

.NET Training

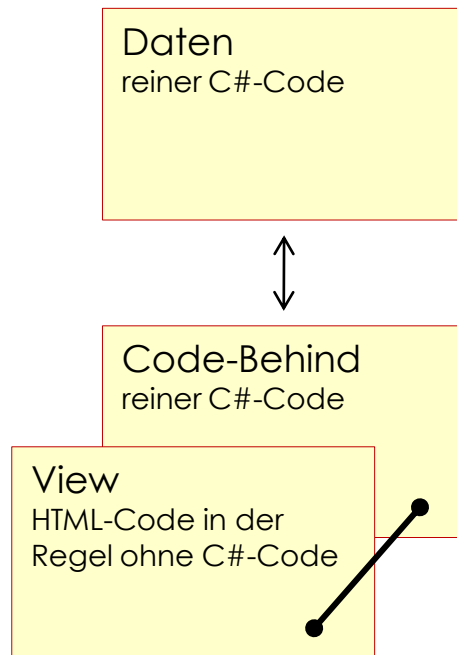
ASP.NET MVC

weroSoft GmbH
Rolf Wenger

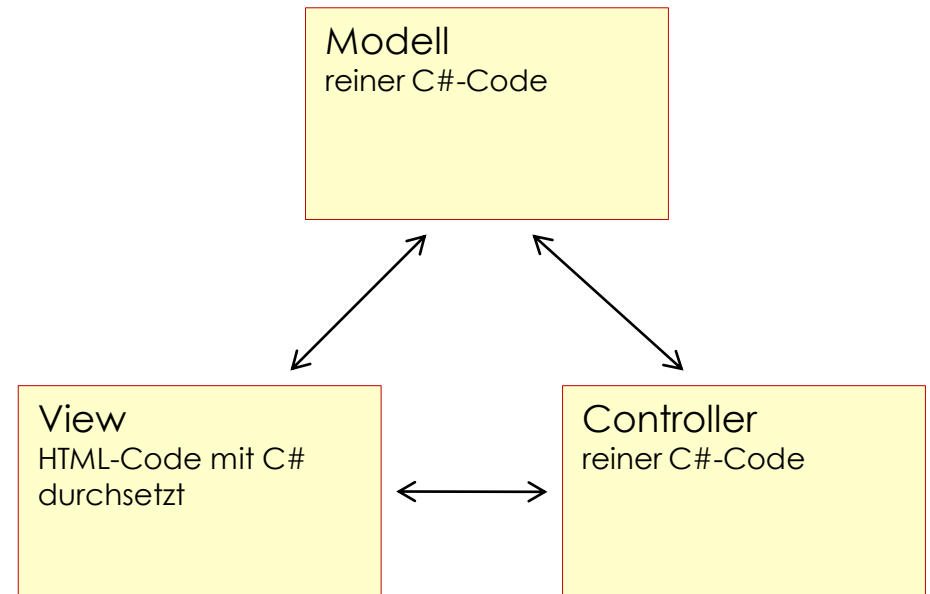
Version 4.0

- ASP.NET MVC ist eine mit .NET 4.0 eingeführte alternative Technik für die Nutzung von ASP.NET
- ASP.NET MVC unterscheidet sich von ASP.NET Forms durch die Architektur

ASP.NET Forms



ASP.NET MVC



- Im Detail liegen die Unterschiede in folgenden Elementen

Merkmal	ASP.NET Forms	ASP.NET MVC
UI Code	Html, JavaScript	Html, Inline C#, JavaScript
Code Behind	Ja	Nein
C# Inline-Code in html	Wenig (Evaluation, Ressourcen)	Standard für Aufbau von Seiten
Wiederverwendbare Controllersteuerung (Multi UI)	Weniger geeignet, aber auch möglich	Durch Architekturmuster direkt unterstützt.
Validierung	Mit Steuerelementen, client- und serverseitig möglich	Auf Datenelementen, client- und serverseitig möglich
Einfach Austauschbare UI	Nein	Ja
Testbarkeit der Funktionalität	Über UI	Ohne UI
Aufteilung Grafik-Design und UI-Funktionalität	CSS, Themes	CSS
Statuskontrolle	Server- und clientseitig	nur serverseitig
URL Benutzerseitig	Datei	Pfad auf Funktionalität
Steuerelemente	Html-Standardelemente, ASP.NET WebElemente, Clientseitige Scripts	Html-Standardelemente, Clientseitige Scripts
Ajax-Unterstützung	Ja	Ja
JQuery-Unterstützung	Ja, Basis und UI	Ja, Basis und UI

- ASP.NET Forms und ASP.NET MVC können in einem Projekt gemischt eingesetzt werden.
- Solange die gegenseitigen Aufrufe der Seiten innerhalb der ASP.NET Technik bleiben sind keine Spezialitäten zu berücksichtigen
- Für Übergänge von der einen Technik in die andere gilt:
 - URL Technik berücksichtigen
 - Zustandsverwaltung berücksichtigen
- Das Mischen von Inhalten in den Masterseiten ist ebenfalls unterstützt
- Ich empfehle eine Mischung allenfalls Gebietsweise und nicht ein kunterbuntes Durcheinander zu veranstalten.

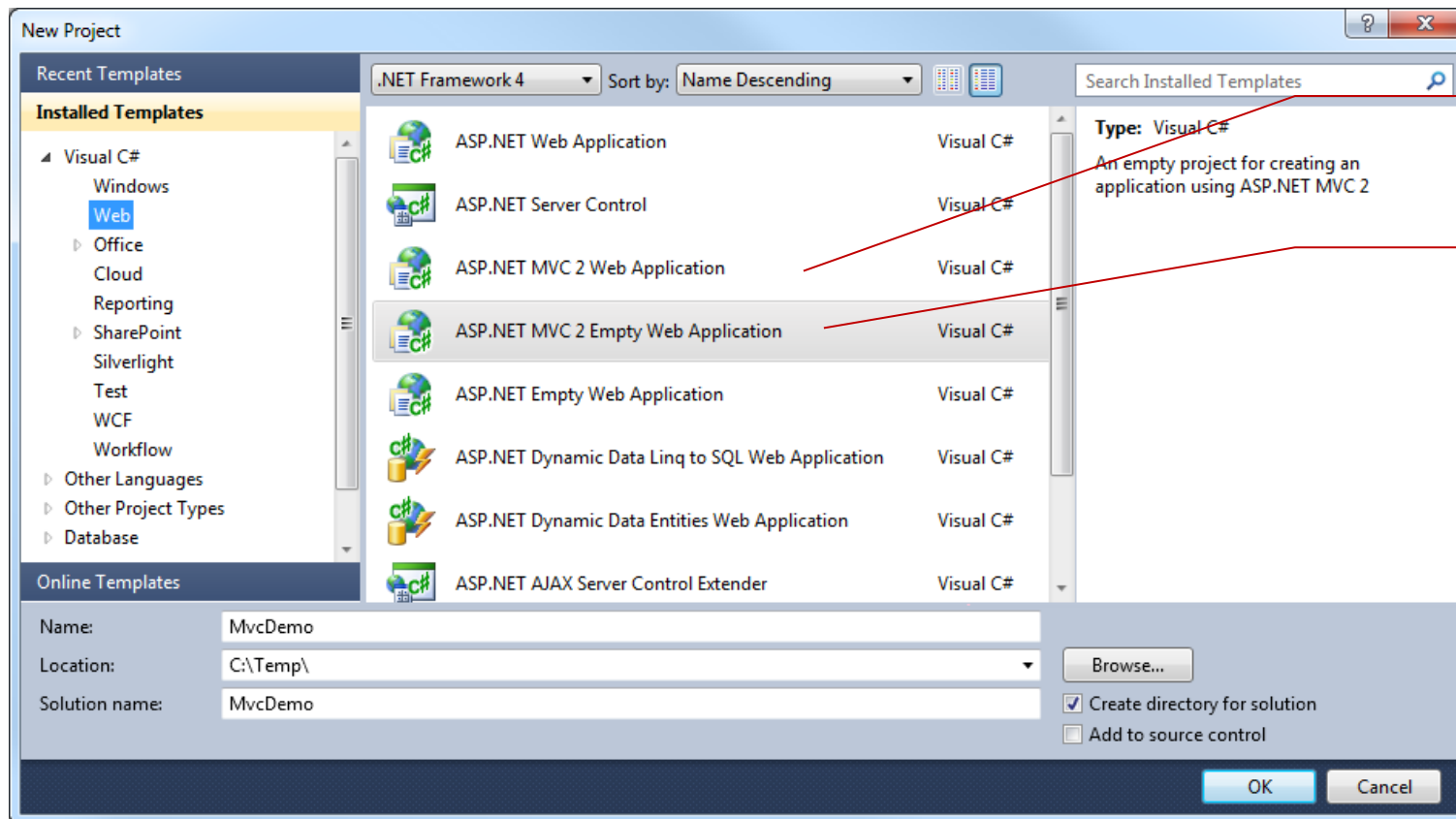
- Das Model-View-Controller Entwurfsmuster (deutsch: Modell-Ansicht-Steuerung) verfolgt primär die Zielsetzung der Flexibilisierung der Anwendung in Bezug auf Änderungen und Wiederverwendung.
- In Bezug auf ASP.NET stehen dabei die Flexibilität in Bezug auf Testbarkeit (Automation durch Unittests) und die Wiederverwendung für verschiedene Benutzeroberflächen zur Verfügung
- Das Model enthält die darzustellenden Daten. Die Daten sind abgekoppelt von den Funktionalitäten zu implementieren. Die Daten können selbstverständlich auch Funktionalitäten beinhalten, solange diese zur Behandlung der Daten selber dienen. Dazu gehören zum Beispiel Umformungen oder Validierungen.
- Die View kümmert sich lediglich um die Darstellung der Daten, welche in ASP.NET MVC in HTML-Steuerelementen erfolgt. Die View erhält ihre Daten entweder vom Controller oder erhält vom Controller mindestens die Anweisung welche Daten anzuzeigen sind, damit die View die Daten direkt benutzen kann.
- Der Controller implementiert die Aktionslogik des Ganzen. Er ist zuständig für den Empfang von Veränderungen aus der View und die daraus resultierende Behandlung der Daten. Dazu gehört auch die Ansteuerung der Persistenz. Der Controller kann seinerseits wieder auf eine Businesslogik zurückgreifen. In diesem Fall reduziert sich seine Funktionalität oft auf ein einfaches Routing der Aktionen zu den eigentlichen Elementen.



Grundlegender Aufbau

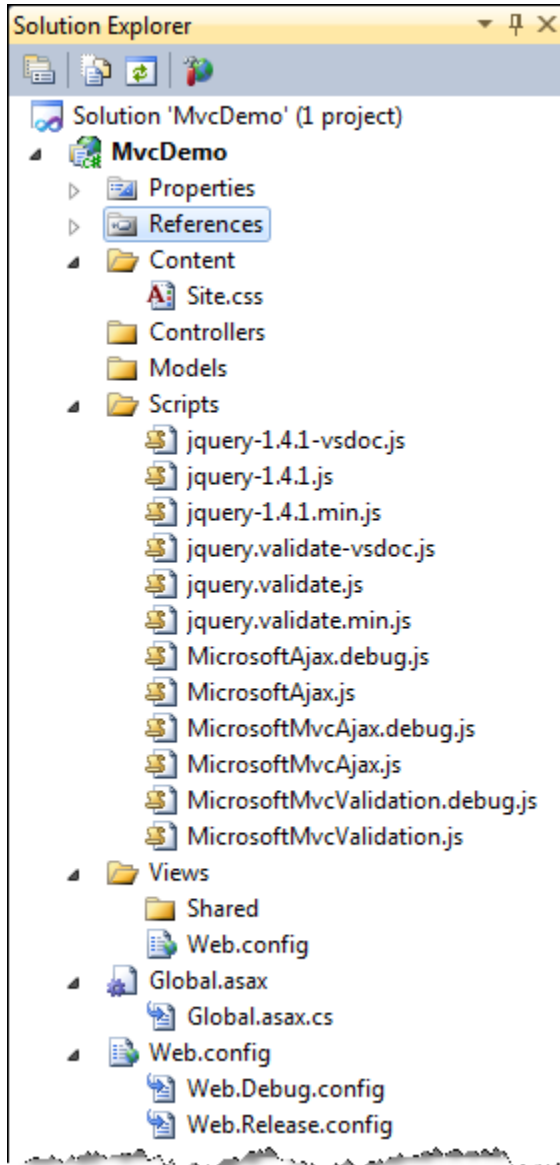
Wie immer – Der Start erfolgt mit dem Assistenten

- ASP.NET MVC Anwendungen werden im Stil einer Web-Anwendung mit einem separaten Assistenten erstellt (siehe auch ASP.NET)



Anwendung mit
Standardlayout und
Loginfunktionalität.

Leere Anwendung mit
eigenem Grundaufbau.



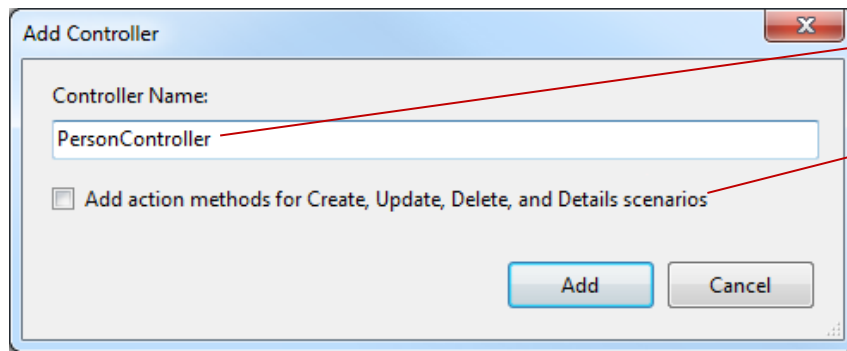
- Nach dem Generieren des Projekts gestaltet sich das Projekt mit einer Struktur und darin teilweise bereits vorhandenen Daten.
- Zu den Daten gehören:
 - Eine vorbereitete Struktur für die Aufnahme der Klassen des MVC-Patterns (Verzeichnisse Controllers, Models, Views)
 - Ein einfaches CSS das eine Handvoll Einträge für die Formatierung von Validierungsfehler definiert (keine Bodyeinträge)
 - Mehrere JavaScript-Dateien je in einer Debug- und einer optimierten Release-Version
 - Vorbereitete Anwendungskonfigurationen (eine für das gesamte Projekt und eine für die Beeinflussung der Views)
 - Die Klasse für das Anwendungsobjekt
- Die Projekteinstellungen entsprechen des Einstellungen für Web-Anwendungen und allgemeinen .NET-Anwendungen

- Um mit dem Kontroller eine sinnvolle Aktion auszulösen brauchen wir zuerst eine Klasse (oder natürlich die Klassen) für die Abbildung unserer Daten
- Diese Klasse ergänzen wir als solche im Verzeichnis Model (Add/Class...)
- Die Klasse wird als nächstes mit den Datenelementen ergänzt

```
namespace WeroSoft.Samples.Models
{
    public class CwlbPerson
    {
        public string Id { get; set; }
        public string Name { get; set; }
        public string FirstName { get; set; }
        public string Address { get; set; }
        public string Email { get; set; }
        public DateTime Birthdate { get; set; }
    }
}
```

- Beachten Sie, dass Visual Studio aufgrund der Verzeichnisstrukturen auch die Namensräume jeweils entsprechend den Strukturen ergänzt.

- Als nächstes definieren wir, was wir mit den Daten tun wollen. Dazu verwenden wir Controller
- Wir erstellen eine Controller-Klasse mit dem Assistenten über Visual Studio, beginnend mit dem Menübefehl "Add/Controller..." im Kontextmenü des Controller Verzeichnisses.
- Im darauf folgenden Dialog definieren wir den Namen der Controllerklasse



Achten Sie darauf, dass Sie den Suffix „Controller“ stehen lassen!

Mit dieser Option generieren Sie einen vollwertigen CRUD-Controller.

```
namespace WeroSoft.Samples.Controllers
{
    public class PersonController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Das Resultat der Generierung finden wir in der Datei PersonController.cs im Verzeichnis Controllers.

Der Code definiert die so genannte Standardaktion.

- Damit unsere Demo auch tatsächlich mit Daten arbeiten kann, soll diese zunächst statisch ein paar Daten anlegen und diese als Liste zur Verfügung stellen. Wir ergänzen diesen nicht ganz Typischen Code direkt im Controller

```
public class PersonController : Controller
{
    // Notwendiger Code für die Demo
    private static List<Models.CwlbPerson> _cobjData = new List<Models.CwlbPerson>();

    public PersonController()
    {
        _cobjData.Add(new Models.CwlbPerson { Id = "1", Name = "Plumber", ... });
        _cobjData.Add(new Models.CwlbPerson { Id = "2", Name = "Hutmacher", ... });
        _cobjData.Add(new Models.CwlbPerson { Id = "3", Name = "Gilgur", ... });
        _cobjData.Add(new Models.CwlbPerson { Id = "4", Name = "Zanetti", ... });
        _cobjData.Add(new Models.CwlbPerson { Id = "5", Name = "Agadir", ... });
    }

    // Effektiver Code des Controller
    public ActionResult Index()
    {
        return View(_cobjData);
    }
}
```

Unser Controller gibt die Daten als Liste wieder.

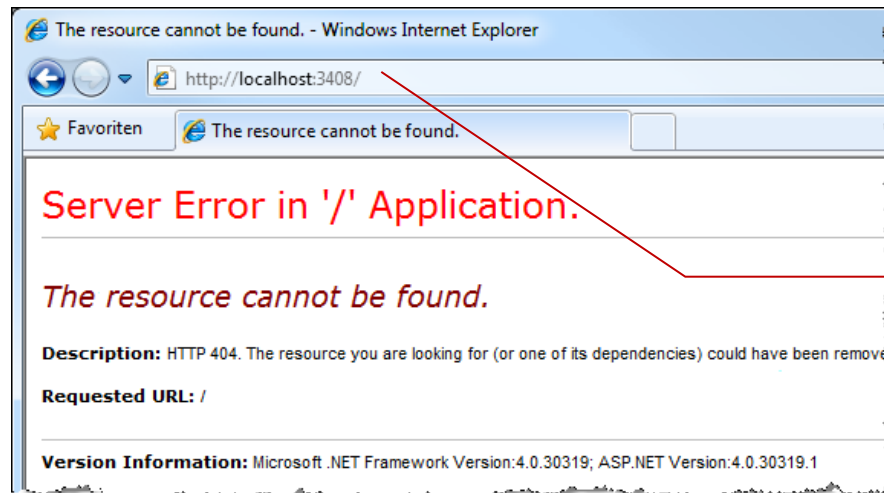
- Für die erste erfolgreiche Benützung der Anwendung fehlt uns nun noch eine Ansicht, die die erstellten Daten darstellt.
- Wir generieren eine View über den Menübefehl "Add View" des Kontextmenüs der entsprechenden Methode des Controllers (hier Index)

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<IEnumerable<WeroSoft.Samples.Models.CwlbPerson>>" %>
<!DOCTYPE html PUBLIC ...>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
</head>
<body>
    <table>
        <tr><th></th><th>Id</th><th>Name</th>...</tr>
        <% foreach (var item in Model) { %>
            <tr>
                <td>
                    <%: Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) %> |
                    <%: Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ })%> |
                    <%: Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })%>
                </td>
                <td><%: item.Id %></td><td><%: item.Name %></td>...
            </tr>
        <% } %>
    </table>
    <p>
        <%: Html.ActionLink("Create New", "Create") %>
    </p>
</body>
</html>
```

Tabellenkopf

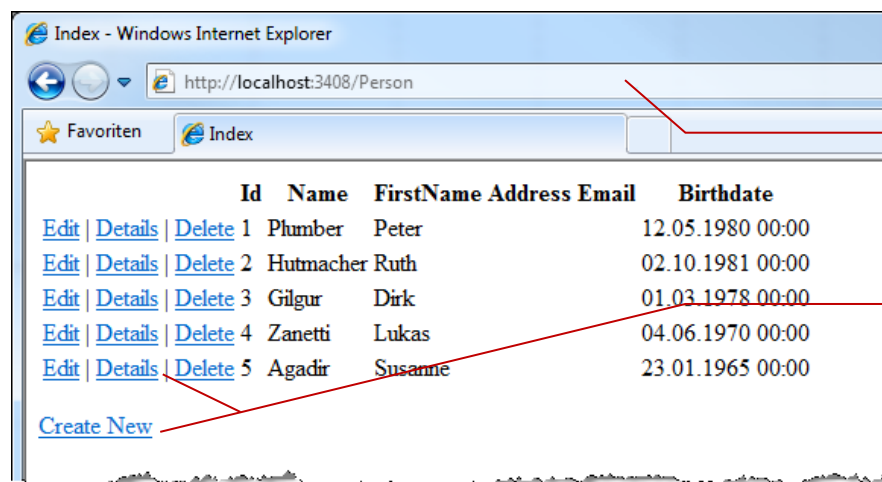
Wiedergabe der Daten aus der Liste.

- Visual Studio erlaubt im MVC Pattern den Aufruf der einzelnen Views leider nicht direkt
- Der Aufruf erfolgt über einen Pfad direkt zum Kontroller
- Wir starten den Browser und Ergänzen die URL mit den notwendigen Angaben



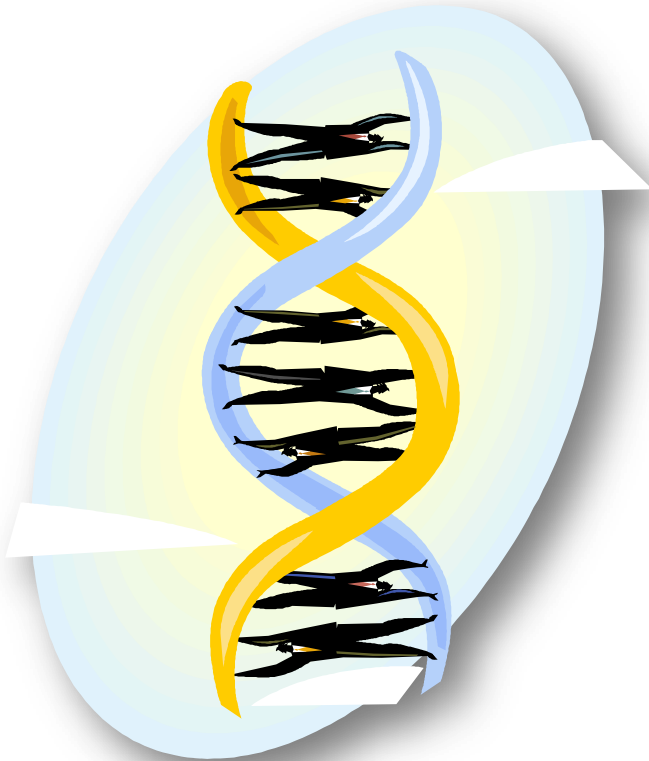
Mit dem reinen Starten der Anwendung sind wir nicht erfolgreich, weil das MVC Verhalten der Anwendung keinen entsprechenden Controller definiert.

Der definierte Defaultname wäre "Home".



Beim zweiten Versuch ergänzen wir den Namen des von uns hergestellten Controllers "Person" und schon funktioniert unsere Lösung.

Die generierten Verknüpfungen funktionieren noch nicht!



Anatomie der ASP.NET MVC-Anwendung

- In einer ASP.NET-Webseite wird eine URL der Browseranforderung normalerweise einer Datei auf dem Server zugeordnet (Beispiel: <http://www.werosoft.net/public/dynamiccontent.aspx>)
- Das ASP.NET-MVC –Framework ordnet einer Browseranforderung stattdessen einer Aktion eines Controllers zu (Beispiel: <http://localhost:3408/Person>)
- Der Controller verarbeitet den Input direkt und liefert dem Framework Daten, die dieses mit Hilfe einer definierten View-Klasse in HTML rendert und dem Browser zurücksendet.
- Die Routingregeln für den Controller werden in der Anwendungsklasse beim Starten der Anwendung definiert.

```
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            "Default", // Route name
            "{controller}/{action}/{id}", // URL with parameters
            new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }

    protected void Application_Start() { ... }
}
```

- Die Definition einer Route basiert auf dem Prinzip eines Pfades mit der Hierarchie:
 - Controller
 - Aktion
 - Parameter
- Die Route wird in einem Verzeichnis abgelegt und erhält dazu einen Namen

```
routes.MapRoute(  
    "Default",  
    "{controller}/{action}/{id}",  
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
);
```

Name der Route

Muster der Route

Standardwerte der Route

- Das Routing einer URL auf einen Controller kann somit sehr einfach vorgenommen werden

```
routes.MapRoute(  
    routes.MapRoute(  
        "OldRouteToNew",  
        "OldControllerName/OldActionName",  
        new { controller = "Person", action = "Index" }  
    )  
);
```

Name der Route

Muster der Route (zum Beispiel alte Route)

Standardwerte der Route alten Route

```
routes.MapRoute(  
    "Default",  
    "{controller}/{action}/{id}",  
    new { controller = "Person", action = "Index", id = UrlParameter.Optional }  
);
```


- Controller werden im Verzeichnis Controller untergebracht
- Die Klassennamen der Controller tragen den Suffix Controller
- Ein Controller kann beliebig viele Aktionen beinhalten
- Aktionen können überladen werden
- Aktionen können für HTTP get und HTTP post mit einem Attribut unterschieden werden. Wird das Attribut `HttpPostAttribut` nicht definiert gilt HTTP get.
- Aktionen liefern in der Regel Objekte als Views
- Aktionen können Routings definieren. Das kann zum Beispiel zum Verzweigen zu einer Folgeseite benutzt werden.
- Aktionen haben Zugriff auf den `HttpContext` und somit auf folgende Elemente:
 - `Application`
 - `ApplicationState`
 - `SessionState`
 - `QueryString`
 - `Request`
 - `Response`

- Ein Controller kann über Visual Studio für folgende Aktionen generiert werden
 - Auflisten einer Übersicht
 - Erstellen einer neuen Information (get und post)
 - Anschauen einer bestehenden Information
 - Löschen einer bestehenden Information
- Das Grundmuster der dabei zum Einsatz gelangenden Technik ist immer das selbe

```
public class Person2Controller : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult Details(int id)
    {
        return View();
    }

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

    [HttpPost]
    public ActionResult Create(FormCollection collection)
    {
        try
        {
            // TODO: Add insert logic here
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }
}
```

Übernimmt einen benannten Parameter aus dem Modell

Definiert die Aktion als HTTP Post-Verarbeitung

HTTP Post, nimmt eine Liste der Daten entgegen, die übergeben werden

Routing einer Verarbeitung zu einer andern Aktion

...

- Eine View wird als Seite oder Inhaltsseite einer Masterseite hergestellt.
- Die Direktive definiert eine Klasse des Typs `ViewPage<T>`, wobei T ein Typ der Modellklassen ist
- Die Klasse enthält im weiteren den notwendigen HTML Code. Alle Elemente des HTML Codes die Variabel sind müssen zur Laufzeit eingesetzt werden. Das wird mit C# Inline-Code gemacht.
- Verwenden Sie für Inline-Code folgende Schreibweise:

`<%: %>`
- Für den Zugriff auf die Daten des Modells benutzen Sie die Eigenschaft `Model`
- Für die Erstellung von HTML Elementen benutzen Sie die Klasse `HtmlHelper`, die über die Eigenschaft `Html` der Seite verfügbar ist
- In Bezug auf Scripting, Cascading Style Sheet und Eingliederung in Masterseite gelten die üblichen Regeln für Web-Anwendungen

- Die Klasse verfügt aufgrund der Vererbung von der bekannten Klasse Page über eine Unmenge von Funktionalität. Zusätzlich definiert sie folgende Elemente:

Member	Zweck
Ajax	Liefert ein Hilfsobjekt für die Programmierung in Szenarien mit Ajax-Einsatz
Html	Liefert ein Hilfsobjekt für den programmatischen Umgang mit Html-Elementen
Model	Verkörpert die Daten des eigenen Modells
ViewData	Liefert eine Auflistung aller Daten die zwischen Controller und View übergeben werden

- Beachten Sie, dass das ASP.NET-MVC Viewstate arbeitet
- Auch werden Postback-Events von Elementen nicht in den gewünschten Events der Steuerelemente landen, sondern in den Controllern. Die wiederum kennen die View nicht, was zur Folge hat, dass die Web-Steuerelemente nicht wirklich arbeiten.

- Für die Klasse HtmlHelper sind vor allem die Erweiterungsmethoden von Bedeutung. Hier eine Zusammenfassung der wichtigsten Methoden:

Methode	Zweck
ActionLink()	Erzeugt einen Link auf eine Action eines Controllers
BeginForm()	Rendert das korrekte Formular Tag für den Aufruf des Controllers mit der richtigen action URL
CheckBox	Erzeugt ein HTML Checkbox Steuerelement
DropDownList()	Erzeugt ein HTML DropDownList Steuerelement
Hidden()	Erzeugt ein HTML Hidden Steuerelement
Label()	Erzeugt ein HTML Label Steuerelement
ListBox()	Erzeugt ein HTML ListBox Steuerelement
Password()	Erzeugt ein HTML Password Steuerelement
RadioButton()	Erzeugt ein HTML RadioButton Steuerelement
TextArea()	Erzeugt ein HTML Steuerelement für mehrzeilige Texteingabe
TextBox()	Erzeugt ein HTML TextBox Steuerelement
ValidationMessage()	Erlaubt die Anzeige einer Fehlermeldung, wenn ein Fehler bei der Validierung eintritt.
ValidationSummary()	Erlaubt die Erstellung eines Validation Summary

- Die Methoden existieren jeweils in der Form gemäss obenstehender Tabelle oder in der Schreibweise `XxxxFor<TModel, Y>()`, die per Lambda den Zugriff auf das Modell erlaubt. Die Methoden `XxxxFor()` können besser mit Null-Referenzen umgehen.

- Die Zustandsverwaltung für ASP.NET-MVC-Anwendungen werden wie folgt unterstützt:

Name der Technik	ASP.NET	ASP.NET-MVC
Viewstate	Ja	Nein
Controlstate	Ja	Nein
Hidden Fields	Ja	Ja
QueryString	Ja	Ja
Cookies	Ja	Ja
Applicationstate	Ja	Ja
Cache	Ja	Ja
Sessionstate	Ja	Ja
Profiling	Ja	Ja

- Beachten Sie, dass in den Controller-Klassen die Zustandsverwaltung zum Teil über die Eigenschaft HttpContext angesprochen werden muss.

- Auch in ASP.NET-MVC ist die automatisierte Validierung von Eingaben ein wichtiges Thema
- Da das Eventmodell und die Web-Steuerelemente nicht verwendbar sind, wurden neue Validierungsmechanismen eingebracht
- Die neuen Validierungsmechanismen basieren auf der Definition von Regeln auf den Elementen der Klassen des Modells
- Es stehen dazu folgende Attribute zur Verfügung

Methode	Zweck
RangeAttribut	Erlaubt die Definition eines numerischen Eingabebereichs (Unterstützt auch Datum)
RegularExpressionAttribute	Erlaubt die Prüfung der Eingabe mit einem regulären Ausdruck
RequiredAttribute	Erlaubt die Prüfung einer Muss-Eingabe
StringLengthAttribute	Erlaubt die Prüfung einer Stringlänge.

- Die Prüfungen können rein serverseitig oder clientseitig vorgenommen werden

- Ein Element darf so viele Prüfausdrücke auf sich vereinen wie notwendig sind

```
public class CwlbPerson
{
    public string Id { get; set; }

    [Required(AllowEmptyStrings=false, ErrorMessage="Name muss eingegeben werden.")]
    [RegularExpression(@"^[a-zA-Z' '-'\s]{1,40}$")]
    public string Name { get; set; }

    [StringLength(40, ErrorMessage="Der Vorname darf nicht länger als 40 Zeichen sein.")]
    public string FirstName { get; set; }

    public string Address { get; set; }

    [RegularExpression(@"[...])?", ErrorMessage = "Scheint keine gültige Email-Adresse zu sein.")]
    public string Email { get; set; }

    [Range(typeof(DateTime), "1900.01.01", "2050.01.01",
        ErrorMessage="Das Datum muss zwischen 1900 und 2050 sein.")]
    public DateTime Birthdate { get; set; }
}
```

- Bei der direkten Verwendung von EF Klassen müssen die Prüfungen über eine spezielle Metadatenklasse eingebracht werden. Siehe auch Attribut `MetadataTypeAttribute`. Das ist notwendig, weil das dekorieren der Attribute in der EF-Klasse vom Generator überschrieben würde.

- Für die clientseitige Prüfung sind lediglich ein paar JavaScripte einzuschalten und eine Anweisung im Body anzubringen:

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<WeroSoft.Samples.Models.CwlbPerson>" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Create</title>
    <script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>
</head>
<body>

    <% Html.EnableClientValidation(); %>

    <% using (Html.BeginForm()) {%>

        <%: Html.ValidationSummary(true) %>
        <%: Html.HiddenFor(model => model.Id) %>

        <fieldset>
            <legend>Fields</legend>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Name) %>
            </div>
            <div class="editor-field">
```



jQuery

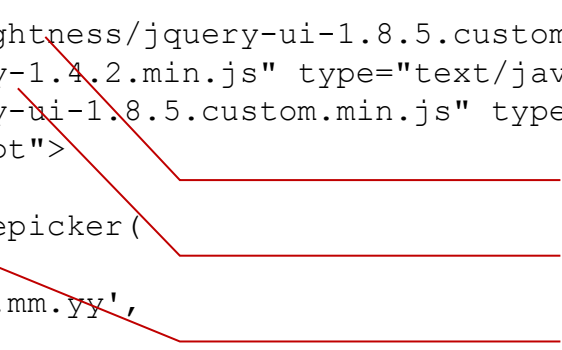
- JQuery ist eine JavaScript-Bibliothek für die vereinfachte Handhabung von HTML-Dokumenten, die Nutzung des Eventmodells im DOM, die Steuerung von Animationen und die Interaktion mit AJAX.
- Der Zusatz JQuery UI liefert zudem eine vielfältige Bibliothek für Steuerelemente und Clientseitige Dialoghandhabung
- ASP.NET-MVC integriert den Standardteil von JQuery automatisch in ein Projekt.
- Der Zusatz JQuery UI können Sie von <http://jqueryui.com> herunterladen
- Die Bibliotheken liegen in zwei bis drei Versionen vor
 - Optimierte Version für die Produktion
 - Debugversion für die Entwicklung
 - Dokumentierte Version für das Erlernen des Umgangs
- Für das Entwickeln setzten Sie wahlweise den Debugger von Visual Studio oder ein Debugger für das DOM integriert in den Browser ein
- Beachten Sie, dass Sie dazu das Debugging im Browser einschalten müssen

- Nach dem Herunterladen der Queries installieren Sie diese in Ihrem Projekt
 - Direkt aus dem zip in das Projekt
 - Oder feingliedrig in die eigene Struktur verstreuen
- Ergänzen Sie das gewählte css und die notwendigen Skripts in der jeweiligen Seite
- Ergänzen Sie den Code für die Elemente, die eine entsprechende Dekoration erhalten

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<WeroSoft.Samples.Models.CwlbPerson>" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Create</title>
    <link href="../../../css/ui-lightness/jquery-ui-1.8.5.custom.css" rel="stylesheet" type="text/css" />
    <script src="../../../js/jquery-1.4.2.min.js" type="text/javascript"></script>
    <script src="../../../js/jquery-ui-1.8.5.custom.min.js" type="text/javascript"></script>
    <script type="text/javascript">
        $(function () {
            $('#Birthdate').datepicker(
            {
                dateFormat: 'dd.mm.yy',
                showWeek: true
            });
        });
    </script>
```



Ergänztes css.

Ergänztes Skripte

Skriptblock für die Konfiguration des eigenen Feldes Birthdate

- Das Erstellen eines clientseitigen Dialogs erfordert folgende Vorkehrungen:
 - Dialog als HTML-Vorlage definieren (ausserhalb des Formulars)
 - Link zum Starten des Dialogs einrichten (Hyperlink, Schaltfläche ...)
 - Script für Dialog erstellen
 - Script für Link zum Starten des Dialogs erstellen
- Die HTML Vorlage könnte so aussehen (Eingabe von drei Feldern):

```
<body>
  <!-- Dialog --%>
  <div id="dlgAddress" title="Adresse eingeben">
    <table>
      <tr>
        <td>Strasse</td>
        <td><input type="text" id="dlgAddressStreet" /></td>
      </tr>
      <tr>
        <td>PLZ</td>
        <td><input type="text" id="dlgAddressZip" /></td>
      </tr>
      <tr>
        <td>Ort</td>
        <td><input type="text" id="dlgAddressCity" /></td>
      </tr>
    </table>
  </div>
  <!-- Dialog --%>
  ...
```

Die Daten des Dialogs sind ausserhalb des Formulars, weil der Dialog auch die Abbrechen Funktionalität unterstützen soll und die Daten zu diesem Zweck fallbezogen übernommen werden.

- Nun werden die restlichen Daten der View für die Nutzung des Dialogs vorbereitet:

```
<!-- Adresse -->
<%= Html.Hidden("Street", Model.Street) %>
<%= Html.Hidden("ZIP", Model.ZIP) %>
<%= Html.Hidden("City", Model.City) %>
<div class="editor-label">
    <%= Html.LabelFor(model => model.Address) %>
</div>
<div class="editor-field">
    <%= Html.TextBox("Address", Model.Address,
        new Dictionary<string, object>{{ "readonly", true }})%>
<a href="#" id="link_address">Adress Details eingeben</a>
</div>
```

Die effektiven Nutzdaten werden hier als versteckte Daten übermittelt. Das ist nicht zwingend notwendig, und hier nur beispielhaft so gelöst.

Die Daten werden ebenfalls als zusammengesetztes Feld in einem

Die effektive Nutzinformation wird als readonly Textfeld eingebracht.

Der Link für das Starten des Dialogs

```
<script type="text/javascript">
$(function () {
    // Handhabung des Dialogs
    $('#dlgAddress').dialog(
    {
        autoOpen: false,
        width: 400,
        height: 250,
        modal: true,
        open: function (event, ui) {
            var address = $('#Street').val();
            var zip = $('#ZIP').val();
            var city = $('#City').val();

            $('#dlgAddressStreet').val(address);
            $('#dlgAddressZip').val(zip);
            $('#dlgAddressCity').val(city);
        },
        buttons: {
            "Speichern": function () {

                var address = $('#dlgAddressStreet').val();
                var zip = $('#dlgAddressZip').val();
                var city = $('#dlgAddressCity').val();

                $('#Street').val(address);
                $('#ZIP').val(zip);
                $('#City').val(city);
                $('#Address').val(address + ', ' + zip + ', ' + city );

                $(this).dialog("close");
            },
            "Abbrechen": function () {
                $(this).dialog("close");
            }
        }
    }
});
```

- Der Schluss des Scripts für das Starten des Dialogs:

```
// Dialog Link
$('#link_address').click(function () {
    $(dlgAddress).dialog('open');
    return false;
});
</script>
```