





## UNIT 12: VIEWS





# CONTENTS

- Purpose of Views
- View conventions
- Strongly typed Views
- View Models
- Razor View Engine
- Layouts
- ViewStart



## Purpose of Views

- View is responsible for providing the user interface (UI) to the user.
- The view needs little or no information from the controller
- View displays data from the model to the user and also enables them to modify the data.

# View convention



- ASP.NET MVC views are stored in **Views** folder.
- Each controller folder contains a view file for each action method, and the file is named the same as the action method. This provides the basis for how views are associated to an action method.

- Render a different view:

- by pass a view name.

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

- Specify a view:

- provide full path to the view

```
public ActionResult Index()  
{  
    return View("~/Views/Example/Index.cshtml");  
}
```

# Passing data from Controller to View



- View is used for data presentation
- Controller must provide a view with the data
- One approach: using ViewBag
  - Controller puts data to ViewBag,
  - View reads ViewBag and renders the data
  - No data binding!
  - pass a little bit of data to the view via the ViewBagEx: `ViewBag.CurrentTime = DateTime.Now`
- Another approach: using ViewData
  - Ex: `ViewData["CurrentTime"] = DateTime.Now;`
- Alternative approach: the view model
  - Strongly typed view approach

# Passing data from Controller to View



- Strongly type view: Allow you to set a model type for a view. This allows you to pass a model object from the controller to the view that's strongly typed on both ends.
- Benefits of strongly type view:
  - Better compile-time checking
  - Richer IntelliSense in VS code editor

# Passing data from Controller to View



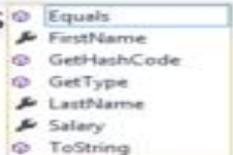
- Use strongly type view:

- Step1: Make View a strongly typed view

```
@model WebApplication1.Models.Employee
```

- Step2: Display data

```
<b>Employee Details</b><br />  
Employee Name : @Model.  
Employee Salary: @emp.S
```



- Step3: Pass Model data from Controller Action method

```
Employee emp = new Employee();  
emp.FirstName = "Sukesh";  
emp.LastName="Marla";  
emp.Salary = 20000;  
return View("MyView",emp);
```

- Step4: Test the output



# Razor view engine



- What is a Razor?
  - A new view-engine
  - Optimized around HTML generation
  - Code-focused templating approach
  - Make the transitions between code and markup as smoothly as possible
- Razor engine: Design goals:
  - Compact, Expressive and Fluid
  - Great Intellisense
  - Unit-testable: Testing views without server, controllers...
- Razor views: .cshtml, .vbhtml



# Razor view engine

- Use @ to translate from markup to code and sometimes also to transition back
  - Code Expression
  - Code Blocks

```
@{
    // this is a block of code. For demonstration purposes,
    // we'll create a "model" inline.
    var items = new string[] { "one", "two", "three" };
}
<html>
<head><title>Sample View</title></head>
<body>
    <h1>Listing @items.Length items.</h1>
    <ul>
        @foreach (var item in items) {
            <li>The item name is @item.</li>
        }
    </ul>
</body>
</html>
```



# Layouts

- Layouts in Razor help maintain a consistent look and feel across multiple views in your application.
- You can use a layout to define a common template for your site (or just part of it). This template contains one or more placeholders that the other views in your application provide content for
  - Use `@RenderBody()`
- This view specifies its layout via the Layout property.
- A layout may have multiple sections
  - `@RenderSection("Footer")`

# Layout



```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Site</title>
  </head>
  <body>

    <div id="header">
      <a href="/">Home</a>
      <a href="/About">About</a>
    </div>

    <div id="body">
      @RenderBody()
    </div>

  </body>
</html>
```

SiteLayout.cshtml

@RenderBody()  
Including specific body content.

# Layout



```
@{
    LayoutPage = "SiteLayout.cshtml";
}
```

Explicitly setting LayoutPage property.

```
<h1>About This Site</h1>
```

```
<p>
```

This is some content that will make up the "about" page of our web-site. We'll use this in conjunction with a layout template. The content you are seeing here comes from the Home.cshtml file.

```
</p>
```

```
<p>
```

And obviously I can have code in here too. Here is the current date/time: @DateTime.Now

```
</p>
```

Complete HTML page.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Site</title>
  </head>
  <body>

    <div id="header">
      <a href="/">Home</a>
      <a href="/About">About</a>
    </div>

    <div id="body">

      <h1>About This Site</h1>

      <p>
        This is some content that will make up the "about"
        page of our web-site. We'll use this in conjunction
        with a layout template. The content you are seeing here
        comes from the Home.cshtml file.
      </p>
      <p>
        And obviously I can have code in here too. Here is the
        current date/time: 7/2/2010 2:53:24 PM
      </p>

    </div>

  </body>
</html>
```

# Layout



```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Site</title>
  </head>
  <body>

    <div id="header">
      <a href="/">Home</a>
      <a href="/About">About</a>
    </div>

    <div id="left-menu">
      @RenderSection("menu", optional:true)
    </div>

    <div id="body">
      @RenderBody()
    </div>

    <div id="footer">
      @RenderSection("footer", optional:true)
    </div>

  </body>
</html>
```

This section is optional.

This section is optional.



# Layout



```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple Site</title>
  </head>
  <body>

    <div id="header">
      <a href="/">Home</a>
      <a href="/About">About</a>
    </div>

    <div id="left-menu">
      @RenderSection("menu", optional:true)
    </div>

    <div id="body">
      @RenderBody()
    </div>

    <div id="footer">
      @RenderSection("footer", optional:true)
    </div>

  </body>
</html>
```

```
<h1>About This Site</h1>
```

```
<p>
```

This is some content that will make up the "about" page of our web-site. We'll use this in conjunction with a layout template. The content you are seeing here comes from the Home.cshtml file.

```
</p>
```

```
<p>
```

And obviously I can have code in here too. Here is the current date/time: @DateTime.Now

```
</p>
```

```
@section menu {
```

```
  <ul id="sub-menu">
```

```
    <li>About Item 1</li>
```

```
    <li>About Item 2</li>
```

```
  </ul>
```

```
}
```

```
@section footer {
```

```
  <p>This is my custom footer for Home</p>
```

```
}
```

Named section.

Named section.



## ViewStart

- Each view specified its layout page using the Layout property. For a group of views that all use the same layout, this can get a bit redundant and harder to maintain.
- You can use the `_ViewStart.cshtml` page to remove this redundancy. The code within this file is executed before the code in any view placed in the same directory.



# ViewStart



- Specify a partial view:
  - an action method can also return a partial view in the form of a PartialViewResult via the PartialView method

```
public class HomeController : Controller {  
    public ActionResult Message() {  
        ViewBag.Message = "This is a partial view.";  
        return PartialView();  
    }  
}
```

- The partial view itself looks much like a normal view, except it doesn't specify a layout