# ASP.NET MVC

Views (Buch Kap. 3)

# Views

- Verantwortlich für:
  - Benutzerschnittstelle (UI)
- Nicht direkt zugreifbar, wie zum Beispiel php-Skripte.
- Immer vom Controller ausgeführt (gerendert).
  - Controller stellt Daten (model) bereit.
  - View bereitet die Daten (model) in ein benutzerfreundliches Format (HTML).
  - Views rendern nicht immer HTML, z.B. PDF-Datei, Jpeg-Bild…

# ViewBag

```
//HomeController.cs
public ActionResult Index()
{
   return View();
}
```

- Die Actionmethode Index() des HomeControllers rendert die *Home Index* View.

```
//HomeController.cs
public ActionResult Index()
{
   ViewBag.Title = "WO IST DER TITEL";
   return View();
}
```

# ViewBag

```
//HomeController.cs
public ActionResult About()
{
   ViewBag.Title = "WO IST DER TITEL";
   ViewBag.Message = "WO WIRD DIESER TEXT ANGEZEIGT??";
   return View();
}
```

```
//About.cshtml
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>

<p>wir lernen ASP.NET</p>
```

# View Conventions

- Controller ruft View() auf ohne einen Dateinamen anzugeben.
- Für jeden Controller ein View-Verzeichnis.
- Für jede action Methode eine View Datei.
  - Dateiname ist der gleiche wie der action Methodenname.
    - `About.cshtml, Index.cshtml, Contact.cshtml`
- Die Auswahllogik sucht die View mit demselben Namen wie die action Methode im Verzeichnis `/Views/ControllerName`
  - `ControllerName` ohne den Suffix Controller.
  - `/Views/Home/About.cshtml`

# Explizite View-Auswahl

```
//HomeController.cs
public ActionResult About()
{
   ViewBag.Title = "WO IST DER TITEL";
   ViewBag.Message = "WO WIRD DIESER TEXT ANGEZEIGT??";
   return View("Gugus");
}
```

- Sucht im `/Views/Home` Verzeichnis nach der view `Gugus.cshtml`.

- Mit der **Tilde-Syntax** muss der vollständige Pfad inklusive der Datei-Erweiterung (Endung) angegeben werden.

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

# ViewBag

- Einschränkungen
- Nützlich, um wenige Daten an View zu übergeben.
- Bei grösseren Daten z.B. folgende Nachteile:

```
//HomeController.cs
public ActionResult List()
{
   var albums = new List<Album>();
   for(int i = 0; i < 10; i++)
      albums.Add(new Album {Title = "Album " + i});

   ViewBag.Albums = albums;
   return View();
}
```

# ViewBag

```
//List.cshtml
<ul>
@foreach(Album a in (ViewBag.Albums as IEnumerable<Album>)) {
   <li>@a.Title</li>
}
</ul>
```

- `ViewBag.Albums` ist `dynamic` und muss zu `IEnumerable<Album>` Typumgewandelt werden (type cast).

```
//List.cshtml
<ul>
@foreach(dynamic a in ViewBag.Albums) {
   <li>@a.Title</li>
}
</ul>
```

- Verlust von IntelliSense bei Verwendung von `dynamic`.

# Strongly Typed Views

- Erlaubt die Übergabe eines Model-Objektes.
  - Compiler-Überprüfung
  - IntelliSense etc.

```
//HomeController.cs
public ActionResult List()
{
   var albums = new List<Album>();
   for(int i = 0; i < 10; i++)
      albums.Add(new Album {Title = "Album " + i});

   ViewBag.Albums = albums;
   return View(albums);
}
```

- Der verwendete Model-Typ muss in der View deklariert werden.
  - **@model**

# Strongly Typed Views

```
//List.cshtml

@model IEnumerable<MvcMusicStore.Models.Album>

<ul>
@foreach(Album a in Model) {
   <li>@a.Title</li>
}
</ul>
```

- Voll qualifizierter Typname (fully qualified type name)
  - `@using MvcMusicStore.Models`
  - `@model IEnumerable<Album>`
- Besser: Deklariere den namespace im `web.config` innerhalb des View-Verzeichnisses.

# ViewBag, ViewData und ViewDataDictionary

- Die Informationen in der ViewBag und das strongly typed model werden auf die gleiche Art und Weise an die View übergeben:
  - Über ein **`ViewDataDictionary`**
- Alle Daten vom Controller an die Views werden über die Property **`ViewData`** übergeben.
  - `ViewData` ist vom Typ `ViewDataDictionary`.
- Wo ist der Zusammenhang / Unterschied ?
  - ViewBag ist ein dynamischer **Wrapper** um ViewData.

```
ViewBag.CurrentTime = DateTime.Now
//ist äquivalent zu
ViewData["CurrentTime"] = DateTime.Now;
```

- Welche Syntax man verwendet, ist Geschmacksache.

# ViewBag != ViewData

- Einzig die Auswahl der Keys setzt grenzen, zum Beispiel ein Leerschlag im Schlüssel.
  - `ViewData["Key with Spaces"] = ….`
  - `ViewBag.Key with Spaces = …. //Compilerfehler!!!`

- Dynamische Werte können nicht als Parameter an Extension Methods übergeben werden.
  - Compiler muss den echten Typ von jedem Parameter kennen.

```
//erzeugt Compiler-Fehler
@Html.TextBox("name", ViewBag.Name);

//zwei Lösungen: 1) über ViewData oder 2) type cast
@Html.TextBox("name", ViewData["Name"]);
@Html.TextBox("name", (string)ViewBag.Name);
```

# Model Property

- ViewData, bzw. ViewDataDictionary hat eine zusätzliches Property Model.
  - Ermöglicht, bzw. stellt der View ein spezifisches Model-Objekt zur Verfügung.
  - ViewData kann nur ein Model-Objekt enthalten.

# View Models

- Zusätzliche Informationen
- Separate View Model-Klasse
  - IntelliSense, Compiler-Überprüfung, etc.
- TODO: Beispiel einfügen

# View hinzufügen

- Von "Hand" oder über den Dialog "Ansicht hinzufügen…"
  - Rechte Maustaste auf der Action Methode
- Temlates:
  - Create
  - Delete
  - Details
  - Edit
  - Empty
  - Empty (without model)
  - List
- Teilansicht (partial view)
- Layoutseite (layout page)
  - Layout reference
  - Fully self-contained view

# Razor Syntax

- Vereint/Verschmilzt C#-Code und HTML
- **@**-Zeichen
  - Das wichtigste Zeichen.
  - „key transition character“
  - Wechsel von HTML zu C#-Code und auch umgekehrt.
  - Es teilt der razor view engine mit, dass nachfolgende Zeichen **Code** und kein HTML ist.
- IntelliSense Unterstützung
- Dateierweiterung `.cshtml`
- Bei Audrücken (Expressions) muss das „Ende“ nicht explizit markiert werden.
  - `<h1>Listing **@items.Length** items.</h1>`
  - Razor Engine ist intelligent genug, um festzustellen, dass der Ausdruck beim Leerschlag aufhört.
  - Beim Leerschlag wechsel zurück zu HTML

# Razor Syntax

```
@{
    //Ein Block mit C#-Code
    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>
```

- Razor wechselt vom Code zu Markup beim Beginn/öffnendem Tag.

# Razor Syntax

```
@{
    string rootNamespace = "MyApp";
}
<span>@rootNamespace.Models</span>
```

- Erwartete Ausgabe: `<span>MyApp.Models</span>`
- Stattdessen erhalten wir einen **Fehler**: `Models` ist keine Property von string.
- Lösung: Explizites Einklammern von Ausdrücken, mit runden Klammern (..)

```
@{
    string rootNamespace = "MyApp";
}
<span>@(rootNamespace).Models</span>
```

# Razor Syntax

```
<span>perkovic@hfu.ch</span>
```

- Wir erwarten einen Fehler, dass die Property „ch“ der variable „hfu“ nicht existiert.
- Razor erkennt das Muster einer E-Mail Adresse.
  - Razor Engine erkennt E-Mail-Adressen und behandelt dies nicht als C#-Ausdruck.
  - Nicht darauf verlassen! Manche korrekten E-Mail-Adressen werden nicht erkannt, oder auch umgekehrt.

```
<li>Item_@item.Length</li>
//Wir erwarten <li>Item_3</li>, stattdessen wird es als E-Mail…
```

```
//Lösung einklammern (wrapping)
<li>Item_@(item.Length)</li>
```

- Escape-Zeichen **@@**

```
<span>perkovic@@hfu.ch</span>
```

# HTML Encoding

- Bei Benutzereingaben, bzw. der späteren Darstellung, besteht die Gefahr einer „**cross-site script injection**“ (**XSS**) Attacke.
- Razor Ausdrücke werden automatisch HTML encodiert.

```
@{
    string message = "<script>alert('hacked!');</script>";
}
<span>@message</span>
```

- Statt einem Alert-Dialog wird der HTML-Code encodiert zu:
  - `<span>&lt;script&gt;alert(&#39;haacked!&#39;);&lt;/script&gt;</span>`

# HTML Encoding

- Zwischendurch möchte man aber den Inhalt inklusive HTML-Tags ausgeben.
  - `System.Web.IHtmlString`
  - Direkt eine Instanz von `HtmlString` erstellen.
  - Praktischer: `Html.Raw()`

```
@{
    string message = "<strong>This is bold!</strong>";
}
<span>@Html.Raw(message)</span>
```

# JavaScript Encoding

```
<script type="text/javascript">
   $(function () {
      var message = 'Hello @ViewBag.Username';
      $("#message").html(message).show('slow');
   });
</script>
```

- Als Username könnte man zum Beispiel folgendes eingeben, was von der JavaScript engine evaluiert wird:
  - `\x3cscript\x3e%20alert(\x27pwnd\x27)%20\x3c/script\x3e`

```
<script type="text/javascript">
 $(function () {
   var message = 'Hello @Ajax.JavaScriptEncode(ViewBag.Username)';
   $("#message").html(message).show('slow');
 });
</script>
```

# Code Blocks

- Razor Engine unterstützt Code-Blöcke und wechselt automatisch zwischen HTML und C#-Code (in beide Richtungen).

```
@foreach(var item in items) {
   <li>The item name is @item.</li>
}
```

```
@foreach(var item in items) { <li>The item name is @item.</li> }
```

```
@{
   string s = "One line of code.";
   ViewBag.Title "Another line of code";
}
```

```
@{Html.RenderPartial("SomePartial");}
```

# Razor Syntax Beispiele

```
//Implizite Code Ausdrücke, werden immer HTML-encodiert!!
<span>@model.Message</span>
```

```
//Explizite Code Ausdrücke, HTML-encodiert!
<span>1 + 2 = @(1+2)</span>
```

```
//Uncodierte Ausdrücke, um sicherzustellen, dass der Wert nicht
//                      HTML-encodier wird
<span>@Html.Raw(model.Message)</span>
```

```
//Code-Blöcke
@{
    int x = 123;
    string y = "because.";
}
```

- Ausdrücke (code expression) werden evaluiert und zur Antwort (response) geschrieben.
- Code Blöcke werden hingegen nur ausgeführt. Z.B. Variablendeklaration…

# Razor Syntax Beispiele

```
//HTML, C#-Code und Text, automatischer wechsel zu Markup…
@foreach(var item in items) {
   <span>Item @item.Name.</span>
}
```

```
//Mischung Code und Klartext, automatischer Wechsel…
@if(showMessage) {
   <text>Das ist Klartext.</span>
}
//Ohne diese spez. Syntax wäre der Wechsel zu Markup nicht möglich
@if(showMessage) { @:Das ist Klartext.
}
```

- `<text>` ist ein spezieller Tag und wird selbst nicht in die Antwort geschrieben, nur sein Inhalt.
- **@:** ist eine spezielle Syntax für den Wechsel zurück zu Klartext, bzw. HTML.
  - Funktioniert nur für eine Zeile Text.

# Razor Syntax Beispiele

```
//Serverseiteige Auskommentierung von HTML und/oder Code
@*
Das ist ein mehrzeiliger serverseitiger Kommentar
@if (showMessage) {
   <h1>@ViewBag.Message</h1>
}
All of this is commented out.
*@
```

```
//Generische Methodenaufrufe müssen explizit eingeklammert werden.
//Die Eckigen Klammern veranlassen die Razor Engine zurück zu HTML
//zu wechseln.
@(Html.SomeMethod<AType>())
```

# Layouts

- Layout helfen durchgehend ein einheitliches Erscheinungsbild über mehrere Views oder die gesamte Webanwendung zu erhalten.

# @RenderBody()

```
//SiteLayout.cshtml
<!DOCTYPE html>
<html>
<head><title>@ViewBag.Title</title></head>
<body>
  <h1>@ViewBag.Title</h1>
  <div id="main-content">@RenderBody()</div>
</body>
</html>
```

- @RenderBody() ist ein Platzhalter, welcher die Stelle definiert, wo die Views (der Inhalt der Views) dargestellt werden.

```
//Index.cshtml
@{
   Layout = "~/Views/Shared/SiteLayout.cshtml";
   ViewBag.Title = "The Index!";
}
<p>This is the main content!</p>
```

# @RenderBody()

- Wenn die View Index gerendert wird, so wird ihr Inhalt in das Div-Element `main-content` vom `SiteLayout.cshtml` eingefügt.

```
//SiteLayout.cshtml
<!DOCTYPE html>
<html>
<head><title>The Index!</title></head>
<body>
  <h1>The Index!</h1>
  <div id="main-content"><p>This is the main content!</p></div>
</body>
</html>
```

# Sections

- Ein Layout kann mehrere Sektionen enthalten, bzw. in mehrere Sektionen unterteilt werden.
- Zum Beispiel: eine „footer“-Sektion

```
//SiteLayout.cshtml
<!DOCTYPE html>
<html>
<head><title>@ViewBag.Title</title></head>
<body>
  <h1>@ViewBag.Title</h1>
  <div id="main-content">@RenderBody()</div>
  <footer>@RenderSection("Footer")</footer>
</body>
</html>
```

- Falls die Sektion „Footer“ nicht definiert wurde, wird eine Exception geworfen. Die View muss jede definierte Sektion zur Verfügung stellen.

# Sections

- Die Angepasste View mit der „Footer“-Sektion:

```
//Index.cshtml
@{
   Layout = "~/Views/Shared/SiteLayout.cshtml";
   ViewBag.Title = "The Index!";
}
<p>This is the main content!</p>

@section Footer {
    This is the <strong>footer</strong>.
}
```

# Optionale Sections

- Im Layout können die Sektionen als **optional** definiert werden.

```
//SiteLayout.cshtml
...
 <footer>@RenderSection("Footer", required: false)</footer>
...
```

- Default-Inhalt, falls eine Sektion in der View nicht definiert wird.

```
//SiteLayout.cshtml
...
<footer>
   @if (IsSectionDefined("Footer")) {
      RenderSection("Footer");
   }
   else {
      <span>This is the default footer.</span>
   }
</footer>
```

# ViewStart

- Falls mehrere Views dasselbe Layout verwenden, so kann dies in der Datei `_ViewStart.cshtml` definiert werden.
- Pro Verzeichnis kann eine Datei `_ViewStart.cshtml` definiert werden. Der Code in dieser Datei wird vor allen anderen Views im denselben Verzeichnis ausgeführt.
- Wird rekursiv auf alle Views in den Unterverzeichnissen angewendet.
- Gemeinsame Einstellungen können so in `_ViewStart.cshtml` definiert werden und in jeder View überschrieben…

# Teilansicht - Partial View

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

- Die View `Message.cshtml` wird gerendert.
- Das definierte Layout in `_ViewStart.cshtml` wird **nicht** gerendert.
- Sehr nützlich, wenn zum Beispiel per Ajax nur gewisse Teile neu geladen werden sollen, siehe Beispiel nächste Folie.

# Teilansicht - Partial View

```
//Message.cshtml
<h2>@ViewBag.Message</h2>
```

```
<div id="result"></div>

@section scripts {
<script type="text/javascript">
   $(function(){
      $('#result').load('/home/message');
   });
</script>
}
```