05. Forms and HTML Helpers

HTML Formulare

HTML Helper

- Wozu benötigen wird sogenannte "HTML Helfer"?
  - URLs in Links sind gültig und verweisen immer an den richtigen Ort.
  - Form Elemente haben korrekte Namen und Werte. (Model Binding)
  - Zugehörige Validierungsfehler

HTML Forms

- Sehr mächtig! Ohne das HTML Tag Form wäre eine Benutzerinteraktion nicht möglich.
- ASP.NET Web Forms, Vorgänger von MVC/Razor, nutzt nicht alle Möglichkeiten des `form` Tags.
- Form ist eine Art Container für Eingabeelemente, Buttons, Checkboxes, …
- `action`-Attribut
  - Teilt dem Browser mit, wohin die Informationen zu senden sind.
  - Relativ URL
  - Absolute URL, falls die Informationen zu einem fremden Server gesendet…
- `method`-Attribut
  - HTTP GET, ist der default
    - Der Browser setzt alle Parameternamen mit Werten in den Query String.
    - Die HTTP Anfrage kann als Hyperlink in der E-Mail versendet oder „gebookmarked“
    - In der Regel wird der Zustand der Daten auf dem Server nicht geändert.

HTML Forms

- `method`-Attribut
  - HTTP POST
    - Der Browser setzt alle Parameter mit Werten in den Body der HTTP-Anfrage.
    - Konvention: eine POST Anfrage ändert den Zustand der Daten auf dem Server.
    - VORTEIL: Falls die Seite im Browser neu geladen wird, so fragt der Browser nach, ob die Anfrage/Daten erneut gesendet werden sollen.
- GET oder POST?
  - HTTP GET ist vorzuziehen.
    - Bookmark (die komplette Anfrage)
    - Anfrage kann als URL in Links benutzt werden und zum Beispiel per E-Mail…
- Konvention:
  - GET fürs Lesen.
  - POST fürs Schreiben/Updates.

Such-Formular

```
//Ein Suchformular
<form action="/Home/Search" method="get">
  <input type="text" name="q" />
 <input type="submit" value="Search" />
</form>
```

```
//HomeController.cs
public ActionResult Search(string q)
{
   var albums = storeDB.Albums
                       .Include("Artist")
                       .Where(a => a.Title.Contains(q))
                       .Take(10);
   return View(albums);
}
```

- Beachte das hard-codierte `action`-Attribut.
  - Falls sich z.B. die Route Definitionen ändern…

Such-Formular

```
//Search.cshtml
@model IEnumerable<MvcMusicStore.Models.Album>
@{ ViewBag.Title = "Search"; }

<h2>Results</h2>
<table class="table table-condensed table-striped">
  <tr>
    <th>Artist</th>
    <th>Title</th>
    <th>Price</th>
  </tr>
  @foreach (var item in Model) {
    <tr>
      <td>@item.Artist.Name</td>
      <td>@item.Title</td>
      <td>@String.Format("{0:c}", item.Price)</td>
    </tr>
  }
</table>
```

Action-Attribute automatisch setzen

```
//Ein Suchformular
<form action="/Home/Search" method="get">
  <input type="text" name="q" />
 <input type="submit" value="Search" />
</form>
```

- Die fest codierte URL im `action`-Attribut sollte vermieden werden.
  - Z.B. falls sich die Route-Definitionen ändern oder die Webanwendung nicht im Root-Verzeichnis bereitgestellt wird, könnte der Browser die Anfrage an eine nicht existierende Ressource leiten.
- Besserer Ansatz:

```
//Ein Suchformular
@using (Html.BeginForm("Search", "Home", FormMethod.Get)) {
   <input type="text" name="q" />
   <input type="submit" value="Search" />
}
```

BeginForm HTML Helper

- Der `BeginForm` Helfer fragt die Routing Engine, wie die `Search` Action (Methode) des `HomeController` zu erreichen ist.
- Im Hintergrund passiert folgendes:
  - `BeginForm` benutzt die Funktion `GetVirtualPath`, um die (relative) URL zu erhalten.
- Ohne `BeginForm()` müsste der folgende Code geschrieben werden:

```
//Ein Suchformular
@{
   var context = this.ViewContext.RequestContext;
   var values = new RouteValueDictionary{ {"controller", "home" },
                                          { "action", "index" } };
   var path = RouteTable.Routes.GetVirtualPath(context, values);
}
<form action="@path.VirtualPath" method="get">
   <input type="text" name="q" />
   <input type="submit" value="Search2" />
</form>
```

HTML Helpers

- Die View bietet nützliche Properties an:
  - `Html`
  - `Url` (auch im Controller verfügbar)
  - `Ajax`
- Die **HTML Helpers**, Funktionen der Properties `Html`, `Url` und `Ajax`, helfen uns die View auf eine möglichst einfache und kurze Art zu erstellen.
- Die Meisten HTML Helfer generieren HTML Markup als Output.
- Mit Benutzung von `using` wird das schliessende Tag automatisch gesetzt.
  - Z.B. `<form> ... </form>`
  - Der Helfer `BeginForm` muss `IDisposable` implementieren.
  - `using` ruft beim erreichen der schliessenden geschweiften Klammer implizit die Funktion `Dispose` auf. Als Aufräumarbeit wird hier das schliessende Tag gerendert.

HTML Helpers

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get)) {
   <input type="text" name="q" />
   <input type="submit" value="Search" />
}
```

- Das Benutzen von `using` ist nicht vorgeschrieben.
- Das schliessende Tag kann auch explizit gesetzt werden.
  - Diese Art ähnelt an die Symmetrie in HTML:

```
//Explizites rendern des form-Endtags
@{Html.BeginForm("Search", "Home", FormMethod.Get);}
   <input type="text" name="q" />
   <input type="submit" value="Search" />
@{Html.EndForm();}
```

Automatisches encodieren

- Der Output von allen Helpern ist HTML-kodiert.
- Zum Beispiel:

```
@Html.TextArea("text", "Hallo <br /> Welt!!")
```

- Folgendes Markup wird generiert:

```
<textarea cols="20" id="text" name="text" rows="2">
   hello &lt;br /&gt; world
</textarea>
```

- Automatisches Encoding verhindert Cross-Site-Scripting Attacken.

HTML-Attribute

```
@using Html.BeginForm("Search", "Home", FormMethod.Get,
                                        new { target = "_blank" }))
{
   <input type="text" name="q" />
   <input type="submit" value="Search" />
}
```

```
//folgender öffnender form-Tag wird generiert
<form action="/Home/Search" method="get" target="_blank">
```

- Fast alle HTML Helper haben überladene Methoden mit einem `htmlAttributes` Parameter
  - anonymously typed object
  - `IDictionary<string, object>`
- Problematische Attribute, z.B. `class`
  - `class` ist ein Schlüsselwort in C#
  - Lösung: Das @-Zeichen allen problematischen Bezeichnern voranstellen.

HTML-Attribute

```
@using(Html.BeginForm("Search", "Home", FormMethod.Get,
                      new {target = "_blank", @class="editForm"}))
```

- Alle Attribute mit einem **Bindestrich**, z.B. `data-val` sind problematisch
  - C# erlaubt keine Bindestriche in Eigenschaftsnamen.
- Die HTML Helper wandeln beim Rendern alle **Untestriche in Bindestriche** um:

```
@using(Html.BeginForm("Search", "Home", FormMethod.Get,
                       new { target = "_blank", @class="editForm",
                             data_validatable=true }))
```

- Generiert folgendes HTML:

```
<form action="/Home/Search" class="editForm"
    data-validatable="true" method="get" target="_blank">
```

Inside HTML Helpers

- TODO:
- System.Web.Mvc.HtmlHelper<T>
- EnableClientValidation
- Extension Methods
- …

Edit Formular

```
@using(Html.BeginForm()) {
   @Html.ValidationSummary(excludePropertyErrors: true)
   <fieldset>
      <legend>Edit Album</legend>
      <p>
         <input type="submit" value="Save" />
      </p>
   </fieldset>
}
```

- Ein möglicher Anfang, um eine View zu erstellen für das Editieren eines Albums.
- `Html.BeginForm()` ohne Parameter sendet eine HTTP POST an die aktuelle URL.
- Z.B. falls die View eine Antwort ist auf `/StoreManager/Edit/52`, so wird folgender öffnender Tag generiert:

```
<form action="/StoreManager/Edit/52" method="post">
```

Html.ValidationSummary

- `Html.ValidationSummary` generiert eine unsortierte Liste von Validierungsfehlern im `ModelState` Dictionary.
- Mit dem Parameter `excludePropertyErrors` steuern wir das anzeigen von Property-Level Errors, bzw. ob spezifische Modell-Eigenschaften-Fehler angezeigt werden sollen oder nicht.

```
ModelState.AddModelError("", "This is all wrong!");
ModelState.AddModelError("Title", "What a terrible name!");
```

  - Erster Fehler ist ein **Model-Level Error**.
  - Zweier Fehler ist ein **Property-Level Error**, da spezifisch für die Eigenschaft `Title` gilt.

```
<div class="validation-summary-errors">
   <ul>
      <li>This is all wrong!</li>
   </ul>
</div>
```

Inputs - Formular

- Siehe Listing Buch S. 118 – 120.
- HTML Helpers:
  - TextBox
  - TextArea
  - Label, LabelFor
  - DropDownList und ListBox
  - ValidationMessageFor
  - ValidationSummary
  - HiddenFor

Html.TextBox()

- `TextBox()` rendert ein HTML `input` Tag mit dem Attribut `type="text"`.
- Zum Beispiel, angenommen Model.Title = "Midnight Oil"

```
@Html.TextBox("Title", Model.Title)
```

- generiert:

```
<input id="Title" name="Title" type="text" value="Midnight Oil" />
```

- `TextBox()` Helper stellt Überladungen zur Verfügung, um individuelle HTML Attribute zu setzen.

Html.TextArea()

- `TextArea()` rendert ein HTML `textarea` Tag für mehrzeiligen Text.
- Zum Beispiel:

```
@Html.TextArea("text", "hello <br/> world")
```

- Generiert:

```
<textarea cols="20" id="text" name="text" rows="2">
  hello &lt;br /&gt; world
</textarea>
```

- Wie bereits erwähnt, alle HTML Helper encodieren automatisch alle Modell-Werte und Attribute.
- Grösse, bzw. Anzahl Spalten und Zeilen können angegeben werden:

```
@Html.TextArea("text", "hello <br /> world", 10, 80, null)
```

```
<textarea id="text" name="text" cols="80" rows="10">
  hello &lt;br /&gt; world
</textarea>
```

Html.Label()

- `Label()` rendert ein HTML `label` Tag.
  - Mit dem `string` Parameter wird der Text und das `for` Attribut bestimmt.
- Zum Beispiel:

```
@Html.Label("GenreId")
```

- generiert:

```
<label for="GenreId">Genre</label>
```

- Der Zweck von Labels ist Informationen an Input Elemente anzufügen.
- Das `for` Attribut enthält in der Regel die ID des verknüpften Input Elements.
- Labels vergrössern das Gebiet für Mausklicks. (Nützlich bei Checkboxes…)
- Helpers benutzen alle zur Verfügung stehende Modell-Metadaten, um die Anzeige zu generieren.
  - Übergeben wurde "`GenreID`", aber es wird die Bezeichnung "`Genre`" angezeigt.

Html.DropDownList() und Html.ListBox()

- `DropDownList()` und ListBox(), beide rendern ein HTML `select` Tag.
  - DropDownList erlaubt nur die Auswahl eines Elements.
  - ListBox erlaubt die Auswahl mehrerer Elemente.
    - Indem `multiple="multiple"` gesetzt wird.
- Daten/Elemente werden in der Regel im Controller aufbereitet.
- Eine Liste benötigt eine Collection von **`SelectListItem`** Instanzen.
  - `SelectListItem` Objekt hat die Eigenschaften `Text`, `Value` und `Selected`.
- Statt eigene Collections von SelectListItems aufzubauen, können die Hilfsklassen **`SelectList`** oder **`MultiSelectList`** verwendet werden.
  - Diese Hilfsklassen wandeln eine `IEnumerable`-Sequenz in eine Sequenz von `SelectListItem` Objekten um.
- Zum Beispiel:

Html.DropDownList() und Html.ListBox()

```
//StoreManagerController.cs
public ActionResult Edit(int id)
{
   var album = storeDB.Albums.Single(a => a.AlbumId == id);
   var genres = storeDB.Genres.OrderBy(g => g.Name);

   ViewBag.Genres =
         new SelectList(genres, "GenreId", "Name", album.GenreId);

   return View(album);
}
```

- Parameters des `SelectList`-Konstruktorrs:
  - 1. Parameter: eine Collection, z.B. `Genres` aus der Datenbank.
  - 2. Parameter: Eigenschaft, welche als value (`GenreId`) benutzt wird.
  - 3. Parameter: Eigenschaft, welche als Text (`Name`) benutzt wird.
  - 4. Parameter: Wert, welcher ausgewählt ist.
- Alternative siehe Buch S. 123, (Performance, Reflection Mehraufwand)

Html.ValidationMessage()

- Falls im `ModelState` Dictionary für ein bestimmtes Feld ein Fehler vorhanden ist, kann dieser mit dem `ValidationMessage` Helper angezeigt werden.

```
//explizit einen Fehler hinzufügen.
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
   var album = storeDB.Albums.Find(id);
   ModelState.AddModelError("Title", "What a terrible name!");
   return View(album);
}
```

```
//Die Fehlernachricht kann in der View wie folgt angezeigt werden:
@Html.ValidationMessage("Title")
```

```
<span class="field-validation-error" data-valmsg-for="Title"
                                     data-valmsg-replace="true">
  What a terrible name!
</span>
```

Html.ValidationMessage()

- Die Fehlernachricht wird nur angezeigt, falls im Model State ein Fehler existiert für den Schlüssel `Title`.
- Falls ein Fehler existiert, so kann die Nachricht überschrieben werden.

```
//ersetzen der Fehlernachricht
@Html.ValidationMessage("Title",
                        "Something is wrong with your title")
```

```
//folgender HTML Code wird generiert:
<span class="field-validation-error" data-valmsg-for="Title"
                                     data-valmsg-replace="false">
   Something is wrong with your title
</span>
```

Helpers, Models, and ViewData

- Alle Helper überprüfen das `ViewData` Objekt um die anzuzeigenden Werte zu erhalten.
- Beispiel: Der Preis eines Albums könnte wie folgt gesetzt werden:

```
public ActionResult Edit(int id)
{
   ViewBag.Price = 10.0;
   return View();
}
```

- Um den Preis anzuzeigen, kann in der View eine `TextBox` gerendert werden. Der `TextBox` muss der gleiche Name (Property/Key) wie der des Wertes in der `ViewBag` übergeben werden.

```
@Html.TextBox("Price")
```

- Der `TextBox` Helper generiert folgendes HTML:

```
<input id="Price" name="Price" type="text" value="10.0" />
```

Helpers, Models, and ViewData

- Wenn die Helper in `ViewData` nach Objekten suchen, so können Sie auch deren Properties sehen.
- Wir ändern die vorherige Action Methode:

```
public ActionResult Edit(int id)
{
   ViewBag.Album = new Album {Price = 11};
   return View();
}
```

- Der Preis wird wie folgt mit dem `TextBox` Helper angezeigt:

```
@Html.TextBox("Album.Price")
```

- ABER, der `TextBox` Helper generiert ein leicht anderes HTML:

```
<input id="Album_Price" name="Album.Price" type="text" value="11"/>
```

- Beachte: das Attribut `id` benutzt einen **Unterstrich** statt einen **Punkt**.
- `HtmlHelper.IdAttributeDotReplacement`
- Ohne gültige `id` Attribute können client-seitige JavaScript Bibliotheken nicht ausgeführt werden.

Strongly Typed ViewData

```
public ActionResult Edit(int id)
{
   var album = new Album {Price = 12.0m};
   return View(album);
}
```

- Jetzt muss nur noch der Name der Property dem Helper angegeben werden:

```
@Html.TextBox("Price")
```

- Folgendes HTML wird generiert:

```
<input id="Price" name="Price" type="text" value="12.0"/>
```

Explizite Werte

```
public ActionResult Edit(int id)
{
   var album = new Album {Title = "Midnight Oil"};
   return View(album);
}
```

```
//die dazugehörige strongly typed view:
@{
   ViewBag.Title = "Edit - " + Model.Title;
}
@Html.TextBox("Title")
```

- Welcher Title wird ausgeben?
  - `"Edit – Midnight Oil"` oder `"Midnight Oil"` ??
- Um das automatische "data lookup" in ViewData zu vermeiden, können explizite Werte angegeben werden (zweiter Parameter).

```
@Html.TextBox("Title", Model.Title)
```

Helpers, Explizite Werte, ViewData

- Die Helpers suchen zuerst innerhalb von `ViewData`, und erst danach überprüfen sie das strongly typed model.
  - Das Bedeutet, falls der `TextBox` Helper nur mit dem einen Parameter "`Title`" aufgerufen wird, sucht er zuerst in `ViewData` und findet dort den Titel, welcher in der View gesetzt wurde.
- Um Verwechslungen zu vermeiden, können Präfixe verwendet werden.
  - Z.B. statt `ViewBag.Title` könnte `ViewBag.Page_Title` verwendet werden.

Strongly Typed Helpers

- Den Strongly Typed Helpers wird ein **Lambda Ausdruck** als Parameter übergeben, um die Model Eigenschaft fürs Rendering zu definieren.
- Der Modell-Typ des Lambda Ausdrucks ist der gleiche wie der Typ des View Modells (**@model**…).
- Strongly Typed Helpers haben dieselben Namen wie die bisherigen Helpers, jedoch mit dem Suffix **For**.
  - `Label – LabelFor`
  - `DropDownList - DropDownListFor`
- Strongly Typed Helpers generieren denselben HTML Code.
- Strings durch Lambda Ausdrücke zu ersetzen, bieten zahlreiche Vorteile:
  - IntelliSense, Compile-Time Error Checking, Refactoring

```
//Search.cshtml
@model MvcMusicStore.Models.Album
...
```

Strongly Typed Helpers

```
@using (Html.BeginForm())
{
   @Html.ValidationSummary(excludePropertyErrors: true)
   <fieldset>
      <legend>Edit Album</legend>
      <p>
         @Html.LabelFor(m => m.GenreId)
         @Html.DropDownListFor(m => m.GenreId,
                               ViewBag.Genres as SelectList)
      </p>
      <p>
         @Html.TextBoxFor(m => m.Title)
         @Html.ValidationMessageFor(m => m.Title)
      </p>
      <input type="submit" value="Save" />
   </fieldset>
}
```

Helpers und Modell Metadata

- Helpers suchen nicht nur in `ViewData`, sondern berücksichtigen auch verfügbare **Modell Metadata**.
- Zum Beispiel:

```
@Html.Label("GenreId")
```

- Generiert:

```
<label for="GenreId">Genre</label>
```

- Warum setzt der Helper als Text "`Genre`" und nicht "`GenreId`"?
- Der Helper fragt nach, ob irgendwelche Modell Metadaten für `GenreId` verfügbar sind.
  - Das Laufzeitsystem stellt die Informationen vom Attribut `DisplayName` zur Verfügung.

```
[DisplayName("Genre")]
public int GenreId { get; set; }
```

- Mehr dazu im Kapitel 6 des Buches.

Templated Helpers

- `Html.Display` und `Html.DisplayFor`
- `Html.Editor` und `Html.EditorFor`
- Und das Gegenstück für das Gesamtmodell:
  - `Html.DisplayForModel`
  - `Html.EditorForModel`
- Die Templated Helpers generieren HTML mit Hilfe von Metadaten und einem Template.
- Beispiel, der `Html.TextBoxFor` Helper generiert folgendes HTML für einen Albumtitel:

```
<input id="Title" name="Title" type="text" value="Midnight Oil" />
```

- Folgender Helper generiert den gleichen HTML Code:

```
@Html.EditorFor(m => m.Title)
```

- Was ist der Vorteil von `EditorFor` gegenüber `TextBoxFor`?
  - Über Data Annotationen kann die Generierung des HTML beeinflusst, gesteuert und geändert werden.

Templated Helpers

- Falls wir die Eigenschaft Title des Modells mit Data Annotationen dekorieren, zum Beispiel:

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
[DataType(DataType.MultilineText)]
public string Title { get; set; }
```

- So generiert `EditorFor` nun folgendes HTML:

```
<textarea class="text-box multi-line" id="Title" name="Title">
   Midnight Oil
</textarea>
```

- Das UI wird beeinflusst, ohne die View zu ändern.
- Custom templates, siehe Kapitel 15 im Buch.

Helpers und ModelState

- Alle Helper, welche Formular-Werte anzeigen, arbeiten mit `ModelState`.
- Zur Erinnerung: ModelState enthält alle Validierungsfehler.
  - ModelState enthält auch die Ur-Werte der gesendeten Formular-Daten.
- Beim Rendern suchen die Helper **zuerst im `ModelState`**, ob der Wert, bzw. der Schlüssel, vorhanden ist, erst danach in `ViewData`.
  - Warum macht das Sinn?

Html.Hidden

- `Hidden()` rendert ein HTML `input` Tag mit dem Attribut `type="hidden"`.
  - `HiddenFor()` ist der Strongly Typed.
- Zum Beispiel:

```
@Html.Hidden("wizardStep", "1")
@Html.HiddenFor(m => m.WizardStep) //Strongly Typed
```

- Beide generieren:

```
<input id="wizardStep" name="wizardStep" type="hidden" value="1" />
```

Html.Password

- `Password()` rendert ein HTML `input` Tag mit dem Attribut `type="password"`.
  - `PasswordFor()` ist der Strongly Typed.
- Zum Beispiel:

```
@Html.Password("UserPassword")
@Html.PasswordFor(m => m.UserPassword) //Strongly Typed
```

- Beide generieren:

```
<input id="UserPassword" name="UserPassword" type="password"
       value="" />
```

Html.RadioButton

- `RadioButton()` rendert ein HTML `input` Tag mit dem Attribut `type="radio"`.
  - `RadioButtonFor()` ist der Strongly Typed.
- Radio Buttons sind normalerweise gruppiert.
  - Jeder Radio Button hat den gleichen Namen, aber unterschiedliche Werte.
  - Nur der ausgewählte sendet seinen Wert zurück an den Server beim Submit.
- Zum Beispiel:

```
@Html.RadioButton("color", "red") //der Wert muss übergeben werden.
@Html.RadioButton("color", "blue", true)
@Html.RadioButton("color", "green")
```

```
//generiertes HTML:
<input id="color" name="color" type="radio" value="red" />
<input checked="checked" id="color" name="color" type="radio"
       value="blue" />
<input id="color" name="color" type="radio" value="green" />
```

Html.CheckBox

- `CheckBox()` rendert **zwei** HTML `input` Tags.
  - Ein input Tag mit dem Attribut `type="checkbox"`.
  - Ein input Tag mit dem Attribut `type="hidden"`.
- `CheckBoxFor()` ist der Strongly Typed.
- Der Hidden Input ist notwendig, weil der Browser nur gesetzte (ausgewählte) CheckBoxes an den Server zurückschickt. Damit auch nicht ausgewählte mit dem Wert false zurückgeschickt werden, braucht es ein zusätzliches Hidden Input Feld.
- Zum Beispiel:

```
@Html.CheckBox("IsDiscounted")
```

- Generiert:

```
<input id="IsDiscounted" name="IsDiscounted" type="checkbox"
       value="true" />
<input name="IsDiscounted" type="hidden" value="false" />
```

Html.ActionLink

- `ActionLink()` rendert ein HTML `href` Tag.
  - Der Helper rendert einen Hyperlink zu einer anderen Controller Action Methode.
  - Im Hintergrund wird fürs Rendering die Routing API benutzt.
- Zum Beispiel, falls zu einer anderen Action Methoden im selben Controller verlinkt werden soll, so reicht der Action Name aus:

```
@Html.ActionLink("Link Text", "AnotherAction")
```

- Folgendes HTML wird generiert:

```
<a href="/Home/AnotherAction">LinkText</a>
```

- Falls ein Hyperlink auf eine Action eines fremden Controllers benötigt wird, kann der Controller als dritter Parameter angegeben werden:

```
@Html.ActionLink("Link Text", "Index", "ShoppingCart")
```

Html.ActionLink

- Oft werden applikations-spezifische Routeparameter im Hyperlink benötigt, wie zum Beispiel die ID…
  - anonymously typed object
  - `RouteValueDictionary`
- Zum Beispiel: Einen Link generieren für das Bearbeiten (Edit) eines Albums mit der ID 10720:

```
@Html.ActionLink("Edit link text", "Edit", "StoreManager",
                 new {id=10720}, null)
```

- Der letzte Parameter ist das `htmlAttributes` Argument.
  - Falls nicht benötigt, `null` übergeben.

Html.RouteLink

- Der Helper `RouteLink()` folgt genau dem gleichen Schema wie der Helper `ActionLink()`
  - Akzeptiert einen Routenamen.
  - Hat keine Argumente für einen Controller- und Action-Namen.
- Folgende zwei Helper generieren einen äquivalenten Hyperlink.

```
@Html.ActionLink("Link Text", "AnotherAction")

@Html.RouteLink("Link Text", new {action="AnotherAction"})
```

URL Helpers: Action, Content und RouteUrl

- Die URL Helper sind ähnlich zu den `ActionLink` und `RouteLink` Helpern.
  - Generieren keine HTML Tags, sondern geben URLs als String zurück.
- `Url.Action()` ist genau gleich wie `Html.ActionLink()`, gibt aber einen String zurück.

```
@Url.Action("Browse", "Store", new { genre = "Jazz" }, null)
```

- Gibt eifach den Link als String zurück:

```
/Store/Browse?genre=Jazz
```

- Der `Url.RouteUrl()` Helper folgt dem gleichen Schema wie `Action()`
  - Akzeptiert einen Routenamen.
  - Hat keine Argumente für Controller- und Actionnamen.

URL.Content

- Der `Url.Content()` Helper kann einen relativen in einen absoluten Applikations-Pfad umwandeln.
- TODO:

Html.Partial und Html.RenderPartial

- Der `Partial()` Helper generiert eine Teilansicht (**partial view**) als String.
  - Wiederverwendbarer HTML Code, z.B. ein Menu
- `RenderPartial()` ist ähnlich wie `Partial()`, schreibt aber den generierten HTML Code direkt in den Response Stream.
- Folgende zwei Zeilen Code generieren denselben HTML Code im Output Stream:

```
@{Html.RenderPartial("AlbumDisplay "); }
@Html.Partial("AlbumDisplay ")
```

- Welchen Helper soll man verwenden?
  - Persönlicher Programmierstil…
  - `Partial` benötigt weniger Schreibaufwand.
  - `Renderpartial` könnte bei vielen Aufrufen eine bessere Performance liefern, da der Helper direkt in den Response Stream schreibt.

Html.Action und Html.RenderAction

- `Action()` ruft und führt eine andere Action-Methode aus.
  - Mehr Flexibilität, Wiederverwendbarkeit.
  - Erstellen eines anderen Modells.
  - Separater Controller Context.
- `Action()` gibt den generierten HTML Code als String zurück.
- `RenderAction()` schreibt direkt in den Response Stream.
- Beispiel:

```
//MyController.cs
public class MyController : Controller {

   public ActionResult Index() {
      return View();
   }

   //...
}
```

Html.Action und Html.RenderAction

```
//Index.cshtml
<html>
<head><title>Index with Menu</title></head>
<body>
    @Html.Action("Menu")
    <h1>Welcome to the Index View</h1>
</body>
</html>
```

```
//MyController.cs
public class MyController : Controller {
   //...

   [ChildActionOnly]
   public ActionResult Menu() {
      var menu = GetMenuFromSomewhere();
      return PartialView(menu);
   }
}
```

Html.Action und Html.RenderAction

```
//Menu.cshtml
@model Menu
<ul>
   @foreach (var item in Model.MenuItem) {
   <li>@item.Text</li>
   }
</ul>
```

- `ChildActionOnlyAttribute`
  - Verhindert, dass zur Laufzeit die Action Methode direkt über eine URL aufgerufen wird.
  - Nur ein `Action` oder `RenderAction` kann ein Child Action aufrufen.
- `IsChildAction` Eigenschaft des `ControllerContext` (seit MVC3).
  - Ist auf `true` gesetzt, falls die Methode via `Action` oder `RenderAction` aufgerufen wurde, sonst `false` (via URL).