

## 2.3 Model trong ASP.NET MVC

- Định nghĩa và mô tả model
- Giải thích cách tạo 1 model
- Mô tả cách truyền dữ liệu model từ controller sang view
- Giải thích cách tạo ra strongly typed model
- Giải thích vai trò của model binder
- Giải thích cách sử dụng scaffolding trong Visual Studio .NET

- ◆ Trong ứng dụng ASP.NET MVC, 1 model:
  - ◆ Là 1 class chứa các thuộc tính biểu thị dữ liệu của ứng dụng
  - ◆ Biểu diễn dữ liệu được liên kết với ứng dụng.

ASP.NET MVC Framework dựa theo mô hình MVC.

- ◆ Mô hình MVC định nghĩa ra 3 kiểu model, mỗi kiểu đều có chức năng riêng biệt:
  - ◆ **Data model:** Biểu diễn các class tương tác với cơ sở dữ liệu. Các data model là tập hợp các class có thể tuân theo phương pháp tiếp cận database-first hoặc tiếp cận code-first
  - ◆ **Business model:** Biểu diễn các class triển khai một chức năng biểu thị business logic của ứng dụng. Khi xử lý business logic, các class của model có thể tương tác với các class chứa trong data model để truy xuất và lưu dữ liệu trong cơ sở dữ liệu
  - ◆ **View model:** Biểu diễn các class cung cấp thông tin được truyền giữa controller và view.

# Tạo một Model

- ◆ Để tạo model trong ứng dụng ASP.NET MVC, ta cần:
  - ◆ Tạo 1 public class.
  - ◆ Khai báo các thuộc tính là public với mỗi thông tin model biểu diễn
- ◆ Đoạn code sau biểu diễn 1 class model ***User***:

## Code Snippet:

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

Đoạn code tạo một model class ***User*** chứa các thuộc tính ***Id***, ***name***, ***address***, và ***email*** được khai báo là public

# Truy cập Model bên trong controller

- ◆ Trong ứng dụng ASP.NET MVC, khi người dùng yêu cầu thông tin, yêu cầu được tiếp nhận bởi phương thức hành động.
- ◆ Trong phương thức hành động, ta cần truy cập model lưu trữ dữ liệu
- ◆ Để truy cập model trong phương thức, ta cần tạo một đối tượng của class model và truy xuất hoặc đặt các giá trị thuộc tính của đối tượng
- ◆ Đoạn code sau biểu thị ví dụ khi tạo một đối tượng của class model trong phương thức hành động **Index()** :

## Code Snippet:

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

Trong đoạn code này, **user** là một đối tượng của class **User**. Các giá trị thuộc tính của model được đặt thành dữ liệu liên quan đến người dùng như *name*, *address*, *email*.

# Truyền dữ liệu Model từ controller tới view

- ◆ Khi đã truy cập tới model bên trong 1 controller, bạn cần làm cho dữ liệu model có thể truy cập được tới 1 view, từ đó view có thể hiển thị dữ liệu tới người dùng
- ◆ Để làm điều này, ta truyền đối tượng model tới view trong khi gọi đến view
- ◆ Ta cũng có thể truyền:
  - ◆ Một đối tượng đơn lẻ
  - ◆ Một tập các đối tượng

# Truyền dữ liệu Model từ controller tới view

- ◆ Trong 1 phương thức hành động, ta tạo một đối tượng model sau đó truyền đối tượng tới 1 view bằng *ViewBag*
- ◆ Đoạn code sau biểu thị cách truyền dữ liệu model từ controller tới view bằng cách sử dụng *ViewBag*:

## Code Snippet:

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

Một đối tượng của class model ***User*** được tạo và khởi tạo với các giá trị. Đối tượng sau đó sẽ được chuyển tới view bằng *ViewBag*

# Truyền dữ liệu model từ controller tới view

- ◆ Ta có thể truy cập dữ liệu của đối tượng model lưu trong đối tượng *ViewBag* từ bên trong view.
- ◆ Đoạn code sau biểu thị việc truy cập dữ liệu của đối tượng model được lưu trong đối tượng *ViewBag*:

## Code Snippet:

```
<!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>
```

Trong đoạn code, view truy cập và hiển thị các thuộc tính của đối tượng *User* lưu trong *ViewBag*



# Truyền dữ liệu model từ controller tới view

- ◆ Ví dụ khi truyền 1 tập các đối tượng tới view:

## Code Snippet:

```
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 dữ liệu model từ controller tới view

- ◆ Đoạn code trên:
  - ◆ Đoạn code tạo và khởi tạo 3 đối tượng của class model **User**
  - ◆ Sau đó, một tập **List<User>** được tạo và các đối tượng model được thêm vào tập này
  - ◆ Cuối cùng, tập được truyền tới view bằng *ViewBag*
  - ◆ Khi bạn truyền 1 tập các đối tượng tới view bằng *ViewBag*:
    - ◆ Bạn có thể truy xuất tập này ở bên trong view.
    - ◆ Sử dụng vòng lặp với tập để truy xuất tới từng đối tượng model
    - ◆ Có thể truy cập thuộc tính của chúng.

# Truyền dữ liệu Model từ controller tới view

- ◆ Đoạn code sau biểu diễn cách hiển thị thuộc tính của các đối tượng model ở bên trong tập :

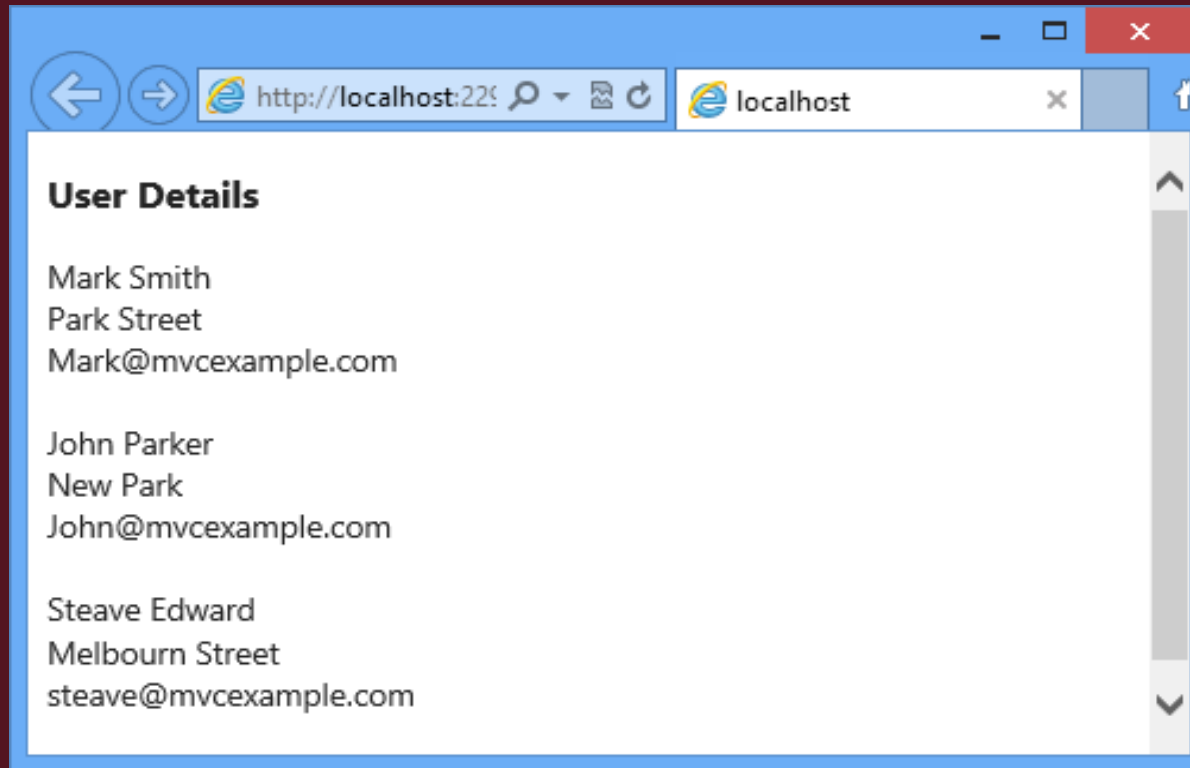
## Code Snippet:

```
<!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>
```

Đoạn code sử dụng foreach để lặp với các đối tượng model có trong tập được lưu trong *ViewBag*. Sau đó, với mỗi đối tượng model, các thuộc tính sẽ được kết xuất như 1 phản hồi.

# Truyền dữ liệu Model từ controller tới view

- ◆ Kết quả của đoạn code trên:



Đoạn code trên sử dụng vòng lặp foreach để duyệt qua các phần tử ở trong tập và hiển thị các thuộc tính của chúng.

# Truyền dữ liệu Model từ controller tới view

- ◆ Đoạn code sau biểu diễn cách hiển thị thuộc tính của các đối tượng model ở bên trong tập :

## Code Snippet:

```
<!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>
```

Đoạn code sử dụng foreach để lặp với các đối tượng model có trong tập được lưu trong *ViewBag*. Sau đó, với mỗi đối tượng model, các thuộc tính sẽ được kết xuất như 1 phản hồi.

# Truyền dữ liệu model từ controller tới view

- ◆ Lấy thông tin của người dùng trong view:

## Code Snippet:

```
<!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 />
    }
</body>
</html>
```

Đoạn code trên chỉ ra cách truy xuất dữ liệu người dùng được truyền tới view bằng cách truyền tập các đối tượng dưới dạng 1 tham số.

# Sử dụng Strong Typing

- ◆ Khi truyền dữ liệu model từ controller tới view, view không thể xác định chính xác kiểu của dữ liệu. Thêm vào đó, ta cũng không có được các lợi ích của việc strong typing và việc kiểm tra thời gian biên dịch của code.
- ◆ Giải pháp được đưa ra là chuyển kiểu dữ liệu model thành một kiểu nhất định

## Code Snippet:

```
<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 đoạn code, đối tượng Model được ép thành kiểu ***MVCModelDemo.Models.User***. Kết quả của việc ép kiểu: đối tượng ***user*** được tạo là một đối tượng với kiểu ***MVCModelDemo.Models.User***, và có thể kiểm tra thời gian biên dịch của code

# Sử dụng Strong Typing

- ◆ Ta có thể bỏ qua việc ép kiểu đối tượng bằng cách tạo strongly typed view.
- ◆ Một strongly type view xác định kiểu của một model mà nó yêu cầu bằng cách sử dụng từ khóa '@model'.
- ◆ Cú pháp của strong typed view:

## Syntax:

```
@model <model_name>
```

Trong đó,

- **model\_name**: tên đầy đủ của class model.
- ◆ Khi sử dụng '@model', ta có thể truy cập các thuộc tính của đối tượng trong view.



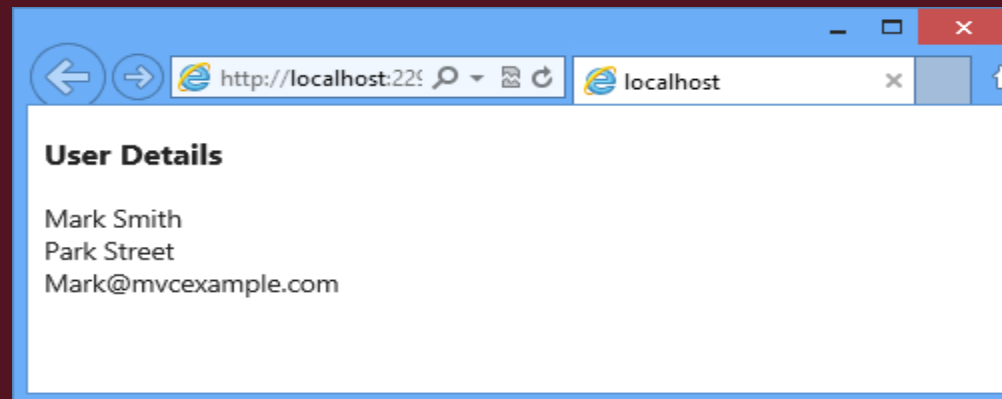
# Sử dụng Strong Typing

- Truy cập các thuộc tính của đối tượng model bằng cách sử dụng từ khóa '@model':

## Code Snippet:

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

- Kết quả:



# Sử dụng Strong Typing

- ◆ Đôi lúc, ta cần truyền một tập các đối tượng tới một view.
- ◆ Những trường hợp như vậy, ta sử dụng từ khóa '@model'.
- ◆ Ví dụ:

## Code Snippet:

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

- ◆ Đoạn code sử dụng '@model' cho một tập các đối tượng ***User***.
- ◆ Sau khi truyền một tập các đối tượng model, ta có thể truy cập chúng trong 1 view.

# Sử dụng Strong Typing

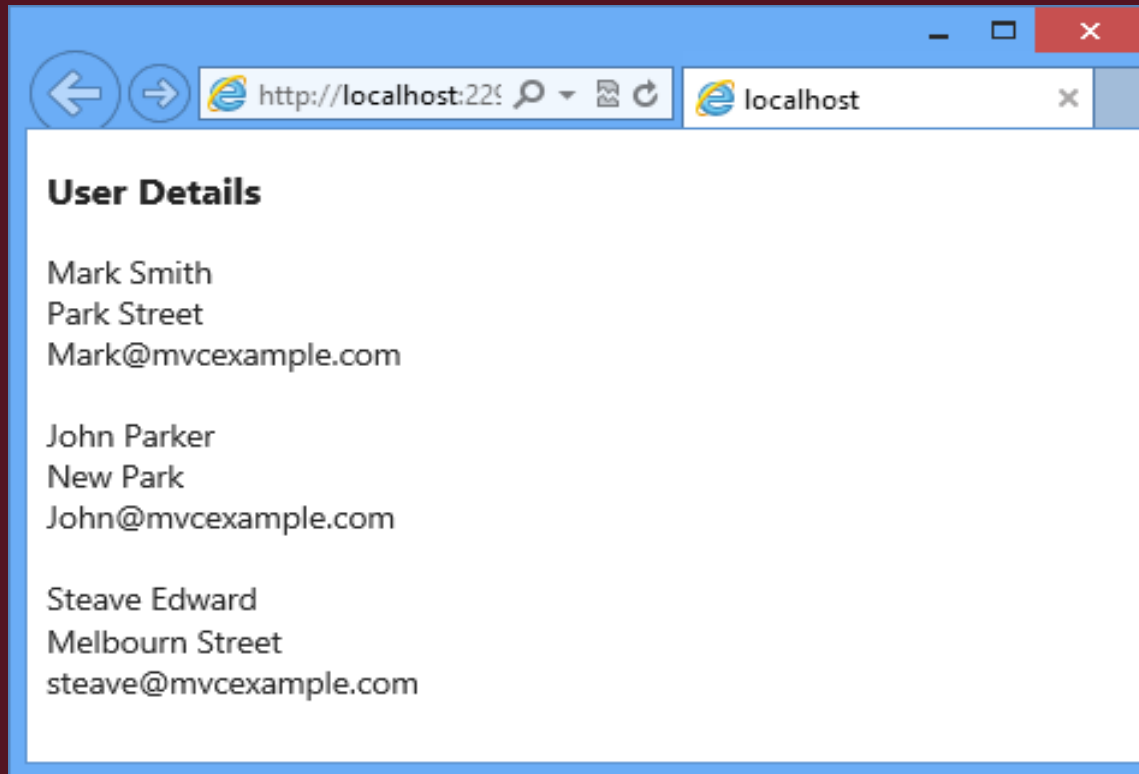
- ◆ Truy cập một tập các model *User* trong một view:

## Code Snippet:

```
@model IEnumerable<MVCDemo.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>
```

# Sử dụng Strong Typing

- ◆ Kết quả của đoạn code trên:



# Phương thức hỗ trợ HTML trong Strongly Typed View

- ◆ The MVC Framework:
  - ◆ Cho phép các phương thức hỗ trợ HTML sử dụng trong view liên kết trực tiếp với các thuộc tính của model trong 1 strongly typed view.
  - ◆ Cung cấp các phương thức hỗ trợ sử dụng chỉ trong Strongly Typed View.
- ◆ Dưới đây là bảng các phương thức hỗ trợ HTML chỉ dùng cho Strongly Typed View:

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

# Phương thức hỗ trợ HTML trong Strongly Typed View

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

# Phương thức hỗ trợ HTML trong Strongly Typed View

- ◆ Ví dụ sau biểu diễn phương thức hỗ trợ HTML trong Strongly Typed View:

## Code Snippet:

```
@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>
```

# Phương thức hỗ trợ HTML trong Strongly Typed View

## Code Snippet:

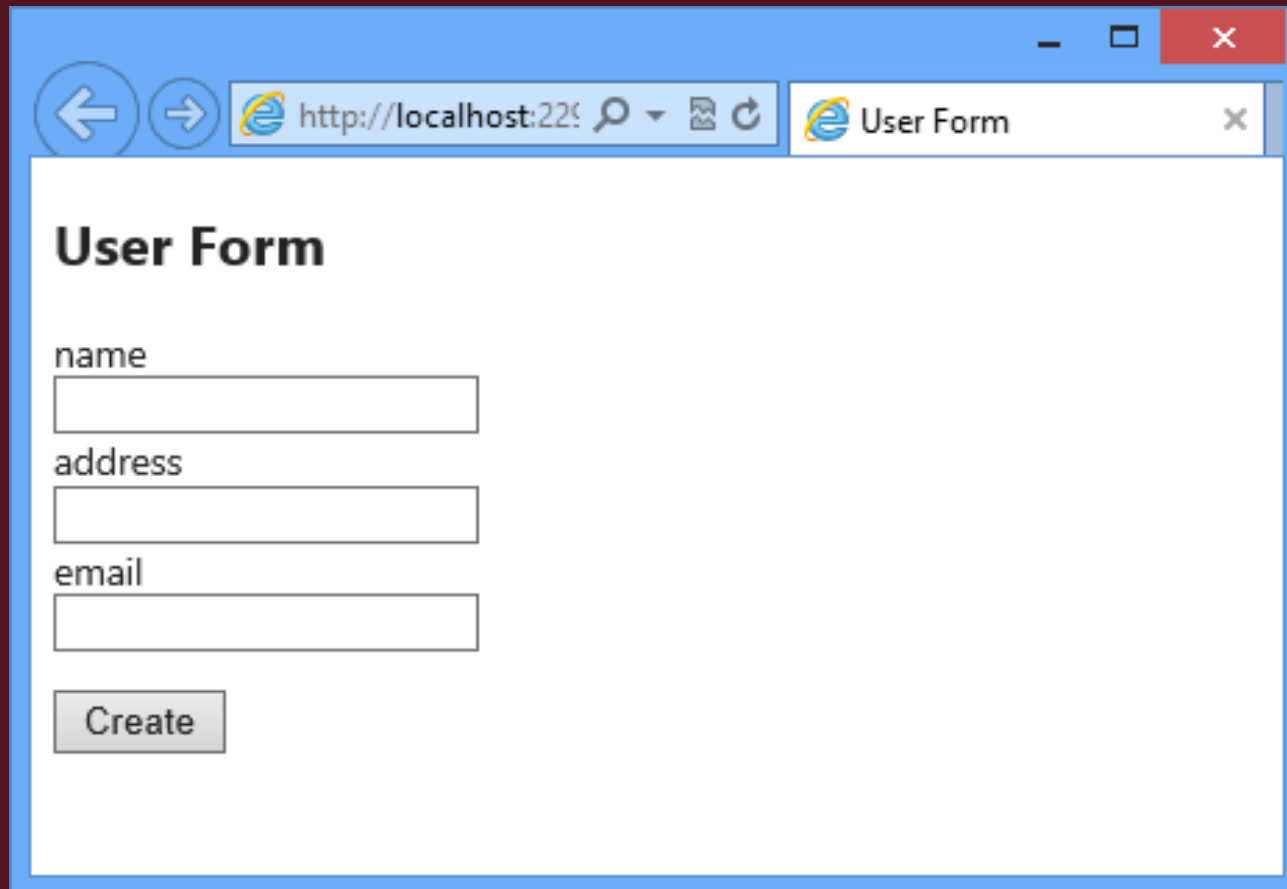
```
@Html.EditorFor(model =>model.address)
<div>
@Html.LabelFor(model =>model.email)
</div>
<div>
@Html.EditorFor(model =>model.email)
</div>
<p>
<input type="submit" value="Create" />
</p> }
```

Trong đoạn code trên, phương thức *Html.LabelFor()* được sử dụng để hiển thị các label dựa trên tên thuộc tính của model. *Html.EditorFor()* được sử dụng để chỉnh sửa các trường của các thuộc tính Model



# Phương thức hỗ trợ HTML trong Strongly Typed View

- ◆ Kết quả:



The screenshot shows a web browser window with the address bar displaying `http://localhost:229` and the page title `User Form`. The page content includes a heading **User Form**, followed by three text input fields labeled `name`, `address`, and `email`. At the bottom of the form is a `Create` button.

## Vai trò của Model binder

- ◆ Khi một người dùng xác nhận thông tin trong 1 form bên trong 1 strongly typed view, ASP.NET MVC sẽ tự động kiểm tra đối tượng HttpRequest và ánh xạ thông tin gửi tới các trường trong đối tượng model.
- ◆ Quy trình ánh xạ thông tin trong đối tượng HttpRequest tới đối tượng Model được gọi là model binding
- ◆ Các lợi ích của model binding:
  - ◆ Tự động trích xuất dữ liệu từ đối tượng HttpRequest
  - ◆ Tự động chuyển kiểu dữ liệu
  - ◆ Xác thực dữ liệu dễ dàng

## Vai trò của Model binder

- ◆ MVC Framework cung cấp một model binder thực hiện model binding trong ứng dụng.
- ◆ Class ***DefaultModelBinder*** triển khai model binder trong MVC Framework.
- ◆ 2 vai trò chính của model binder :
  - ◆ Ràng buộc yêu cầu với các giá trị nguyên thủy
  - ◆ Ràng buộc yêu cầu với các đối tượng

# Binding tới giá trị nguyên thủy

- ◆ Để hiểu cách model binder ràng buộc yêu cầu với các giá trị nguyên thủy, xét trường hợp bạn tạo một form đăng nhập tiếp nhận thông tin đăng nhập của người dùng.
- ◆ Trước hết, ta cần tạo một model **Login** trong ứng dụng.
- ◆ Ví dụ về class model **Login**:

## Code Snippet:

```
public class Login
{
    public string userName { get; set; }

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

Model Login có 2 thuộc tính là **userName** và **password**

Sau khi tạo model class, ta tạo view **Index.cshtml** để hiển thị form đăng nhập

# Binding tới giá trị nguyên thủy

## ♦ Tập tin *Index.cshtml* :

### Code Snippet:

```
@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 tới giá trị nguyên thủy

- ◆ Khi bạn đã tạo view, cần tạo 1 class controller chứa phương thức hành động **Index()** để hiển thị view. Ta cũng cần tạo một phương thức hành động **Index()** khác với thuộc tính **HttpPost** để nhận dữ liệu đăng nhập của người dùng.

## Code Snippet:

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

# Binding tới giá trị nguyên thủy

- ◆ Ở đoạn code trên:
  - ◆ Phương thức ***Index()*** đầu tiên trả về view ***Index.cshtml*** hiển thị form đăng nhập
  - ◆ Phương thức ***Index()*** thứ hai được đánh dấu với thuộc tính **HttpPost**.  
Phương thức này chấp nhận 2 tham số có kiểu nguyên thủy. Phương thức ***Index()*** so sánh các tham số với các giá trị có sẵn và trả về một thông điệp nếu phép so sánh này trả về true. Trường hợp còn lại, ***Index()*** trả về view ***Index.cshtml***.
- ◆ Khi người dùng xác nhận dữ liệu đăng nhập, Model binder mặc định ánh xạ các giá trị của các trường ***userName*** và ***password*** tới các tham số kiểu nguyên thủy của phương thức ***Index()***. Trong phương thức ***Index()***, có thể biểu diễn yêu cầu xác thực và trả về kết quả

## Binding tới đối tượng

- ◆ Để hiểu cách model binder kết nối yêu cầu tới đối tượng, ta xét trường hợp khi tạo form đăng nhập.
- ◆ Ta đã tạo được model *Login* và view *Index.cshtml*..
- ◆ Để kết nối yêu cầu với đối tượng, bạn cần cập nhật class controller để nó tiếp nhận đối tượng *Login* như một tham số thay vì đối tượng *HttpRequest*.



- ◆ Class controller được cập nhật thêm như sau:

## Code Snippet:

```
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();  
        }  
    }  
}
```

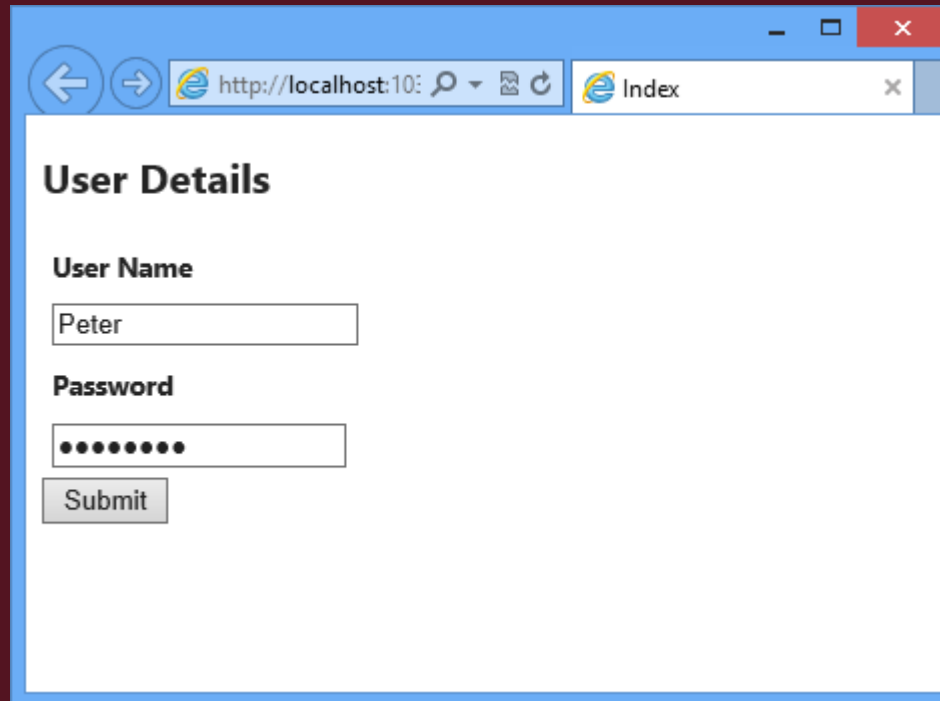
Trong đoạn code trên, phương thức ***Index()*** đầu tiên trả về view ***Index.cshtml*** hiển thị form đăng nhập. Phương thức ***Index()*** thứ hai tự động xóa dữ liệu từ đối tượng ***HttpRequest*** và đưa vào đối tượng ***Login***.

## Binding tới đối tượng

- ◆ Khi người dùng xác nhận dữ liệu đăng nhập, phương thức ***Index()*** xác nhận ***username*** và ***password*** được truyền vào đối tượng ***Login***.
  - ◆ Khi xác nhận thành công, View hiển thị một tin nhắn chào mừng.
  - ◆ Khi truy cập ứng dụng từ trình duyệt, View ***Index.cshtml*** hiển thị form đăng nhập
1. Nhập '*Peter*' vào phần **User Name** và '*pass@123*' vào phần **Password**

# Binding tới đối tượng

- ◆ Form đăng nhập với 2 trường **User Name** và **Password**:



The screenshot shows a web browser window with a blue title bar. The address bar displays 'http://localhost:10...' and the page title is 'Index'. The main content area has a white background and is titled 'User Details' in bold black text. Below the title, there are two input fields: 'User Name' with the text 'Peter' and 'Password' with masked characters (dots). A 'Submit' button is located below the password field.

**User Details**

**User Name**

Peter

**Password**

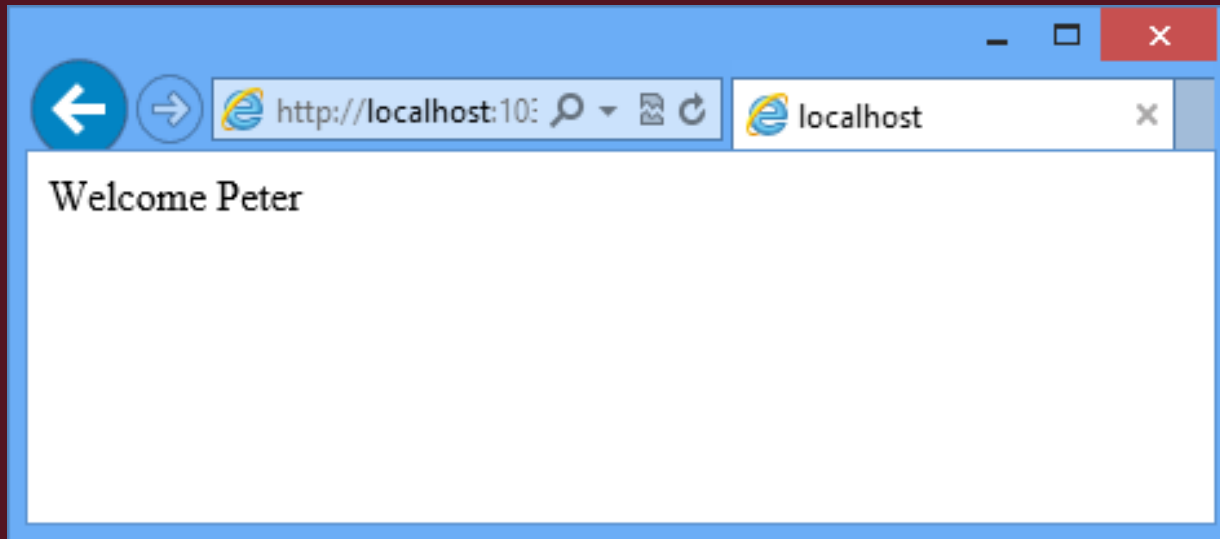
.....

Submit

2. Nhấn ***Submit***. Thông báo '*Welcome Peter*' xuất hiện

# Binding tới đối tượng

- ◆ Kết quả:



# Visual Studio.NET Scaffolding

- ◆ ASP.NET Framework cung cấp một tính năng gọi là scaffolding, cho phép tạo các view một cách tự động..
- ◆ Theo quy ước, scaffolding sử dụng tên đặc thù cho các view.
- ◆ Sau khi tạo 1 view, nó lưu mã code được tạo tự động ở các mục tương ứng để ứng dụng hoạt động.
- ◆ Có 5 template mà tính năng scaffolding cung cấp để tạo ra các view::
  - **List:** tạo ra markup hiển thị danh sách các đối tượng model.
  - **Create:** tạo ra markup để thêm một đối tượng mới vào danh sách.
  - **Edit:** tạo ra markup để chỉnh sửa một đối tượng model có sẵn.
  - **Details:** tạo ra markup hiển thị thông tin của một đối tượng model có sẵn
  - **Delete:** tạo ra markup để xóa một đối tượng model có sẵn.
- ◆ Trong đoạn code, ***ViewData*** được sử dụng để hiển thị các giá trị của ***Message*** và ***CurrentTime***.

# List Template

- ◆ Sử dụng List template để tạo một view hiển thị 1 danh sách các đối tượng model.
- ◆ Đoạn code sau biểu diễn phương thức hành động ***Index()*** trả về một đối tượng ***ActionResult*** thông qua việc gọi tới phương thức ***View()*** của class controller:

## Code Snippet:

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

Đoạn code biểu thị phương thức ***Index()*** của 1 controller trả về kết quả của việc gọi tới phương thức ***View()***. Kết quả của phương thức ***View()*** là một đối tượng ***ActionResult*** kết xuất 1 view.

# List Template

- ◆ Visual Studio .NET đơn giản hóa quá trình tạo 1 view cho một phương thức hành động sử dụng scaffolding template. Để tạo ra một view sử dụng List, làm theo các bước sau:
  1. Nhấn chuột phải bên trong phương thức hành động mà bạn muốn tạo view
  2. Chọn **Add View** từ thanh menu hiện ra. Hộp thoại **Add View** xuất hiện

The screenshot shows the 'Add View' dialog box in Visual Studio. The 'View name' field contains 'Index'. The 'View engine' is set to 'Razor (CSHTML)'. The 'Create a strongly-typed view' checkbox is unchecked. The 'Model class' field is empty. The 'Scaffold template' is set to 'Empty'. The 'Reference script libraries' checkbox is checked. The 'Create as a partial view' checkbox is unchecked. The 'Use a layout or master page:' checkbox is checked. The text box for the layout name is empty, and the 'ContentPlaceHolder ID' is set to 'MainContent'. The 'Add' and 'Cancel' buttons are at the bottom right.

# List Template

3. Chọn **Create a strongly-typed view**. Các trường lúc này có thể cho phép người dùng xác định class model và scaffolding template
4. Chọn class model từ danh sách Model class
5. Chọn **List** từ danh sách **Scaffold template**

**Add View**

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

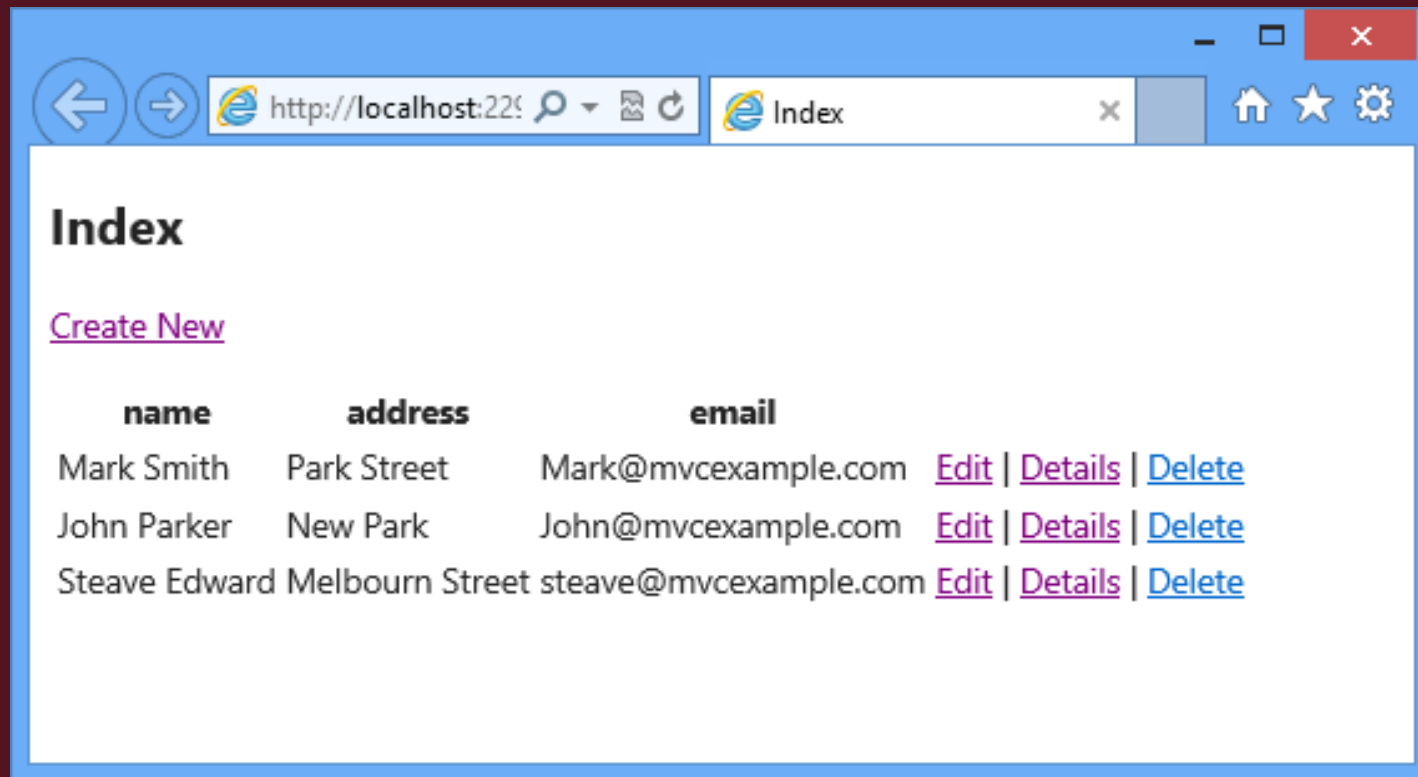
Add Cancel



# List Template

6. Chọn **Add**. Visual Studio .NET tự động tạo cấu trúc thư mục phù hợp và thêm tập tin view vào đó.

◆ Kết quả:



# Create Template

- ◆ Sử dụng Create template để tạo 1 view tiếp nhận các thuộc tính của một đối tượng mới được lưu trong kho dữ liệu.
- ◆ Ta cần tạo một phương thức hành động để hiển thị 1 view dựa trên Create template.
- ◆ Đoạn code dưới đây biểu thị việc tạo một phương thức tên **Create ()** trong controller **Home**:

## Code Snippet:

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

- ◆ Đoạn code tạo ra phương thức **Create ()**. Phương thức này sẽ được gọi đến khi 1 người dùng nhấn vào link **Create** trên view được tạo ra từ List template

## Create Template

Ta cần tạo 1 view sử dụng Create template theo các bước sau:

1. Nhấn chuột phải bên trong phương thức **Create()**
2. Chọn **Add View** từ thanh menu hiện ra. Hộp thoại Add View xuất hiện
3. Chọn **Create a strongly-typed view**
4. Chọn class model từ danh sách Model class
5. Chọn **Create** from the Scaffold templates
6. Chọn **Add**. Visual Studio .NET tự động tạo một view có tên **Create** ở trong cấu trúc thư mục phù hợp

- ◆ Đoạn code dưới đây biểu diễn markup được tạo tự động sử dụng Create template:

## Code Snippet:

```
@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">
```

## Code Snippet:

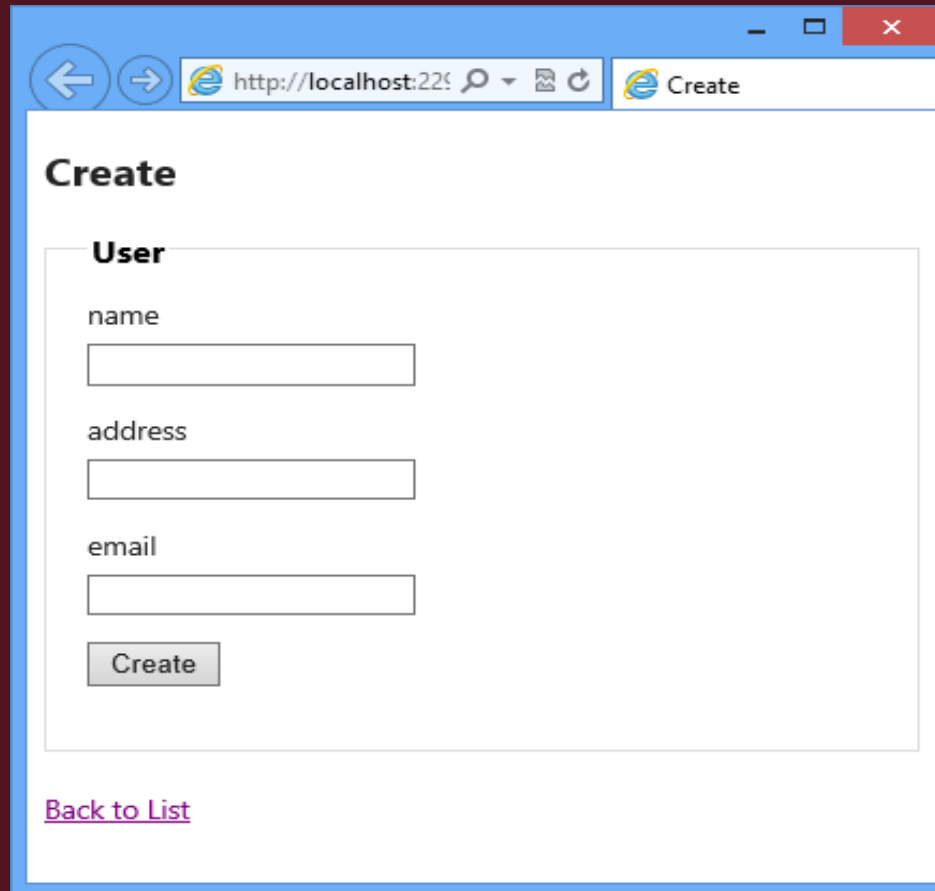
```
@Html.EditorFor(model =>model.address)
@Html.ValidationMessageFor(model =>model.address)
</div>
<div class="editor-label">
@Html.LabelFor(model =>model.email)
</div>
<div class="editor-field">
@Html.EditorFor(model =>model.email)
@Html.ValidationMessageFor(model =>model.email)
</div>
<p>
<input type="submit"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

Trong đoạn code trên,

- Phương thức ***BeginInitForm*** khởi động một form
- ***Html.ValidationSummary()*** hiển thị tất cả các thông báo lỗi
- ***Html.LabelFor()*** hiển thị một thành phần HTML label với tên của thuộc tính
- ***Html.EditorFor()*** hiển thị một textbox tiếp nhận giá trị của một thuộc tính model
- ***Html.ValidationMessageFor()*** hiển thị thông báo lỗi ở thuộc tính model tương ứng

# Create Template

- ◆ Kết quả của đoạn code:



The screenshot shows a web browser window with the address bar displaying `http://localhost:2222/` and the page title 'Create'. The main content area is titled 'Create' and contains a form for creating a new 'User'. The form has three input fields labeled 'name', 'address', and 'email', each followed by a text input box. Below these fields is a 'Create' button. At the bottom of the form, there is a link labeled 'Back to List'.

Khi người dùng nhập dữ liệu vào form và nhấn Create, một yêu cầu HTTP POST được gửi tới phương thức **Create()** của controller. Thêm vào đó, dữ liệu mới được nhập cũng sẽ được gửi cùng với yêu cầu.

# Edit Template

- ◆ Sử dụng Edit template khi bạn cần tạo 1 view được yêu cầu để sửa đổi chi tiết của một đối tượng có sẵn được lưu trong kho dữ liệu
- ◆ Để hiển thị 1 view dựa trên Edit template, ta cần tạo ra một phương thức hành động để truyền đối tượng model được sửa đổi tới view.
- ◆ Phương thức hành động ***Edit()*** :

## Code Snippet:

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

- ◆ Khi đã tạo được phương thức ***Edit()*** trong controller, sử dụng Visual Studio .NET để tạo ra view với Edit template. Để làm được điều đó, ta chọn mục ***Edit*** từ Scaffold templates.
- ◆ Sau khi tạo một view tên ***Edit*** cho model ***User*** với Edit template, Visual Studio .NET sẽ tạo ra markup của view



- ◆ Đoạn code dưới đây biểu thị output của việc tạo view bằng cách sử dụng Edit template:

## Code Snippet:

```
@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>
    </fieldset>
}
```

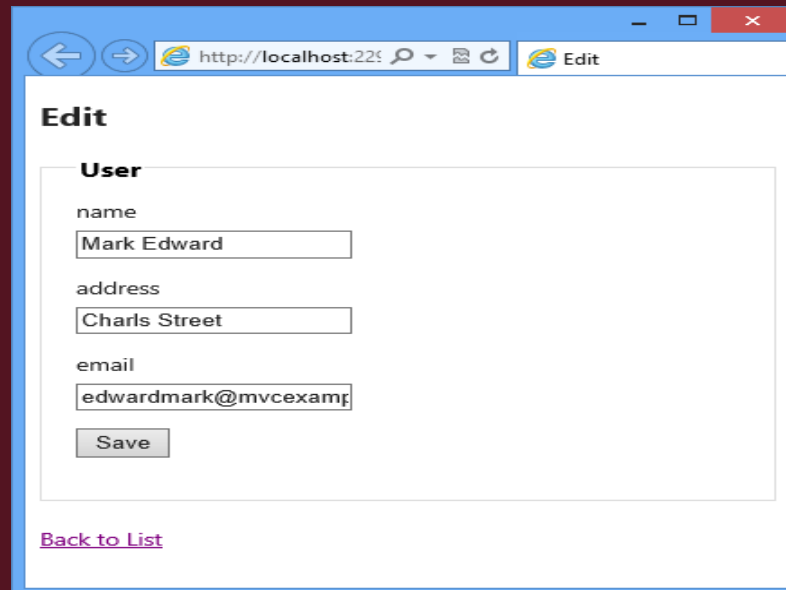
## Code Snippet:

```
<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")
}
```

Trong đoạn code trên,

- ***Html.LabelFor()*** hiển thị một thành phần HTML label với tên thuộc tính
- ***Html.EditorFor()*** hiển thị một textbox tiếp nhận giá trị thuộc tính
- ***Html.ValidationMessageFor()*** hiển thị một thông báo lỗi

◆ Kết quả của đoạn code trên:



The screenshot shows a web browser window with the address bar displaying 'http://localhost:225'. The browser has a single tab titled 'Edit'. The page content is titled 'Edit' and contains a form for editing a user. The form is titled 'User' and has three text input fields: 'name' with the value 'Mark Edward', 'address' with the value 'Charls Street', and 'email' with the value 'edwardmark@mvcexampl'. Below the input fields is a 'Save' button. At the bottom of the form, there is a link labeled 'Back to List'.

# Details Template

- ◆ Sử dụng Details template để tạo 1 view hiển thị chi tiết model **User**.
- ◆ Khi đã tạo được phương thức ***Details()*** trong controller và 1 view với Details template trong Visual Studio .NET, nó sẽ tạo ra markup cho view
- ◆ Đoạn code sau biểu diễn markup được tạo tự động khi ta tạo view bằng Details template:

## Code Snippet:

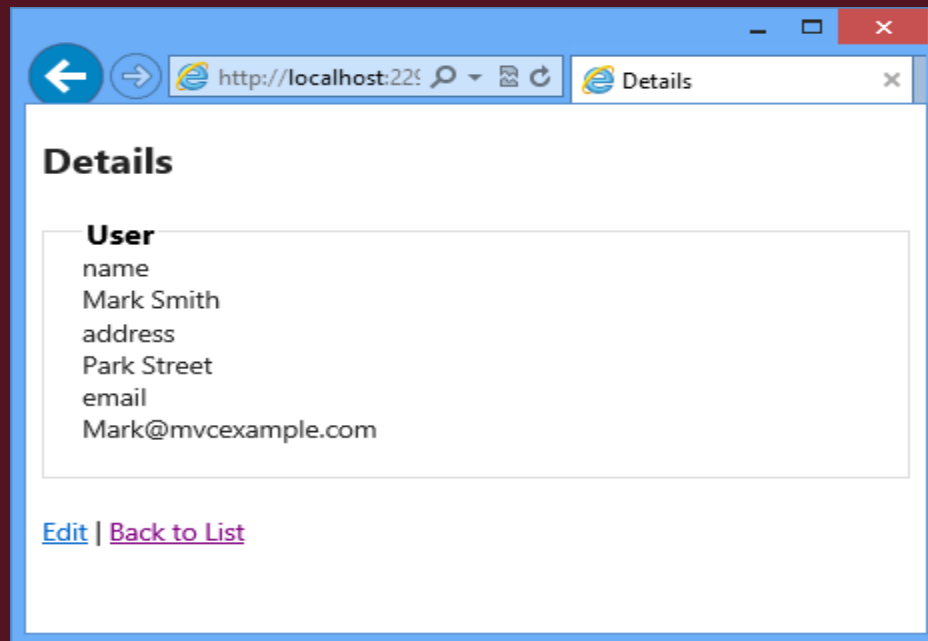
```
@model MVCModelDemo.Models.User
@{
    ViewBag.Title = "Details";
}
<h2>Details</h2>
<fieldset>
<legend>User</legend>
<div class="display-label">
    @Html.DisplayNameFor(model => model.name)
</div>
<div class="display-field">
    @Html.DisplayFor(model => model.name)
</div>
```

## Code Snippet:

```
<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>
```

Trong đoạn code trên, phương thức *Html.DisplayNameFor()* hiển thị tên các thuộc tính model và *Html.DisplayFor()* hiển thị giá trị các thuộc tính đó

- ◆ Kết quả:



## Delete Template

- ◆ Sử dụng Delete template để tạo 1 view cho phép người dùng xóa một đối tượng có sẵn trong kho dữ liệu.
- ◆ Để tạo view dựa trên Delete template, trước hết cần tạo một phương thức hành động truyền đối tượng model được xóa tới view
- ◆ Sau khi tạo phương thức *Delete ()* và 1 view cho model *User* sử dụng Delete template trong Visual Studio .NET, nó sẽ tạo ra một markup cho view

# Delete Template

- ◆ Đoạn code dưới đây biểu diễn markup được tạo tự động khi ta tạo một view bằng cách sử dụng Delete template :

## Code Snippet:

```
@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>
```



# Delete Template

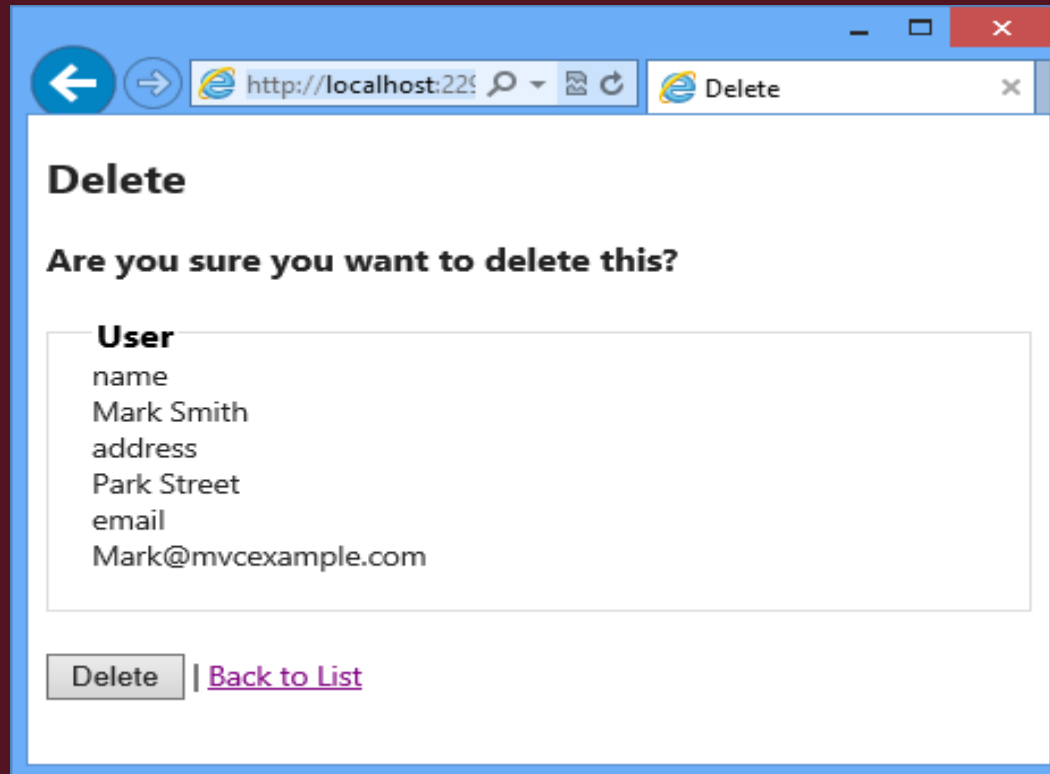
## Code Snippet:

```
<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>
}
```

Trong đoạn code trên, phương thức *Html.DisplayNameFor()* hiển thị tên các thuộc tính model và *Html.DisplayFor()* hiển thị giá trị các thuộc tính model.

# Delete Template

- ◆ Kết quả:



- Trong ứng dụng ASP.NET MVC, 1 model biểu diễn dữ liệu được liên kết với ứng dụng
- Trong mô hình MVC, có 3 loại model, mỗi loại có 1 chức năng khác nhau
- MVC Framework cung cấp các phương thức hỗ trợ được sử dụng trong các strongly-typed view
- Quy trình ánh xạ dữ liệu trong đối tượng HttpRequest tới 1 đối tượng model gọi là model binding
- ASP.NET MVC Framework cung cấp tính năng scaffolding cho phép tạo ra các views một cách tự động
- Visual Studio .NET đơn giản hóa quá trình tạo các view cho một phương thức hành động với các scaffolding template khác nhau.