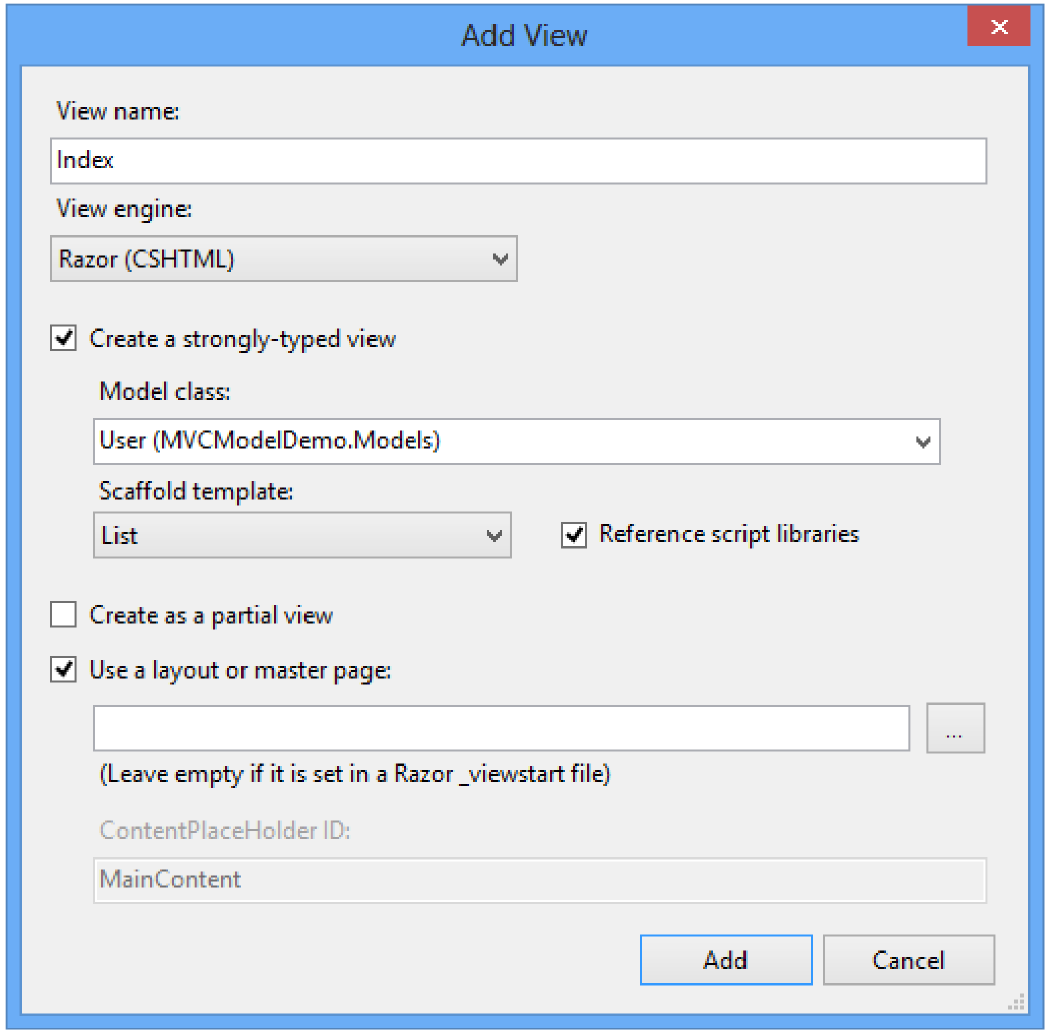
- To create a view using the List template, you need to perform the following steps:
- 1.Right-click inside the action method for which you need to create a view.
- 2.Select **Add View** from the context menu that appears. The **Add View** dialog box is displayed, as shown in the following figure:



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

List template

- 3.Select the **Create a strongly-typed view** check box. This will enable the fields that allow you to specify the model class and scaffolding template.
- 4.Select the model class from the **Model class** drop-down list.
- 5.Select **List** from the **Scaffold templates** drop-down list. Following figure specifies the model class and the scaffolding template:



View name: Index

View engine: Razor (CSHTML)

[✓] Create a strongly-typed view

Model class: User (MVCModelDemo.Models)

Scaffold template: List

[✓] Reference script libraries

[ ] Create as a partial view

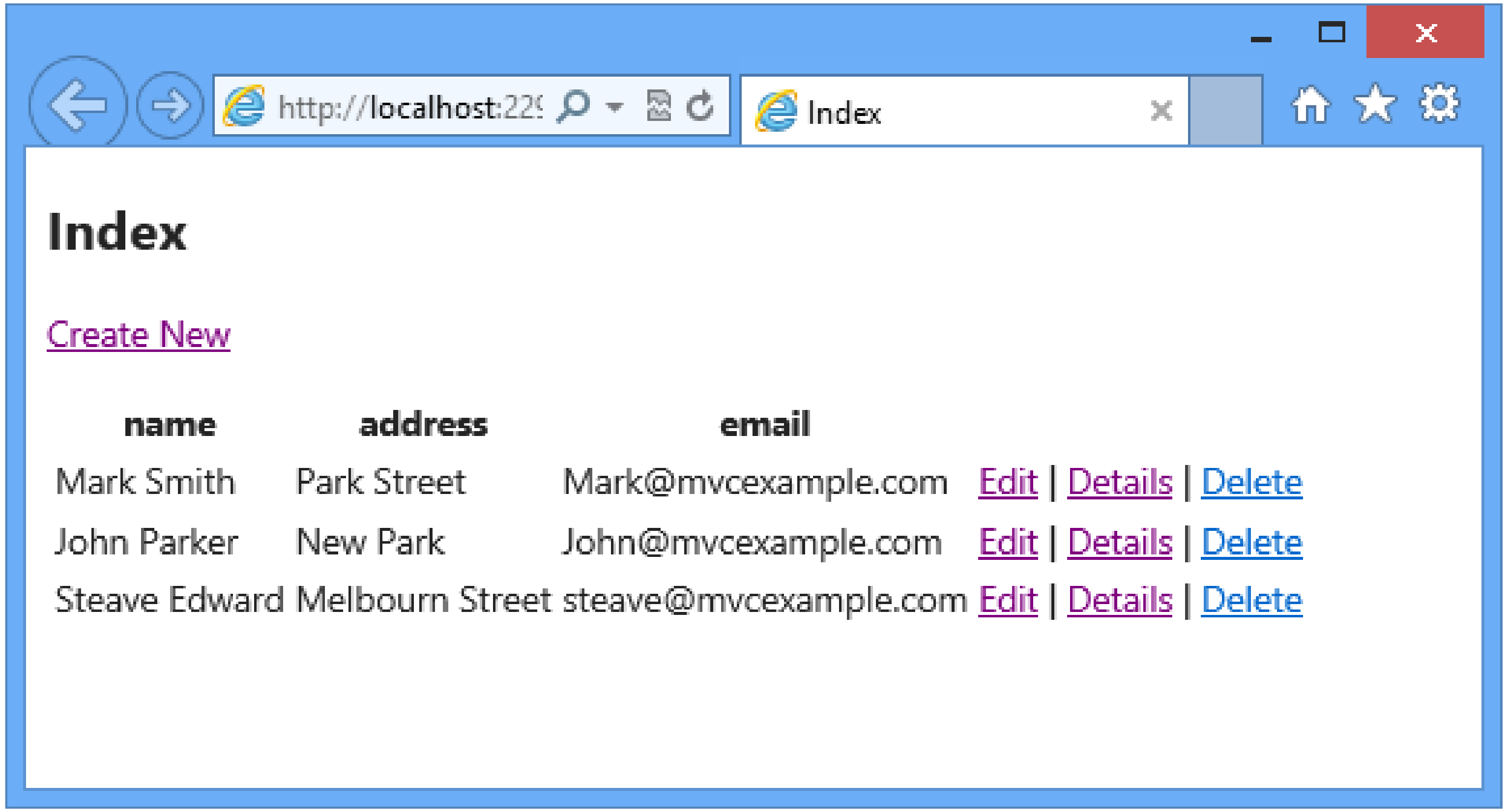
[✓] Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID: MainContent

List template

- 6. Click **Add**. Visual Studio.NET automatically creates the appropriate directory structure and adds the view file to it.
- Following figure shows the output of the auto-generated markup:



Index

Create New

name	address	email	
Mark Smith	Park Street	Mark@mvcexample.com	Edit | Details | Delete
John Parker	New Park	John@mvcexample.com	Edit | Details | Delete
Steave Edward	Melbourn Street	steave@mvcexample.com	Edit | Details | Delete

Create template

- The Create template allows you to generate a view that accepts the details of a new object to be stored in a data store.
- You need to create an action method, to display a view based on the Create template.
- Following code shows creating an action method named, Create()in the HomeControllercontroller:

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

- This code creates the Create() action method that will invoke when a user clicks the Createlink on the view generated using the List template.

Create template

- To create a view using the Create template, you need to perform the following steps:
  - 1.Right-click inside the Create() action method.
  - 2.Select **Add View** from the context menu that appears. The **Add View** dialog box appears.
  - 3.Select the **Create a strongly-typed view** checkbox.
  - 4.Select the model class from the **Model class**drop-down list.
  - 5.Select **Create from the Scaffold templates**drop-down list.
  - 6.Click **Add**. Visual Studio.NET automatically creates a view named, Create in the appropriate directory structure.

Create template

♦ Following code snippet shows the auto-generated markup using the Create template:

```
@model MVCModelDemo.Models.User
@{ ViewBag.Title = “Create”;}
<h2>Create</h2>
@using (Html.BeginForm()) {
@Html.ValidationSummary(true)
<fieldset>
<legend>User</legend>
<div class=”editor-label”>@Html.LabelFor(model => model.name)</div>
<div class=”editor-field”>
      @Html.EditorFor(model => model.name)
      @Html.ValidationMessageFor(model => model.name)
</div>
<div class=”editor-label”>
      @Html.LabelFor(model =>model.address)
</div>
<div class=”editor-field”>
```

Create template

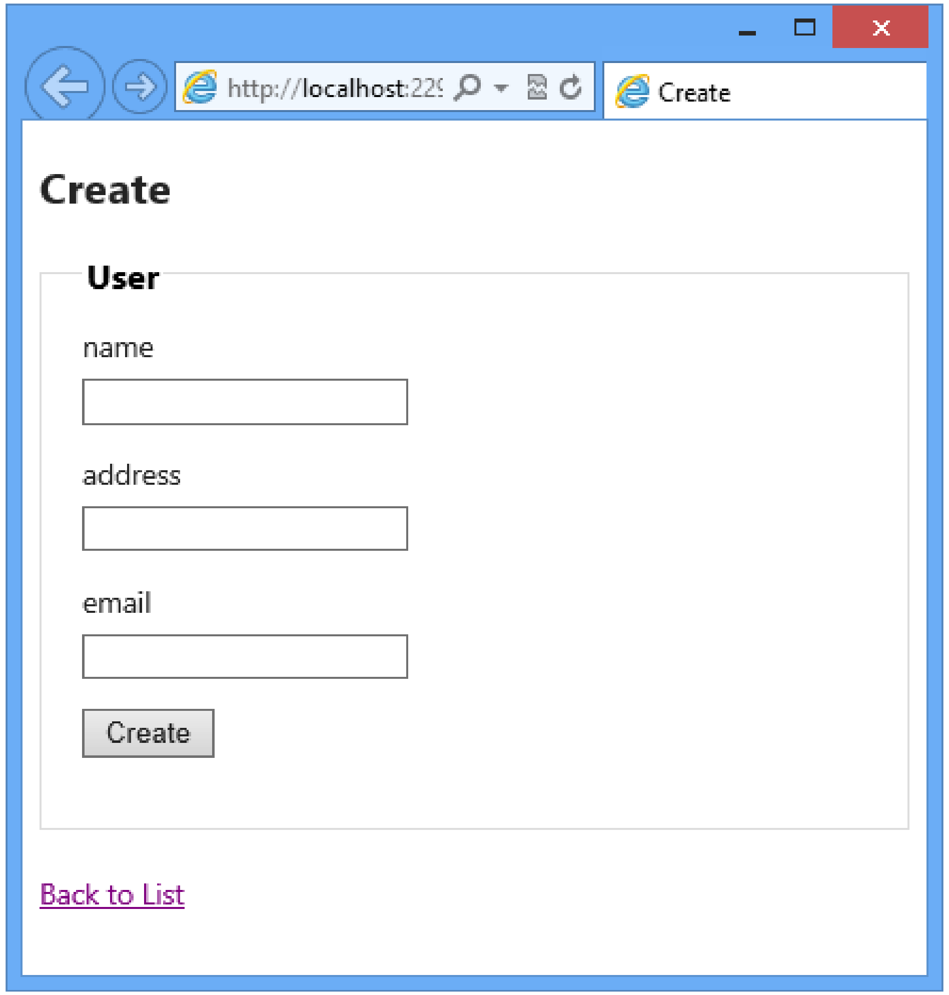
```
        @Html.EditorFor(model =>model.address)
        @Html.ValidationMessageFor(model =>model.address)
</div>
<div class=”editor-label”>@Html.LabelFor(model =>model.email)</div>
<div class=”editor-field”>
        @Html.EditorFor(model =>model.email)
        @Html.ValidationMessageFor(model =>model.email)
</div>
<p><input type=”submit” value=”Create” /></p>
</fieldset>
}
<div>@Html.ActionLink(“Back to List”, “Index”)</div>
@section Scripts {
@Scripts.Render(“~/bundles/jqueryval”)
}
```

Create template

- In this code,
  - The BeginFormhelper method starts a form.
  - The Html.ValidationSummary() validation helper displays the summary of all the error messages at one place.
  - The Html.LabelFor() helper method displays an HTML label element with the name of the property.
  - The Html.EditorFor() helper method displays a textbox that accepts the value of a model property.
  - The Html.ValidationMessageFor() validation helper method displays a validation error message for the associated model property.

Create template

- ♦ Following figure shows the output of auto-generated markup using the Create template:



http://localhost:22

Create

Create

User

name

address

email

Create

Back to List

Edit Template

- The Edit template can be used whenever you want to generate a view that required to be used for modifying the details of an existing object stored in a data store.
- To display a view based on the Edit template, you need to create an action method to pass the model object to be edited to the view.
- Following code shows the action method named Edit()

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

- Once you have created the Edit()action method in the controller, you can use Visual Studio.NET to create the view using the Edit template. To create the view using the Edit template in Visual Studio.NET, you need to select **Edit** from the **Scaffold templates** drop-down list.

Edit Template

- After creating a view named, Editfor the User model using the Edit template, Visual Studio.NET generates the markup for the view.
- In this code, ViewData is used to display the values of the Message and CurrentTime key

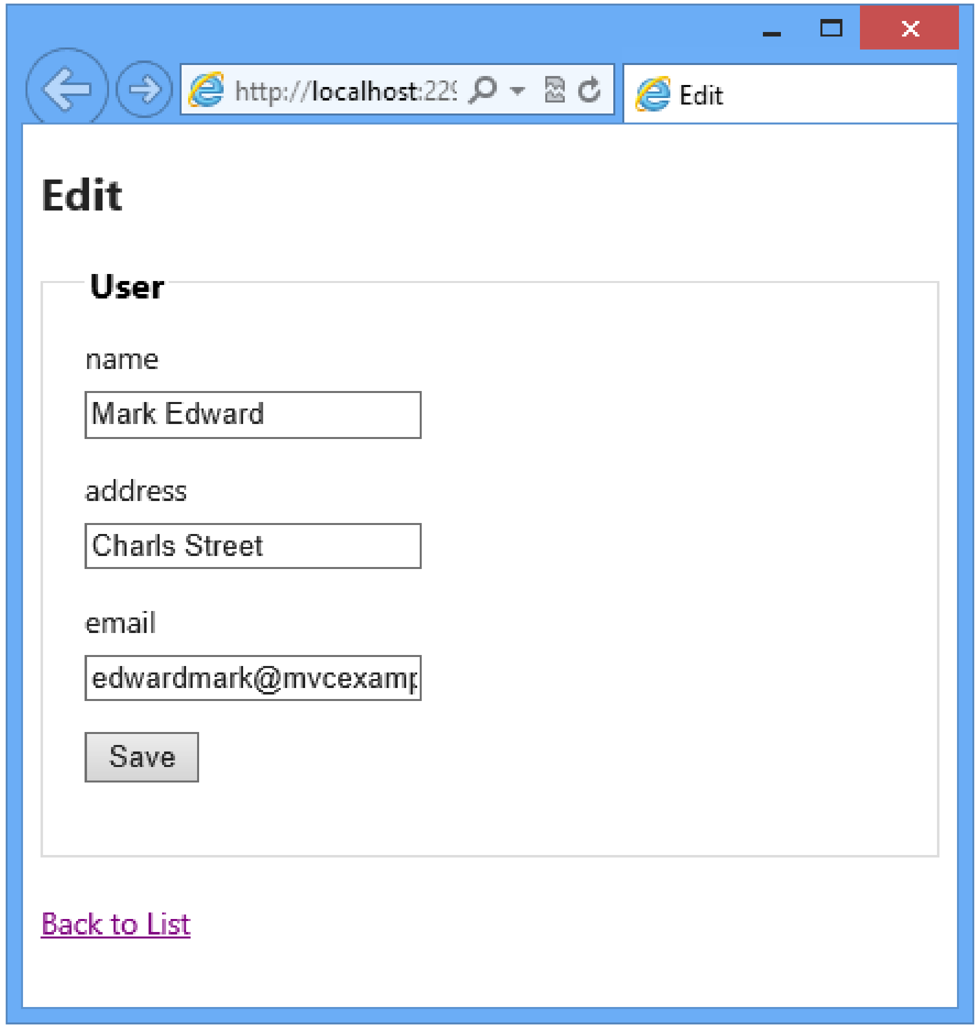
```
@model MVCModelDemo.Models.User
@{ViewBag.Title = “Edit”;}
<h2>Edit</h2>
@using (Html.BeginForm()) {
        @Html.ValidationSummary(true)
        <fieldset>
                <legend>User</legend>
                @Html.HiddenFor(model => model.Id)
                <div class=”editor-label”> @Html.LabelFor(model => model.name)</div>
                <div class=”editor-field”>
                                @Html.EditorFor(model => model.name)
                                @Html.ValidationMessageFor(model => model.name)
                </div>
                <div class=”editor-label”>@Html.LabelFor(model =>model.address)</div>
```

Edit Template

```
        <div class=”editor-field”>
                @Html.EditorFor(model =>model.address)
                @Html.ValidationMessageFor(model =>model.address)
        </div>
        <div class=”editor-label”>@Html.LabelFor(model =>model.email)</div>
        <div class=”editor-field”>
                @Html.EditorFor(model =>model.email)
                @Html.ValidationMessageFor(model =>model.email)
        </div>
        <p> <input type=”submit” value=”Save” /></p>
        </fieldset>
}
<div>@Html.ActionLink(“Back to List”, “Index”)</div>
@section Scripts {
@Scripts.Render(“~/bundles/jqueryval”)
}
```

Edit template

- The Html.LabelFor()helper method displays an HTML label element with the name of the property.
- The Html.EditorFor()helper method displays a textbox to accept the value of a model property.
- The Html.ValidationMessageFor() helper method displays a validation error message.
- Following figure shows the output of creating the view using the Edit template:



Edit

User

name

Mark Edward

address

Charls Street

email

edwardmark@mvcexamp

Save

Back to List

Detail Template

- The Details template allows you to create a view that displays details of the Usermodel.
- Once you have created the Details()action method in the controller, and a view using the Details template in Visual Studio.NET, it generates the markup for the view.
- Following code snippet shows the auto-generated markup when you create the view using the Details template:

```
@model MVCModelDemo.Models.User
@{ViewBag.Title = “Details”;}
<h2>Details</h2>
<fieldset>
      <legend>User</legend>
      <div class=”display-label”>@Html.DisplayNameFor(model => model.name)</div>
      <div class=”display-field”>       @Html.DisplayFor(model => model.name)</div>
```

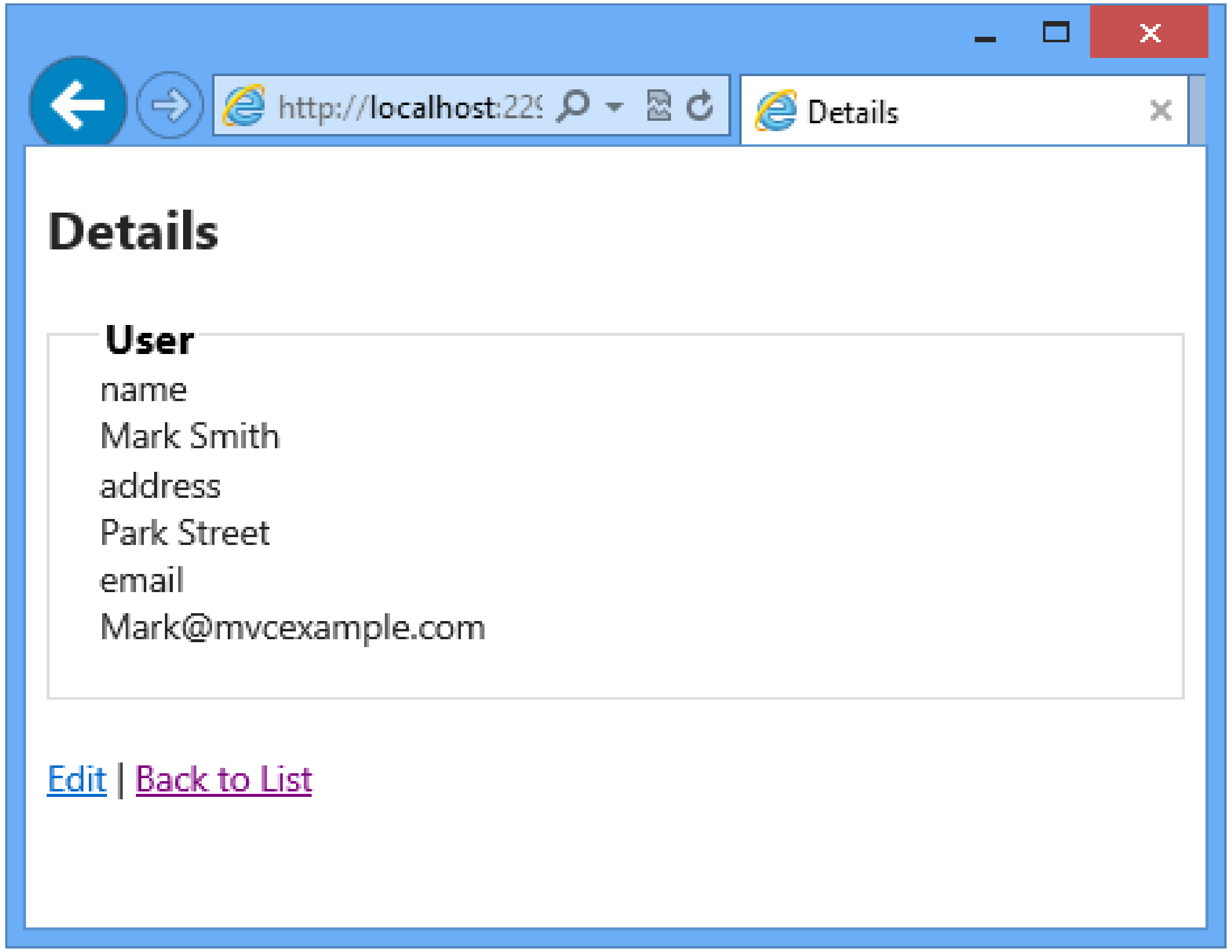
Detail Template

```
        <div class=”display-label”>@Html.DisplayNameFor(model =>model.address)</div>
        <div class=”display-field”>@Html.DisplayFor(model =>model.address)</div>
        <div class=”display-label”>@Html.DisplayNameFor(model =>model.email)</div>
        <div class=”display-field”>@Html.DisplayFor(model =>model.email)</div>
</fieldset>
<p>
        @Html.ActionLink(“Edit”, “Edit”, new { id=Model.Id }) |
        @Html.ActionLink(“Back to List”, “Index”)
</p>
```

- In this code, the Html.DisplayNameFor()helper method displays the name of model properties and the Html.DisplayFor()helper method displays the values of the model properties.

Detail Template

♦ Following figure shows the output of auto-generated markup for Details template:



http://localhost:22

Details

Details

User

name

Mark Smith

address

Park Street

email

Mark@mvcexample.com

Edit | Back to List

Delete Template

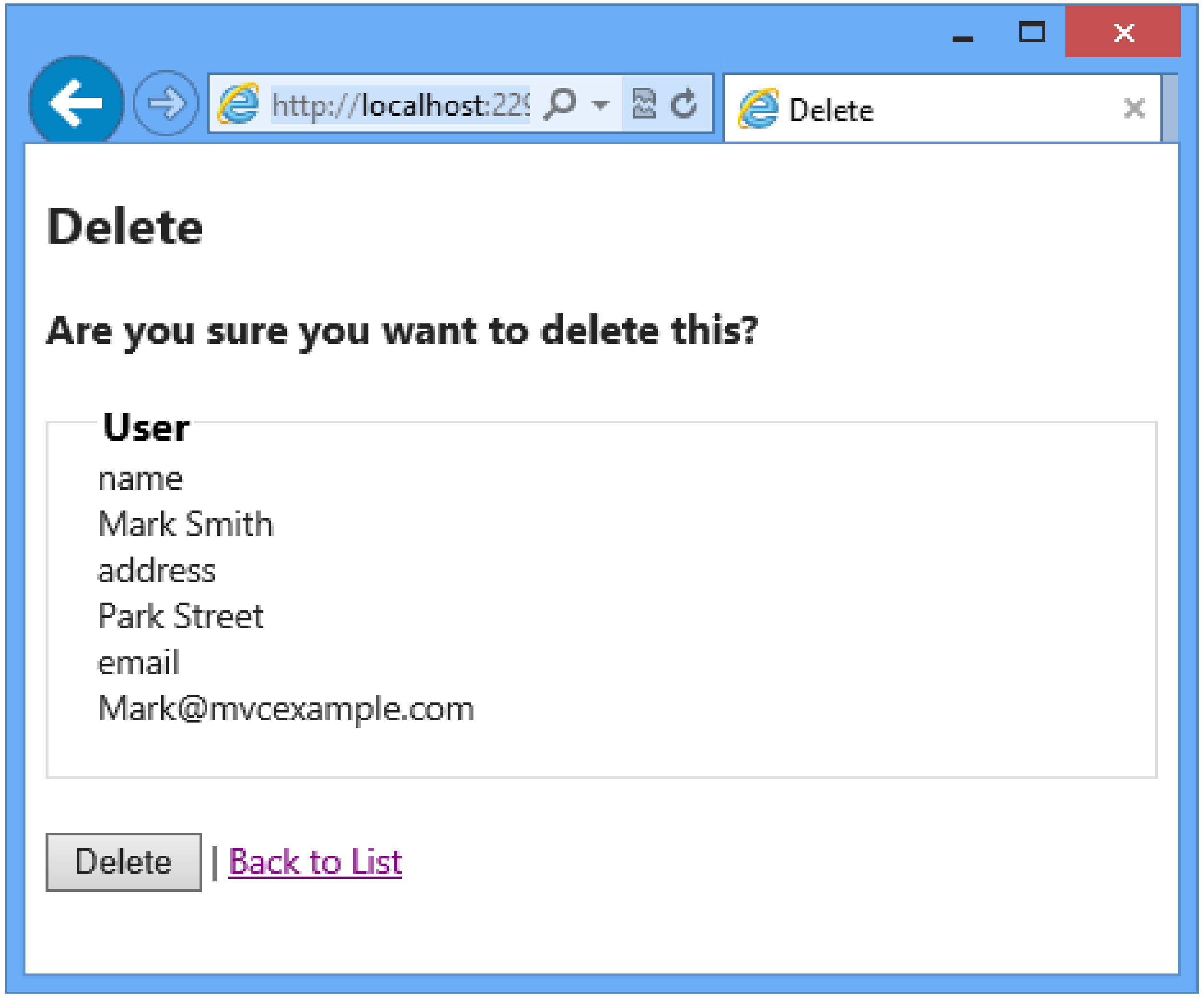
- Allows generating a view that allows a user to delete an existing object from a data store.
- To create a view based on the Delete template, first you need to create an action method that passes the model object to be deleted to the view.
- After you create the Delete()action method, and a view for the User model using the Delete template in Visual Studio.NET, it generates the markup for the view.
- Following code snippet shows the auto-generated markup when you create a view using the Delete template:

Delete Template

```
@model MVCModelDemo.Models.User
@{ViewBag.Title = “Delete”;}
<h2>Delete</h2>
<h3>Are you sure you want to delete this?</h3>
<fieldset>
	<legend>User</legend>
	<div class=”display-label”>@Html.DisplayNameFor(model => model.name)</div>
	<div class=”display-field”>@Html.DisplayFor(model => model.name)</div>
	<div class=”display-label”>@Html.DisplayNameFor(model =>model.address)</div>
	<div class=”display-field”>@Html.DisplayFor(model =>model.address)</div>
	<div class=”display-label”>@Html.DisplayNameFor(model =>model.email)</div>
	<div class=”display-field”>@Html.DisplayFor(model =>model.email)</div>
</fieldset>
@using (Html.BeginForm()) {
	<p>
	<input type=”submit” value=”Delete” /> |@Html.ActionLink(“Back to List”, “Index”)
	</p>
}
```

Delete Template

- Following figure shows the output of creating view using the Delete template:



Delete

Are you sure you want to delete this?

User

name
Mark Smith
address
Park Street
email
Mark@mvcexample.com

Summary

- In an ASP.NET MVC application, a model represents data associated with the application. In the MVC pattern, there are three types of models, where each model has specific purpose.
- The MVC Framework provides helper methods that you can use only in strongly typed views.
- The process of mapping the data in an HttpRequest object to a model object is known as model binding.
- The MVC Framework provides a model binder that performs model binding in application.
- The ASP.NET MVC Framework provides a feature called scaffolding that allows you to generate views automatically.
- Visual Studio.NET simplifies the process of creating views for an action method using the different scaffolding template.