Kapitel 13

ASP.NET Model View Controller

In diesem Kapitel:

Architektur von ASP.NET MVC	924
Ein erster Entwicklungszyklus	927
Festlegen von Weiterleitungen	933
Den Aufbau des Modells fertigstellen	935
Der Aufbau eines Controllers	936
Der Aufbau einer View	947
Daten validieren	955
Herstellen von partiellen Views	958

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Mit ASP.NET MVC bietet Microsoft einen alternativen Weg der Realisierung von Webanwendungen, indem das MVC-Entwurfsmuster (Model View Controller) als Basis für die Verarbeitung von Daten dient. Die Auswirkungen des veränderten Designs sind eine Teilung der bisher verwendeten Seite, bestehend aus XHTML und Code-Behind in zwei Elemente – die View und den Controller – und der Ersatz von Postback und Ansichtszustand durch einen neuen Routingmechanismus für die Verarbeitung und einen veränderten Lebenszyklus von Seiten.

Architektur von ASP.NET MVC

Auch mit dem Einsatz von ASP.NET MVC bleibt die Zweiteilung der Webanwendung in einen Client- und einen Serverteil selbstverständlich erhalten. Weiterhin sendet der Client, gesteuert durch seinen Browser, Anforderungen an den Server und die Aufgabe des Servers ist es, eine Anforderung zu verarbeiten und eine Antwort als vollständige HTML-Seite zu formulieren und an den Client zurückzusenden.

Der Unterschied von ASP.NET MVC beginnt mit dem URL, der nicht mehr eine *.aspx*-Seite anspricht, sondern direkt einen Controller des MVC-Entwurfsmusters. Der Controller ist es denn auch, der die Arbeit auf der Serverseite übernimmt und mit seiner Logik für die Benutzung des Modells und der View sorgt. Die Abbildung 13.1 visualisiert das Prinzip der Verarbeitung mit ASP.NET MVC.

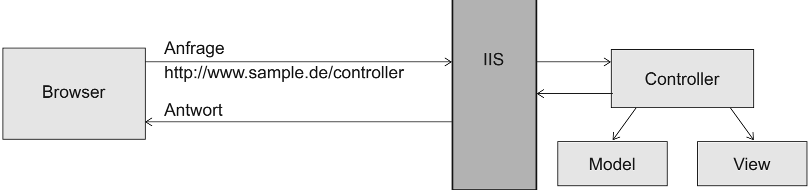

Abbildung 13.1 Übersicht über die Verarbeitung einer Anforderung mit ASP.NET MVC

Aufteilung des MVC-Entwurfsmusters

Die bisherigen Betrachtungen von ASP.NET haben auf der Serverseite die *.aspx*-Datei in das Zentrum der Verarbeitung gerückt. Die gesamte Programmierung der Ereignisverarbeitung und das Rendering des Modells erfolgen beim normalen ASP.NET innerhalb der Seitenklasse. Genau hier finden wir in ASP.NET MVC den größten Unterschied der beiden Architekturen. Die bisherige Seitenklasse wird zweigeteilt in eine Klasse für die Kontrolle der Verarbeitung (Controller) und eine Ansicht.

Abbildung 13.2 zeigt die unterschiedliche Aufteilung der Elemente im Vergleich zwischen ASP.NET (links) und ASP.NET MVC (rechts).

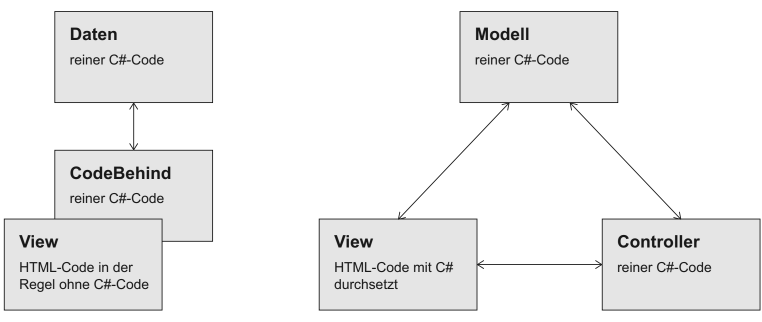

Abbildung 13.2 Unterschiedliche Aufteilung der verwendeten Elemente zwischen ASP.NET und ASP.NET MVC

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Die Verantwortlichkeiten der einzelnen Komponenten des MVC-Entwurfsmusters sind wie folgt definiert:

- **Model** Das Modell repräsentiert in einfachen Anwendungen direkt die Datenschicht der Anwendung. In größeren Anwendungen repräsentiert das Modell den Zugriff auf die Geschäftslogik, die wiederum auf anderen Servern losgelöst von der Benutzeroberfläche realisiert werden kann. Das Modell ist an und für sich in einer normalen ASP.NET-Anwendung auch vorhanden. Dadurch unterscheiden sich die beiden Architekturen also am wenigsten.
- **View** Die View definiert die Ansicht der Daten, die vom Controller verarbeitet werden. Sie ist somit für die Formatierung der Ausgabe zuständig. Die View ist jedoch nicht zuständig für die Verarbeitung von Eingabedaten. Sie wird ja auch vom Browser nicht angesprochen. Die View kombiniert die Ausgabe mit den Daten aus dem Modell.
- **Controller** Der Controller ist das verarbeitende Element. Er ist zuständig für die Verarbeitung der Eingaben, die Validierung der Daten, das Ansprechen des Modells und die Zusammenführung der Modelldaten mit der View.

Der Lebenszyklus einer Verarbeitung mit ASP.NET MVC

Der Lebenszyklus einer Verarbeitung in ASP.NET MVC beginnt nach dem Eintreffen einer Anfrage im URL-Routingmodul. Dieses Modul implementiert die Schnittstelle `IHttpModule` und ist zuständig für das Festlegen der Weiterleitung der Anfrage. Zu diesem Zweck stellt eine Anwendung in der Datei *global.asax* spezielle Informationen bereit. Nach dem Festlegen der Weiterleitung erstellt das Routingmodul den für die Verarbeitung wichtigen HTTP-Kontext und kombiniert diesen mit der Weiterleitung und der Anfrage zu einem Verarbeitungsobjekt.

Nach dieser Vorverarbeitung werden die gewonnenen Anfragedaten an den *MvcRouteHandler* weitergereicht. Seine Aufgabe besteht darin, die Anfrage zu analysieren und festzulegen, welche Controllerklasse für die Verarbeitung benutzt werden soll. Diese Information wird letztendlich an den *MvcHandler* weitergereicht, der nun seinerseits für die Instanziierung und die Ausführung des Controllers verantwortlich ist. Am Ende der Ausführung der Funktionalität erstellt der Controller die View, rendert diese als Antwort und löst die Übermittlung an den Client aus.

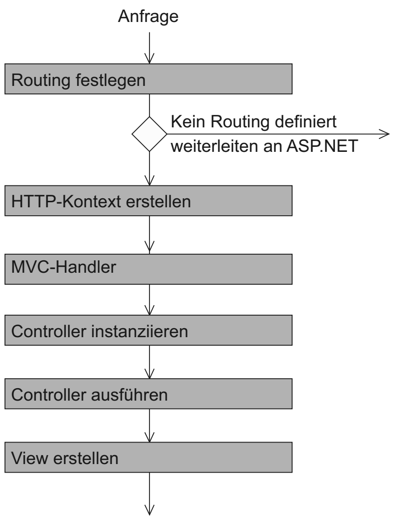

Abbildung 13.3 Lebenszyklus einer Verarbeitung mit ASP.NET MVC

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

HINWEIS Kann am Anfang der Verarbeitung das URL-Weiterleitungsmodul keinen Verarbeitungsweg festlegen, wird die Anfrage an ASP.NET zur normalen Regelverarbeitung weitergeleitet. Es ist dann möglich, eine Webanwendung als normale ASP.NET Anwendung durchmischt mit ASP.NET MVC zu entwerfen.

Überblick über die Unterschiede zwischen ASP.NET und ASP.NET MVC

Für das Verständnis der Differenzen zweier Dinge, die letztendlich das Gleiche tun, ist es manchmal von Vorteil, die Differenz plakativ herauszustellen. Tabelle 13.1 stellt die beiden Architekturen anhand von ausgesuchten Merkmalen gegenüber.

Merkmal	ASP.NET	ASP.NET MVC
Codierung der Benutzeroberfläche	XHTML, JavaScript	XHTML, JavaScript
CodeBehind	Ja	Nein
C# inline in XHTML	Wenig, für Evaluation, Ressourcen	Standard für Aufbau der Seite
Steuerelemente	HTML-Standardelemente, ASP.NET Webserver-Steuerelemente	HTML-Standardelemente
Wiederverwendbare Controllersteuerung, Multi-UI für mehrere Bildschirmformate wie mobile Clients und normale Desktops	Weniger geeignet	Durch Architektur direkt unterstützt
Unterstützung Masterseiten	Ja	Ja
Unterstützung Anwendungszustand	Ja	Ja
Unterstützung Sitzung	Ja	Ja
Unterstützung Ansichtszustand	Ja	Nein
Unterstützung Cookies	Ja	Ja
Unterstützung Abfragezeichenfolge	Ja	Ja
Validierung	Mit Steuerelementen, client- und serverseitig möglich	Auf Datenelementen, client- und serverseitig möglich
Einfach austauschbare UI	Nein	Nicht direkt, aber geringer Aufwand, da UI mehrheitlich funktionslos ist
Testbarkeit der Funktionalität	Über UI	Ohne UI möglich
Unterstützung Gestaltung	Themen und CSS	CSS
URL benutzerseitig	URL auf Ressource, normalerweise eine *.aspx*-Datei	URL als Pfad auf angesprochene Funktionalität
AJAX-Unterstützung	Ja	Ja
jQuery-Unterstützung	Ja, Basis und UI	Ja, Basis und UI

Tabelle 13.1 Unterschiede zwischen ASP.NET und ASP.NET MVC

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Ein erster Entwicklungszyklus

Um möglichst rasch mit der Entwicklung einer Anwendung mit der Architektur von ASP.NET MVC vertraut zu werden, erstellen wir in einem ersten Entwicklungszyklus ein erstes Beispiel. Wir finden den Einstieg auch hier im vertrauten Assistenten für die Erstellung von neuen Projekten. In den Vorlagen für die Webentwicklung stehen für die Erstellung von ASP.NET MVC-Anwendungen zwei Vorlagen zur Verfügung.

Die Vorlage *ASP.NET MVC 2 Empty Application* generiert die Solution und das dazugehörige Projekt mit seiner Projektstruktur ohne Komponenten. Verwenden Sie diese Vorlage, wenn Sie von Grund auf eine neue Anwendung selbst aufbauen wollen. Die zweite Vorlage mit Namen *ASP.NET MVC 2 Application* erzeugt ein Projekt, das bereits mit einer Masterseite ausgerüstet ist, und für jede Komponente des MVC auch bereits eine Standardklasse definiert. Für unser Vorhaben der Untersuchung des Aufbaus einer ASP.NET MVC-Anwendung ist die erste genannte Vorlage besser. Nach dem Generieren des Projekts gestaltet sich dessen Inhalt gemäß Abbildung 13.4.

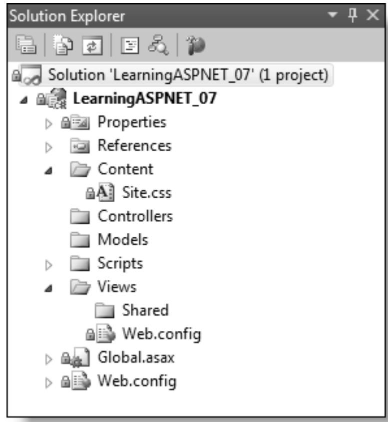

Abbildung 13.4 Inhalt des Projekts nach dem Generieren aus der Vorlage *ASP.NET MVC 2 Empty Application*

Die generierte Struktur des Projekts definiert mit den Ordnern *Controllers*, *Models* und *Views* bereits die Speicherorte der entsprechenden Klasse, die wir später erzeugen werden. Nehmen Sie an dieser Stelle zur Kenntnis, dass Sie diese Ordner besser nicht umbenennen, denn die Namen sind hier auch Programm. Im Weiteren enthält das generierte Projekt bereits ein Cascading Style Sheet sowie eine *global.asax*-Datei und zwei Anwendungskonfigurationen *web.config*. Die Datei *global.asax* ist für uns sehr interessant, weil diese Datei auch bereits generierten Code beinhalten kann.

Listing 13.1 zeigt denn auch, dass dem tatsächlich so ist. Die definierte Klasse `MvcApplication` erbt von der bekannten Klasse `HttpApplication` (siehe Kapitel 10, Abschnitt »Das Anwendungsobjekt«) und benutzt das Ereignis `Application_Start()` für die Initialisierung der Anwendung, indem die eigene Methode `RegisterRoutes()` aufgerufen wird. Diese Methode registriert in der Auflistung der Weiterleitungen eine neue Weiterleitung für die Anwendung. Weiterleitungen werden von ASP.NET MVC benutzt, um die eintreffende Anforderung zu erkennen und mit einer definierten Funktionalität zu verbinden.

Mehr zum Aufbau einer Weiterleitung lesen Sie im Abschnitt »Festlegen von Weiterleitungen«. Für unsere aktuellen Bedürfnisse lassen wir den Code unverändert und wenden uns dem nächsten Schritt zu.

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
// Anwendungsklasse der ASP.NET MVC-Anwendung
public class MvcApplication : System.Web.HttpApplication {

    // Registriert die definierten Weiterleitungen der Anwendung
    public static void RegisterRoutes(RouteCollection routes) {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            "Default",                          // Name der Weiterleitung
            "{controller}/{Aktion}/{id}",       // URL mit Parameter
            new {                               // Parameterstandardwerte
                controller = "Home",
                Aktion = "Index",
                id = UrlParameter.Optional });
    }

    // Löst die Registrierung der Weiterleitungen beim Start der Anwendung aus
    protected void Application_Start(){
        AreaRegistration.RegisterAllAreas();
        RegisterRoutes(RouteTable.Routes);
    }
}
```

Listing 13.1 Generierte Datei *global.asax*

Erstellen eines Modells

Für die Nutzung des MVC-Entwurfsmusters beginnen wir mit der Erstellung einer Modellklasse. Dieser Weg ist klassisch, denn ohne Daten bedarf es in der Regel keines Controllers und dementsprechend auch keiner Views.

Das Beispiel dieses Abschnitts sorgt sich wieder einmal um die Verwaltung von Personen. Dazu benötigen wir eine Klasse für die Abbildung einer Person. Die Klasse erstellen wir im Ordner *Models* als normale C#-Klasse gemäß Listing 13.2.

```
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; }
    }
}
```

Listing 13.2 Verwendete Modellklasse für das Beispiel

Damit das Modell nicht nur eine Person, sondern mehrere Personen verwalten kann, benötigen wir auch noch eine entsprechende Liste. Diese realisieren wir gekapselt in einer normalen C#-Klasse, die uns gleichzeitig als simulierte Datenbank dient. Diese Klasse wird später in den Erläuterungen auch die Veränderung der Personenliste erlauben. Da die Klasse dazu dient, das Modell als Ganzes zusammenzuhalten, speichern wir auch sie im Ordner *Model*.

Listing 13.3 zeigt die vorerst sehr einfache Codierung der Klasse `CwlbDataContext` für die Führung der Daten. Die Klasse ist als Singleton gestaltet und erstellt bei der Instanziierung automatisch ein paar Demodaten.

```
// Simuliert den Datenbankzugriff
public sealed class CwlbDataContext {

    // Liefert den aktuellen Datenkontext der Anwendung
    public static CwlbDataContext Current { get; private set; }

    // Liste der Personen
    private List<CwlbPerson> _cobjPersons = new List<CwlbPerson>();

    /// Liefert eine unveränderliche Liste mit Personen
    public ReadOnlyCollection<CwlbPerson> Persons {
        get { return new ReadOnlyCollection<CwlbPerson>(_cobjPersons); }
    }

    // Erstellt das Modell
    static CwlbDataContext(){
        Current = new CwlbDataContext();
    }

    // Initialisiert eine neue Instanz der Klasse CwlbDataContract
    private CwlbDataContext(){
         _cobjPersons.Add(new CwlbPerson {
            Id = "001", Name = "Albligen", FirstName = "Peter", Address = "Albisweg 34\r\nCH-5678 Berg",
            Email = "p.a@sample.de", Birthdate = new DateTime(1985, 12, 23)
         });
         ...
    }
}
```

Listing 13.3 Klasse für die Simulation einer Datenbank

Ergänzen eines Controllers

Der nächste Schritt im Aufbau der Anwendung besteht darin, einen Controller zu erstellen. Ein Controller definiert eine oder mehrere Funktionalitäten, die vom Browser angesprochen werden können. Die Erstellung des Controllers beginnt im Kontextmenü des Ordners *Controllers* durch die Auswahl des Kontextmenübefehls *Add/Controller*. Im darauf folgenden Dialogfeld *Add Controller* definieren Sie den Namen des Controllers (siehe Abbildung 13.5). Achten Sie darauf, dass Sie den Suffix `Controller` stehen lassen und den gewünschten Namen zweckgebunden davor schreiben. Das angebotene Kontrollkästchen *Add Action methods for Create, Update, Delete and Details scenarios* wird später dazu verwendt, die CRUD-Aktionen des Controllers für die komplette Verarbeitung einer Information direkt zu generieren. Im ersten Schritt ignorieren wir diese Option jedoch.

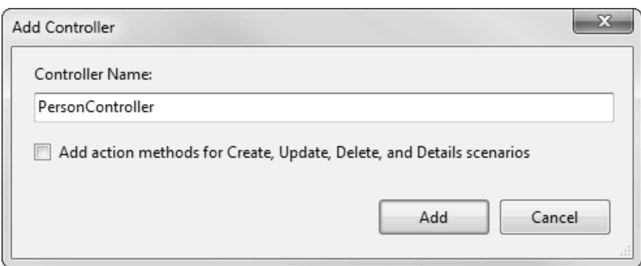

Abbildung 13.5 Dialog für die Erstellung eines Controllers

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Nach der Bestätigung der Eingaben mit der Schaltfläche *Add* wird der sehr einfache Code, wie er in Listing 13.4 dargestellt ist, generiert. Darin ist eine so genannte Aktion mit dem Namen `Index()` enthalten. Eine Aktion ist eine Methode, die vom Browser im Controller angesprochen werden kann, um eine bestimmte Funktionalität auszuführen. Für jeden Controller kann eine Standardaktion definiert werden, die Verwendung findet, wenn die Anforderung des Browsers nur den Controller, aber nicht eine bestimmte Aktion anspricht. Die dazu notwendige Definition ist in der Registrierung für die Weiterleitung in der Datei *global.asax* vorhanden (siehe Listing 13.1) und wird im Abschnitt »Festlegen von Weiterleitungen« besprochen.

```
public class PersonController : Controller {

    public AktionResult Index(){
        return View();
    }
}
```

Listing 13.4 Generierter Code des Controllers `PersonController`

Der generierte Code hat natürlich noch keine Funktionalität und bedarf einer minimalen Anpassung. Die generierte Standardaktion `Index()` wollen wir ausnutzen, um eine Liste aller Personen an den Browser zurückzugeben. Die dazu notwendige Änderung besteht darin, der erzeugten View die Liste der Personen zu übergeben (siehe Listing 13.5).

```
// Controller für die Verarbeitung von Anforderungen an die Verwaltung von Personen
public class PersonController : Controller {

    // Aktion für die Handhabung der Liste der Personen
    public AktionResult Index(){
        return View(CwlbDataContext.Current.Persons);
    }
}
```

Listing 13.5 Leicht angepasster Code für die Erstellung einer Personenliste

Damit ist der erste Controller hergestellt, und wir können uns dem letzten Schritt für diesen ersten Entwicklungszyklus zuwenden – dem Erstellen der View.

Hinzufügen einer View

Da in ASP.NET MVC Views in reinem XHTML hergestellt werden, entstehen aufgrund der Verwendung von unterschiedlichen Steuerelementen in den verschiedenen Situationen, für die verschiedenen CRUD-Operationen in der Regel mindestens je eine *.aspx*-Datei. Das ergibt eine deutlich höhere Zahl Dateien, als dies bei normalen ASP.NET-Anwendungen der Fall ist. Die Strukturen von MVC berücksichtigen dies, indem für jeden Controller im Ordner *Views* ein weiterer Ordner mit dem Namen des Controllers erstellt wird. In unserem Beispiel besteht somit der erste Schritt für die Erstellung der View darin, im Ordner *Views* den neuen Ordner *Person* zu erstellen. Auch hier lassen wir den Suffix `Controller` des Klassennamens weg.

Ausgehend vom soeben erstellten Ordner nutzen wir den Kontextmenübefehl *Add/View* und starten die Erstellung der View. Die Reaktion von Visual Studio besteht in der Anzeige des Dialogfelds *Add View*. Hier definieren wir im Feld *View name* den Namen der Aktion des Controllers, der diese View benutzen wird. In unserem Fall ist das die Aktion `Index()`. Anschließend definieren wir die Erstellung einer streng typisierten

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

View und geben an, dass die View mit einer Liste von Objekten der Klasse `CwlbPerson` arbeiten soll (siehe Abbildung 13.6).

HINWEIS Die Grundlage für die drei Bände des Handbuchs der .NET 4.0-Programmierung bildet die englischsprachige Version von Visual Studio 2010. Infolgedessen werden generierte Steuerelemente der Benutzeroberfläche von Visual Studio in Englisch beschriftet.

Eine Übersetzung in eine andere Sprache ist später im generierten Code direkt möglich, respektive kann durch die Mechanismen der Lokalisierung erfolgen.

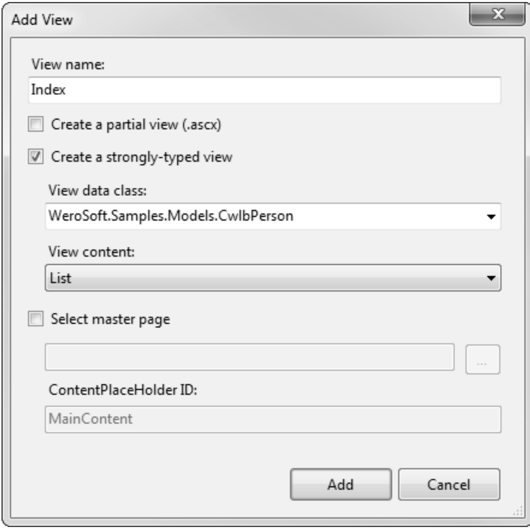

Abbildung 13.6 Dialogfeld für die Erstellung einer View in ASP.NET MVC

Nach der Bestätigung der Eingaben mit der Schaltfläche *Add* erzeugt Visual Studio im definierten Verzeichnis die entsprechende Klasse in Form einer *.aspx*-Datei. Das Resultat entnehmen Sie in gekürzter Form Listing 13.6. Beachten Sie, dass die generierte Datei keinen CodeBehind besitzt, jedoch anstelle dessen ein bisschen Verarbeitungslogik in Form von C#-Inlinecode verwendet. Die dazu notwendige Syntax mit den Zeichen `<% … %>` ist in Kapitel 10, Tabelle 10.11 beschrieben.

Bei genauer Betrachtung des Codes können Sie feststellen, dass mithilfe des Inlinecodes für jedes Personenobjekt in der Liste ein Tabelleneintrag generiert wird. Weitergehende Erklärungen folgen später in diesem Kapitel.

```
<%@ Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<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>Index</title>
</head>
<body>
    <table>
        <tr>
```

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
            <th></th>
            <th>Id</th>
            <th>Name</th>
            <th>FirstName</th>
            <th>Address</th>
            <th>Email</th>
            <th>Birthdate</th>
        </tr>
    <% foreach (var item in Model) { %>
        <tr>
            <td>
                <%: Html.AktionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) %> |
                <%: Html.AktionLink("Details", "Details", new { /* id=item.PrimaryKey */ })%> |
                <%: Html.AktionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })%>
            </td>
            <td><%: item.Id %></td>
            <td><%: item.Name %></td>
            <td><%: item.FirstName %></td>
            <td><%: item.Address %></td>
            <td><%: item.Email %></td>
            <td><%: String.Format("{0:g}", item.Birthdate) %></td>
        </tr>
    <% } %>
    </table>
    <p><%: Html.AktionLink("Create New", "Create") %></p>
</body>
</html>
```

Listing 13.6 Generierte View für die Aktion *Index()* des Beispiels

Benutzen der erstellten Funktionalität

Nach dem Erstellen der Funktionalität des ersten Entwicklungszyklus wollen wir uns natürlich auch ein erstes Erfolgserlebnis gönnen und den erstellten Code ausführen. Zu diesem Zweck führen wir die Webanwendung aus Visual Studio heraus zunächst einfach aus. Das Ergebnis davon ist zugegebenermaßen ernüchternd, denn wir erwarten nicht wirklich einen Fehler nach so viel generiertem Code!

Abbildung 13.7 Nach dem Start der Webanwendung aus Visual Studio zeigt der Browser einen Fehler

Der Grund für den Fehler liegt im falschen Aufruf für die erzeugte ASP.NET MVC-Anwendung. Bei einer solchen müssen wir einen Controller ansprechen und das macht Visual Studio nicht automatisch. Also ergänzen wir nun den aufgerufenen URL im Browser mit dem Namen des Controllers ohne Suffix, und versuchen es erneut.

Das Resultat in Abbildung 13.8 ist vielleicht von der Optik her nicht umwerfend, aber die erwarteten Daten sind vorhanden. Sogar die Möglichkeit, die dargestellten Daten zu verändern oder neue Daten zu erzeugen ist vorhanden, wenn auch die dahinterliegenden Verweise ins Nirwana führen. Wir sind ja schließlich auch noch nicht fertig mit der Anwendung, sondern haben nur einen ersten Entwicklungszyklus betrachtet.

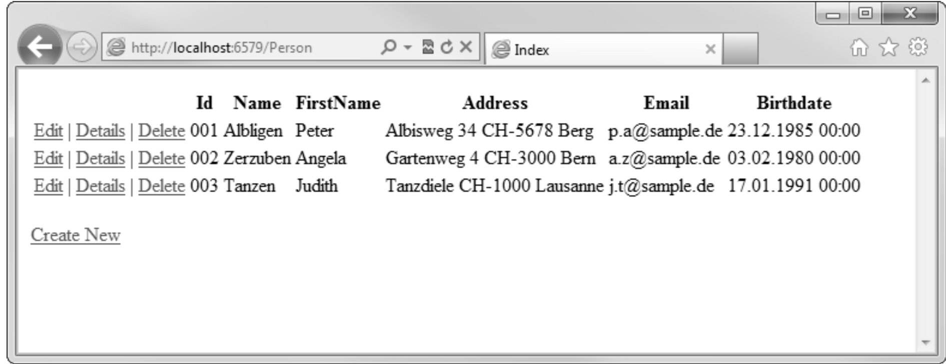

Abbildung 13.8 Generierte Liste der Personen

HINWEIS Es wäre durchaus möglich, die Anwendung so aufzubauen, dass auch bei der Verwendung des URLs ohne Angabe des Controllers ein definierter Controller angesprochen wird. Lesen Sie mehr dazu im folgenden Abschnitt.

Festlegen von Weiterleitungen

Ein sehr wichtiges Element ist das Festlegen von Weiterleitungen (engl. routing) in der Datei *global.asax*. Mit den dort untergebrachten Definitionen ist es für ASP.NET MVC überhaupt erst möglich zu erkennen, welche Funktionalität angesprochen werden soll. Die dazu notwendige Methode registriert in der Weiterleitungstabelle der Anwendung mindestens eine oder mehrere Weiterleitungen.

Die Registrierung erfolgt am einfachsten mit der Verwendung der Methode `MapRoute()`, welche als C#-Erweiterung für die Klasse `RouteCollection` von der .NET-Klassenbibliothek zur Verfügung gestellt wird. Die Methode kennt 5 Überladungen. Die klassische Verwendung, wie wir sie bereits aus Listing 13.1 kennen, definiert neben einem Namen für die Weiterleitung das Muster der Parameter und die dazu passenden Standardwerte für das Routing für den Fall, dass keine oder nicht alle Parameter in der Aufforderung vorhanden sind.

Listing 13.7 zeigt die Zeile aus der Datei *global.asax* noch einmal. Aufgrund der obigen Beschreibung können wir die Zeile nun so interpretieren, dass eine Anforderung den Namen eines Controllers gefolgt vom Namen der gewünschten Aktion und einer Identifikation eines Objekts beinhalten kann. Die drei Angaben sind jeweils mit einem Schrägstrich zu trennen. Fehlen die Angaben, wird der Controller mit Namen *Home* angesprochen. Ist ein solcher Controller nicht vorhanden, wird eine fehlende Ressource gemeldet. Ist der Controller vorhanden, jedoch keine Aktion im URL definiert, wird die Aktion *Index()* aufgerufen.

```
routes.MapRoute(
    "Default",                          // Name der Weiterleitung
    "{controller}/{Aktion}/{id}",       // URL mit Parameter
    new {                               // Parameterstandardwerte
```

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
        controller = "Home",
        Aktion = "Index",
        id = UrlParameter.Optional });
```

Listing 13.7 Die generierte Standardweiterleitung in der Datei *global.asax*

HINWEIS Durch die Veränderung des Standardwerts von `controller="Home"` auf `controller="Person"` kann im vorher betrachteten ersten Entwicklungszyklus erreicht werden, dass das Beispiel funktioniert, ohne dass der URL näher spezifiziert wird.

Da die Weiterleitung eine Abstraktion der effektiv angesprochenen Ressource definiert, kann dieser Mechanismus auch ausgenutzt werden, wenn später in der Entwicklung Änderungen in der Namensgebung auftreten.

Nehmen wir zum Beispiel an, dass aufgrund einer Überarbeitung der Funktionalität, der Controller `PersonController` in `EmployeeController` umbenannt werden soll. In diesem Fall kann ein erster Schritt darin bestehen, dass eine zusätzliche Weiterleitung mit dem neuen Namen definiert wird, aber diese Weiterleitung selbst auf den alten Namen des Controllers »zeigt«. Das Resultat dieser Maßnahme ist, dass der neue Controller von außen sofort angesprochen werden kann, ohne dass der Controller selbst umbenannt wird.

Listing 13.8 zeigt die dazu notwendige Definition in der Datei *global.asax*. Beachten Sie dabei, dass spezifische Weiterleitungen (in diesem Fall der URL *Employee*) jeweils vor den generellen Weiterleitungen definiert werden müssen. Die Reihenfolge der Definition in der Weiterleitungstabelle ist somit relevant für das Auffinden der richtigen Weiterleitung.

```
routes.MapRoute(
    "Employee",                    // Name der Weiterleitung
    "Employee/{Aktion}",           // URL mit Parameter
    new                            // Parameterstandardwerte
    {   controller = "Person",
        Aktion = "Index",
        id = UrlParameter.Optional
    });

routes.MapRoute(
    "Default",                     // Name der Weiterleitung
    "{controller}/{Aktion}/{id}",  // URL mit Parameter
    new                            // Parameterstandardwerte
    {  controller = "Person",
        Aktion = "Index",
        id = UrlParameter.Optional
    });
```

Listing 13.8 Ergänzende Weiterleitung

Für weitergehende Informationen bezüglich Weiterleitungen empfehle ich das Studium des Namenraums `System.Web.Routing` und insbesondere der Klassen gemäß Tabelle 13.2.

Typ	Name	Beschreibung
	`HttpMethodConstrained`	Erlaubt die Definition, mit welcher HTTP-Methode eine Weiterleitung angewendet werden kann
	`Route`	Definiert eine Weiterleitung
	`RouteCollection`	Definiert eine typisierte Auflistung von Weiterleitungen ▶

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Typ	Name	Beschreibung
	RouteData	Definiert eine Klasse für die Zusammenfassung aller Informationen, die zu einer Weiterleitung gehören (Datentoken, Weiterleitung selbst, der Handler der die Weiterleitung bedient, Daten)
	RouteTable	Definiert die Tabelle der Weiterleitungen der Anwendung
	RouteValueDictionary	Typisierte Auflistung für die Handhabung von Schlüssel/Wert-Paaren, die eine Weiterleitung näher spezifizieren.

Tabelle 13.2 Typen für die Nutzung der URL-Weiterleitung

Den Aufbau des Modells fertigstellen

Die Bandbreite der Arbeiten im Modell des MVC-Entwurfsmusters ist sehr stark von der verwendeten Technik für die Bereitstellung der Daten abhängig. Auf der einen Seite können direkt die Entitätsklassen und der Datenkontext des Entity Frameworks verwendet werden, auf der anderen Seite müssen Sie für die Persistenz vielleicht andere, projektspezifische Verarbeitungen vornehmen.

Sie sind aber immer gut beraten, wenn Sie die Modellebene des MVC-Entwurfsmusters so organisieren, dass folgende Voraussetzungen erfüllt sind:

- Zugriff aus mehreren gleichzeitigen Verarbeitungen heraus soll ermöglicht werden
- Das Modell muss mindestens die gewünschten Operationen der Benutzeroberfläche abdecken. Das heißt, dass wenn die Benutzeroberfläche das Erfassen von Daten erlaubt, dies logischerweise auch im Modell als Operation zur Verfügung steht.
- Das Modell soll die Funktionalität kapseln
- Das Modell sollte isoliert getestet werden

In den Erklärungen dieses Kapitels verwenden wir als Modell eine Singleton-Implementierung, die im Grunde genommen der Verwendung einer Datenbank sehr nahe kommt. Die Funktionalität der implementierten Klasse berücksichtigt die obenstehenden Kriterien. Die bereitgestellten Methoden des Modells entsprechen weitgehend den CRUD-Operationen und können Listing 13.9 entnommen werden.

```
// Datenkontext für die Handhabung der Demodaten
public sealed class CwlbDataContext {

    // Liefert den aktuellen Datenkontext der Anwendung
    public static CwlbDataContext Current { get; private set; }

    // Liefert eine unveränderliche Liste mit Personen
    public ReadOnlyCollection<CwlbPerson> Persons { get; }

(Wiedergabe gekürzt)

    // Erfasst ein Personenobjekt in der Datenliste
    public void Create(CwlbPerson objPerson) { ... }

    // Liefert die Personendaten einer definierten Identifikation
    public CwlbPerson Details(string strId) { ... }
```

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
    // Verändert die Daten einer definierten Person
    public void Update(string strId, string strName, string strFirstName,
                       string strAddress, string strEmail, string strBirthdate) { ... }

    // Löscht eine Person aus der Liste.
    public void Delete(string strId) { ... }
}
```

Listing 13.9 Methoden der Modellkomponente des ASP.NET MVC-Entwurfsmusters

HINWEIS Die vollständige Implementierung des Modells entnehmen Sie dem Beispiel *LearningASPNET_08*.

Neben der Klasse `CwlbDataContext`, die die Hauptkomponente für das Modell darstellt, nutzen wir eine oder mehrere Datenklassen für die Handhabung der effektiven Nutzdaten. Selbstverständlich werden diese Datenklassen auch einen Einfluss auf die Funktionalität des Datenkontextes nehmen, denn die Klassen müssen unter Umständen einzeln verwaltet werden können.

Die Datenklasse unseres Beispiels definiert die Information einer Person und ist bereits aus dem Abschnitt »Ein erster Entwicklungszyklus« bekannt. Listing 13.2 zeigt diese Klasse mit ihren Eigenschaften.

Der Aufbau eines Controllers

Die Controllerklassen des MVC-Entwurfsmusters sind zuständig für die Verarbeitung der aufgerufenen Aktionen und die Erstellung einer Antwort dazu. Im Detail gehören zur Verarbeitung das Entgegennehmen der Eingabedaten, die Handhabung der Daten zusammen mit dem Modell, die Bereitstellung von Daten für die Views und die Erstellung der Views für die Antwort.

Controllerklassen werden im Projekt im Ordner *Controller* untergebracht. Für den syntaktischen Aufbau der Klassen gelten folgende Regeln:

- Der Klassenname trägt den Suffix `Controller`
- Die erzeugte Klasse erbt von der Basisklasse `System.Web.Mvc.Controller`
- Ein Controller hat mindestens eine Aktion. Er kann beliebig viele Aktionen besitzen
- Aktionen können überladen werden
- Aktionen können für die Nutzung einer HTTP-Verkehrsrichtung zugeordnet werden
- Aktionen liefern in der Regel ein Objekt der Klasse `ActionResult`

Wie ein Controller erzeugt wird, haben wir bereits im Abschnitt »Ein erster Entwicklungszyklus« in diesem Kapitel gesehen. Eine große Hilfe für die Erstellung eines Controllers bietet die einzige Option in Form des angebotenen Kontrollkästchens *Add Action methods for Create, Update, Delete and Details scenarios*. Mit dessen Hilfe können wir uns einen relativ großen Anteil an Grundlagenarbeit für die Erstellung der CRUD-Operationen sparen.

Listing 13.10 zeigt das Resultat eines vollständig generierten Controllers. Ich habe für Sie die generierten Methoden geordnet und mit einem speziellen Kommentar gruppiert, ansonsten aber den Code syntaktisch so belassen, wie er von Visual Studio generiert wurde. Die Erklärungen des generierten Codes erfolgen anschließend Schritt für Schritt.

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
public class PersonController : Controller {

    // Create Data
    // --------------------------------------------
    public AktionResult Create() {
        return View();
    }

    [HttpPost]
    public AktionResult Create(FormCollection collection) {
        try {
            // TODO: Hier muss die Logik für das Erstellen der Personendaten ergänzt werden
            return RedirectToAktion("Index");
        }
        catch { return View();}
    }

    // Read
    // --------------------------------------------
    public AktionResult Index() {
        return View();
    }

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

    // Update
    // --------------------------------------------
    public AktionResult Edit(int id) {
        return View();
    }

    [HttpPost]
    public AktionResult Edit(int id, FormCollection collection) {
        try {
            // TODO: Hier muss die Logik für das Verändern der Personendaten ergänzt werden
            return RedirectToAktion("Index");
        }
        catch { return View();}
    }

    // Delete
    // --------------------------------------------
    public AktionResult Delete(int id) {
        return View();
    }

    [HttpPost]
    public AktionResult Delete(int id, FormCollection collection) {
        try {
            // TODO: Hier muss die Logik für das Löschen von Personendaten ergänzt warden.
            return RedirectToAktion("Index");
        }
        catch { return View();}
    }
}
```

Listing 13.10 Vollversion des generierten Codes eines Controllers

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Die Basisklasse Controller

Die Basisklasse der generierten Klasse `PersonController` (siehe Listing 13.10) ist die Klasse `Controller`. Diese Klasse stellt wichtige Methoden und Eigenschaften für den Aufbau eines Controllers gemäß Tabelle 13.3 zur Verfügung. Die meisten Methoden sind in der Basisklasse geschützt, also nur in ihrer eigenen Klasse zugänglich. Interessant ist die Tatsache, dass die zentralen Typen der wohlbekannten Eigenschaften `HttpContext`, `Request`, `Response` usw. nicht die normalen Klassen, sondern so genannte Wrapperklassen definieren. Der Grund dieser Änderung ist die Möglichkeit, mit eigenen Klassen zu agieren, die dieselben Eigenschaften zur Verfügung stellen.

Member	Name	Beschreibung
	`HttpContext`	Liefert die HTTP-spezifischen Informationen der Anforderung. Das gelieferte Objekt ist vom Typ `HttpRequestWrapper` und entspricht in Bezug auf die enthaltenen Informationen in etwa der Klasse `HttpContext`.
	`ModelState`	Liefert das Verzeichnis der Statusinformationen des Modells. Diese Eigenschaft benutzen wir insbesondere bei der automatisierten Datenvalidierung. Mehr dazu lesen Sie im Abschnitt »Daten validieren« in diesem Kapitel.
	`Request`	Liefert die Informationen der aktuellen Anfrage. Das gelieferte Objekt ist vom Typ `HttpRequestWrapper` und entspricht in Bezug auf die enthaltenen Informationen der Klasse `HttpRequest`.
	`Response`	Liefert die Informationen der aktuellen Antwort. Das gelieferte Objekt ist vom Typ `HttpResponseWrapper` und entspricht in Bezug auf die enthaltenen Informationen der Klasse `HttpResponse`.
	`RouteData`	Liefert die Weiterleitungsinformationen der aktuellen Anfrage. Dieses Objekt enthält auch das originale Weiterleitungsobjekt der Klasse `Route`.
	`Server`	Liefert die Informationen der aktuellen Anfrage. Das gelieferte Objekt ist vom Typ `HttpServerUtilityWrapper` und entspricht in Bezug auf die enthaltenen Informationen der Klasse `HttpServerUtility`.
	`Session`	Liefert die Informationen der aktuellen Anfrage. Das gelieferte Objekt ist vom Typ `HttpSessionWrapper` und entspricht in Bezug auf die enthaltenen Informationen der Klasse `HttpSessionState`.
	`TempData`	Liefert oder definiert ein Verzeichnis, das für die serverseitige Übernahme von Daten zur nächsten Anforderung benutzt werden kann
	`User`	Liefert die Sicherheitsinformationen der aktuellen Anfrage. Diese Eigenschaft benutzt den Typ `IPrincipal`.
	`ValidateRequest`	Liefert oder definiert `true`, wenn die Validierung der Anfrage eingeschaltet ist. Dieser Wert ist in ASP.NET 4 standardmäßig eingeschaltet. Beachten Sie zum Ausschalten unbedingt die Onlinehilfe.
	`ViewData`	Liefert oder definiert das Verzeichnis der Daten der View
	`File()`	Erstellt ein Resultatobjekt der Klasse `FileContentResult`
	`JavaScript()`	Erstellt ein Resultatobjekt der Klasse `JavaScriptResult`
	`Json()`	Erstellt ein Resultatobjekt der Klasse `JsonResult` ▶

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Member	Name	Beschreibung
	`Redirect()`	Erstellt ein Resultatobjekt der Klasse `RedirectResult`
	`RedirectToAction()`	Leitet die Verarbeitung zu der definierten Aktion um
	`RedirectToRoute()`	Leitet die Verarbeitung zu der definierten Route um
	`UpdateModel<T>()`, `TryUpdateModel<T>()`	Aktualisiert die angegebene Modellinstanz mit Werten vom aktuellen Wertanbieter des Controllers. Die Methode `TryUpdateModel()` arbeitet wie die Methode `UpdateModel()` mit dem Unterschied, dass sie keine Ausnahme auslöst, wenn das Model in einem ungültigen Zustand ist.
	`PartialView()`	Erstellt ein Resultatobjekt der Klasse `PartialViewResult`
	`View()`	Erstellt ein Resultatobjekt

Tabelle 13.3 Wichtigste Eigenschaften und Methoden der Klasse `Controller`

Der vollständig generierte Code, wie er in Listing 13.10 gezeigt wird, dient uns nun als Ausgangslage für die detaillierten Erklärungen der Codierungen der einzelnen CRUD-Operationen. Dabei werden wir immer wieder auch auf das Modell schauen und uns sogar partiell mit den Views auseinandersetzen müssen.

Eine Input-Output-Aktion bereitstellen

Bereits der erste Buchstabe von CRUD, das Create, führt uns zu mehreren, sehr interessanten Betrachtungen des Controllers. Für die Erfassung einer Personeninformation (Methode `Create()`) sieht der natürliche Arbeitsablauf vor, dass der Browser zuerst eine Anfrage für ein leeres Formaler zur Informationserfassung startet. Nach der Erfassung der Daten wird das ausgefüllte Formular an den Server zurückgesendet, um dort die Daten verarbeiten zu lassen.

Die Verarbeitung der Erfassung einer Person bedarf technisch gesehen eines HTTP-GET für die Anforderung des leeren Formulars und eines HTTP-POST für das Zurücksenden des ausgefüllten Formulars an den Server. Die Verarbeitung der beiden Anfragen werden vom Controller des ASP.NET MVC-Entwurfsmusters mit den Aktionen `Create()` bearbeitet. Die Mehrzahl ist hier korrekt angewendet, denn der Controller stellt für beide HTTP-Verkehrsrichtungen je eine Aktion `Create()` zur Verfügung. Da die Aktionen normale Methoden sind, müssen diese zum einen den C#-Regeln entsprechen, und zum anderen von ASP.NET MVC für die HTTP-Verkehrsrichtung erkannt werden.

Die C#-Regeln werden erfüllt, indem die beiden Aktionen über eindeutige Signaturen verfügen. Dass im Fall der Aktionen `Create()` je eine Aktion mit und ohne Parameter versehen ist, hat mit HTTP-GET und HTTP-POST nicht direkt etwas zu tun. Wir werden später sehen, dass es durchaus auch Methoden für die Bearbeitung von HTTP-GETs gibt, die Parameter aufweisen.

ASP.NET MVC erkennt die Verkehrsrichtung an den Attributen, die den Aktionen zugewiesen werden. In Listing 13.11 hat eine Aktion kein Attribut, was dem Standardwert HTTP-GET gleichgestellt ist. Die andere Aktion verfügt explizit über das selbstredende Attribut `HttpPost`.

```
public class PersonController : Controller {

    // Liefert das leere Formular für die Erfassung von Personendaten
    public AktionResult Create() {
        return View();
    }
```

```
    // Verarbeitet ein ausgefülltes Formular, das an den Server gesendet wird
    [HttpPost]
    public AktionResult Create(FormCollection collection) {
        try {
            // TODO: Hier muss die Logik für das Erstellen der Personendaten ergänzt werden
            return RedirectToAktion("Index");
        }
        catch {
           return View();
        }
    }
    ...
```

Listing 13.11 Die beiden `Create()`-Methoden für HTTP-GET (ohne Attribut) und HTTP-POST

Für die Dekoration der Methoden eines Controllers stellt die .NET-Bibliothek die Attribute gemäß Tabelle 13.4 im Namensraum `System.Web.Mvc` zur Verfügung. Beachten Sie dabei, dass Methoden einer Controllerklasse ohne Attribute grundsätzlich als Aktionen betrachtet werden und, solange sie öffentlich sind, auch von außen angesprochen werden können.

Typ	Name	Beschreibung
	`HttpGetAttribute`	Definiert, dass die Aktion Anforderungen des Typs HTTP-GET verarbeitet
	`HttpPostAttribute`	Definiert, dass die Aktion Anforderungen des Typs HTTP-POST verarbeitet
	`HttpPutAttribute`	Definiert, dass die Aktion Anforderungen des Typs HTTP-PUT verarbeitet
	`HttpDeleteAttribute`	Definiert, dass die Aktion Anforderungen des Typs HTTP-DELETE verarbeitet
	`AcceptVerbsAttribute`	Ermöglicht die explizite Definition der Verben des HTTP-Protokolls, die eine Aktion verarbeiten. Mithilfe dieses Attributs können Sie mehrere Verben des HTTP-Protokolls gleichzeitig definieren.
	`NonActionAttribute`	Definiert eine Methode eines Controllers als normale Methode und nicht als Aktion. Alternativ zur Dekoration einer Methode mit diesem Attribut, können Sie die Methode auch mit der Sichtbarkeit `internal` modifizieren. Nicht öffentliche Methoden sind von außen (vom Browser her) nicht aufrufbar.
	`ActionNameAttribute`	Erlaubt, den C#-Namen einer Aktion zu überschreiben. Benutzen Sie dieses Attribut, wenn der aufgerufene Aktionsname im URL vom effektiv implementierten Namen abweichen soll.

Tabelle 13.4 Attributklassen für die Dekoration von Aktionen in einem Controller

Wie wir gerade gelernt haben, agieren die Controller des MVC-Entwurfsmusters als eigentliche Drehscheibe für die Handhabung von Daten. Dabei werden Anforderungen im Controller jeweils in einer Aktion verarbeitet. Die Aktion bildet also sozusagen den Uraltprozess EVA (Eingabe, Verarbeitung, Ausgabe) der Datenverarbeitung ab. Diese Beschreibung führt uns zu den Betrachtungen der Ein- und Ausgabeparameter der Verarbeitung.

Die Eingabeparameter einer Aktion

Eine Aktion wird üblicherweise mittels HTTP-GET oder HTTP-POST angesprochen. Im ersten Fall können ein oder mehrere Parameter Verwendung finden. Die Parameter sind im Aufbau der Weiterleitung definiert.

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Einfache Parameter für HTTP-GET

Die bisher verwendete Weiterleitung (Listing 13.12) definiert einen URL mit je einer Angabe für Controller und Aktion. Für den Fall der Nichtdeklaration dieser beiden Werte im URL, sind im dritten Parameter der Registrierung der Weiterleitung die beiden Standardwerte `Person` und `Index` definiert. Die dritte Angabe im URL definiert die `id`, die wiederum explizit keinen Standardwert besitzt, jedoch als optional gekennzeichnet ist. Die Folge einer solchen Deklaration ist, dass Aktionen in den Controllern über diese Weiterleitung mit oder ohne Parameter aufgerufen werden können.

```
routes.MapRoute(
    "Default",                      // Name der Weiterleitung
    "{controller}/{Aktion}/{id}",   // URL mit Parameter
    new                             // Parameterstandardwerte
    {   controller = "Person",
        Aktion = "Index",
        id = UrlParameter.Optional
    });
```

Listing 13.12 Definition der Weiterleitung mit einem Parameter

Mit der in Listing 13.12 definierten Weiterleitung können somit für eine Webanwendung die URLs gemäß Tabelle 13.5 benutzt werden.

URL im Browser	Nutzung in ASP.NET MVC
http://<Basisadresse>/	Aktivierung des Controllers `PersonController` mit Aufruf der Aktion `Index()` oder `Index(Typ Id)`. In diesem Fall wird `Id` als `null`-Referenz übergeben.
http://<Basisadresse>/Person	Aktivierung des Controllers `PersonController` mit Aufruf der Aktion `Index()` oder `Index(Typ Id)`. In diesem Fall wird `Id` als `null`-Referenz übergeben.
http://<Basisadresse>/Person/Index	Aktivierung des Controllers `PersonController` mit Aufruf der Aktion `Index()` oder `Index(Typ Id)`. In diesem Fall wird `Id` als `null`-Referenz übergeben.
http://<Basisadresse>/Person/Index/10	Aktivierung des Controllers `PersonController` mit Aufruf der Aktion `Index(Typ Id)`. In diesem Fall wird `Id`, abhängig von `Typ` als Wert 10 übergeben (Zahl oder Zeichenfolge). Kann der Typ nicht konvertiert werden, wird eine Ausnahme ausgelöst.

Tabelle 13.5 Mögliche URLs und deren Wirkung auf den Aufruf der Aktion im Server

WICHTIG Tabelle 13.5 definiert die Verwendung der Methode `Index(Typ Id)`. Diese Notation funktioniert nur, wenn `Typ` ein Referenztyp ist, da bei Weglassung des entsprechenden Parameters vom MVC-Handler eine `null`-Referenz übergeben wird. Ferner wird der Parameter nur übernommen, wenn er nach `Typ` konvertierbar ist. Am einfachsten sind dabei Zeichenfolgen als Parameter zu übernehmen, denn diese liegen aus dem URL bereits als Zeichenfolge vor. Der Name des Parameters in der Aktion muss mit dem Namen des Parameters in der Weiterleitung übereinstimmen, wobei die Groß- und Kleinschreibung ignoriert wird.

Zusätzliche Parameter übergeben

Wie bei URLs für normale ASP.NET Webanwendungen können auch in ASP.NET MVC bei einem URL weitere Parameter angegeben werden. Dies hat auf die Definition der Weiterleitung in der Datei *global.asax* keine Auswirkung, sondern muss nur von der entsprechenden Aktion berücksichtigt werden, indem ein entsprechend benannter Parameter bereitgestellt wird.

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Nehmen wir zum Beispiel an, wir möchten die Indexmethode so beeinflussen können, dass über einen URL-Parameter die Sortierrichtung der Auflistung beeinflusst werden kann. Würde dazu der Parameter mit dem Kürzel `Sort` bezeichnet, ergäben sich die Notationen gemäß Tabelle 13.6.

Bei der Vermittlung von URL-Parametern ist der MVC-Handler mit der Typkonvertierung weniger streng. Er versucht zwar auch hier, den Typ über die Identifikation des Namens zuzuordnen und zu konvertieren. Ist das wegen eines falschen Namens oder Typs nicht möglich, wird den entsprechenden Parametern in der Aktion eine `null`-Referenz zugewiesen. Auch hier wird bei der Zuweisung die Groß- und Kleinschreibung ignoriert.

URL im Browser	Nutzung in ASP.NET MVC
http://<Basisadresse>/Person/Index/?sort=desc	Aktivierung des Controllers `PersonController` mit Aufruf der Aktion `Index(string sort)` oder `Index(Typ Id, string sort)`. In diesem Fall wird `Id` als `null`-Referenz übergeben.
http://<Basisadresse>/Person/Index/10?sort=desc	Aktivierung des Controllers `PersonController` mit Aufruf der Aktion `Index(Typ Id, string sort)`.

Tabelle 13.6 Mögliche URLs und deren Wirkung auf den Aufruf der Aktion im Server

Definieren einer speziellen Weiterleitung

Alternativ zur Verwendung von URL-Parametern auf der Standardweiterleitung, kann eine spezielle Weiterleitung definiert werden, die ohne URL-Parameter arbeitet. Das funktioniert aber nur, wenn keine Auslassungen im URL gemacht werden müssen, was bei hierarchischem Aufbau der Parameter (wie Controller, Aktion, Id) der Fall ist.

Listing 13.13 zeigt das Beispiel der Sortierung für die Aktion `Index()` im Controller `PersonController`. Die dabei verwendete Definition eines Standardwerts kann selbstverständlich auch bei der Variante mit URL-Parameter verwendet werden.

```
routes.MapRoute(
    "Index",                        // Name der Weiterleitung
    "Person/Index/{sort}",          // URL mit Parameter
    new                             // Parameterstandardwerte
    {   controller = "Person",
        Aktion = "Index",
        sort = "asc"
    });

routes.MapRoute(
    "Default",                      // Name der Weiterleitung
    "{controller}/{Aktion}/{id}",   // URL mit Parameter
    new                             // Parameterstandardwerte
    {   controller = "Person",
        Aktion = "Index",
        id = UrlParameter.Optional
    });
```

Listing 13.13 Spezialisierte Weiterleitung für die Aktion `Index()` mit Angabe der Sortierfolge und Standardwert

Parameter bei HTTP-POST

Wird von ASP.NET MVC eine Anfrage als HTTP-POST empfangen, ist die Verarbeitung der Weiterleitung grundsätzlich dieselbe. Da in dem Aufruf nun aber Formulardaten enthalten sind, werden diese vom Controller auch an die Aktion übergeben. Zu diesem Zweck sieht die Aktion einen Parameter des Typs `FormCollection` vor (siehe Listing 13.14).

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
public class PersonController : Controller {

    // Verarbeitet ein ausgefülltes Formular das an den Server gesendet wird
    [HttpPost]
    public AktionResult Create(FormCollection collection) {
        ...
    }
    ...
```

Listing 13.14 Die Methode Create() für HTTP-POST

Die Klasse `FormCollection` enthält die Daten des Formulars in Form von Schlüssel/Wert-Paaren. Der Inhalt der Auflistung kann mit den Eigenschaften und Methoden gemäß Tabelle 13.7 bearbeitet werden.

Member	Name	Beschreibung
	`AllKeys`	Liefert eine Auflistung aller Schlüssel im Objekt
	`Count`	Liefert die Anzahl Einträge im Objekt
	`[ ]`	Liefert den Wert eines bestimmten Schlüssels
	`Add()`, `Remove()`	Fügt ein Schlüssel/Wert-Paar zur Auflistung hinzu, respektive löscht ein Paar aus der Auflistung
	`Get()`, `Set()`	Liefert respektive definiert den Wert eines bestimmten Schlüssels
	`Clear()`	Löscht den Inhalt der Auflistung
	`HasKeys()`	Liefert `true`, wenn die Auflistung Schlüssel enthält, die nicht einer `null`-Referenz entsprechen

Tabelle 13.7 Wichtigste Eigenschaften und Methoden der Klasse `FormController`

Die soeben beschriebene Methode für die Entgegennahme von Eingabewerten aus der geposteten Anfrage hat den Vorteil, dass beliebig viele Schlüssel/Wert-Paare an die Aktion übergeben werden können. Die entsprechenden Werte müssen in der Aktion sodann noch ausgelesen und weiterverarbeitet werden.

Für den Fall, dass ein Formular eine festgelegte Anzahl Werte liefert, bietet die Klasse `Controller` die Möglichkeit, dass die in der Auflistung enthaltenen Werte direkt auf definierte Parameter der Aktion umgesetzt werden. Es gelten dabei dieselben Regeln wie bei den Parametern der HTTP-GET-Anfragen:

- Der Schlüsselname und der Parametername müssen gleich sein, wobei die Groß- und Kleinschreibung nicht berücksichtigt wird
- Die Daten werden auf den Typ des Parameters umgesetzt, wenn eine einfache Konvertierung möglich ist
- Ein Parameter der Aktion, der in den Formulardaten keine Entsprechung besitzt, wird als `null`-Referenz übergeben. Definieren Sie also Werttypen als nullfähig.

HINWEIS Bei der Erstellung einer View aufgrund eines Modelobjekts werden die Namen der generierten Steuerelemente standardmäßig aus den Namen der Eigenschaften der Datenklasse gewonnen. Daraus können wir folgern, dass die Schlüssel der empfangenen Daten beim HTTP-POST wiederum den Eigenschaftsnamen der Datenklassen entsprechen und somit sehr einfach direkt auf Parameter der Aktion umgesetzt werden können.

Das untenstehende Listing zeigt dazu ein Beispiel. Die dabei verwendete Kombination aus `FormCollection`-Parametern und direkt aufgelösten Schlüsseln deutet an, dass auch die Mischform verwendet werden darf.

```
public class PersonController : Controller {

    // Verarbeitet ein ausgefülltes Formular, das an den Server gesendet wird
    [HttpPost]
    public AktionResult Create(FormCollection collection, string id, string name, string firstName,
                                                         string address, string email, string birthdate) {
        ...
    }
    ...
```

Bei der Verwendung von streng typisierten Views können Sie auch ein Objekt des Typs der View direkt wieder in der entsprechenden Aktion entgegennehmen. Unser Beispiel würde dann wie folgt aussehen:

```
public class PersonController : Controller {

    // Verarbeitet ein ausgefülltes Formular, das an den Server gesendet wird
    [HttpPost]
    public AktionResult Create(CwlbPerson objPerson) {
        ...
    }
    ...
```

Mehr zum Erstellen von Views lesen Sie im nächsten Abschnitt.

Der Rückgabewert einer Aktion

Der Rückgabewert einer Aktion definiert das Resultat, das an den Browser zurückgesendet wird. Die Aktionen eines Controllers definieren als Rückgabe ein Objekt der Klasse `ActionResult`. Diese Klasse ist abstrakt und dient als Basis für verschiedene konkrete Typen, die in Tabelle 13.8 definiert sind.

Typ	Name	Beschreibung
	`ContentResult`	Stellt einen benutzerdefinierten Inhaltstyp dar
	`EmptyResult`	Stellt ein Ergebnis dar, das nichts bewirkt. Verwenden Sie diese Klasse bei einer Aktion, die eigentlich nichts zurückgibt.
	`FileContentResult`	Sendet einen binären Inhalt als Datei an die Antwort
	`FilePathResult`	Sendet den Inhalt der mit einer Pfadangabe definierten Datei an die Antwort
	`FileStreamResult`	Sendet den binären Inhalt des Streams an die Antwort
	`HttpUnauthorizedResult`	Stellt das Ergebnis einer nicht autorisierten HTTP-Anforderung dar
	`JavaScriptResult`	Definiert ein Resultat, das ein JavaScript an die Antwort sendet
	`JsonResult`	Definiert ein Resultat, das einen JSON-formatierten Inhalt an die Antwort sendet
	`RedirectResult`	Steuert die Verarbeitung von Anwendungsaktionen durch das Weiterleiten an einen angegebenen URL
	`RedirectToRouteResult`	Stellt ein Ergebnis dar, das mithilfe des angegebenen Wörterbuchs eine Weiterleitung ausführt
	`PartialViewResult`	Definiert ein Ergebnis, das einen Teil einer HTML-Seite an die Antwort sendet
	`ViewResult`	Definiert ein Ergebnis, das eine vollständige HTML-Seite, die durch die View-Engine von ASP.NET MVC generiert wird, an die Antwort sendet

Tabelle 13.8 Konkrete Rückgabetypen für Aktionen

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Meistens erzeugen wir die Objekte für die Rückgabe über eine der geerbten Methoden der Klasse `Controller` gemäß Tabelle 13.3. Dabei ist die wohl häufigste Verwendung die Methode `View()`. Diese Methode definiert mehrere Überladungen, bei der Sie sowohl Daten als auch die Namen der zu verwendenden View definieren können. Wird kein Viewname angegeben, wird automatisch nach der View gesucht, deren Name mit dem Namen der ausgeführten Aktion übereinstimmt.

Für das Auffinden der verwendeten View benutzt ASP.NET MVC folgende Prioritäten:

- Verzeichnis *Views/<ControllerName>/View.aspx*
- Verzeichnis *Views/Shared/View.aspx*

Verwenden von Aktionsfiltern

Die Controllerklassen von ASP.NET MVC erlauben die deklarative Form für die Definition von Codeausführungen vor und nach der Ausführung einer Aktion. Das dazu notwendige Mittel ist ein so genannter Aktionsfilter. Ein Aktionsfilter definiert technisch eine Attributklasse, die von der Basisklasse `ActionFilterAttribute` ableitet. Diese Basisklasse wiederum realisiert die Schnittstellen `IActionFilter` und `IResultFilter`.

Aktionsfilter können in zwei Fällen angewendet werden:

- Die Deklaration des Aktionsfilters direkt auf einer Aktion hat zur Folge, dass der Filter nur für die definierte Aktion gilt.
- Die Deklaration des Aktionsfilters auf der Controllerklasse hat zur Folge, dass der Filter für alle Aktionen des Controllers gilt.

Verwendung bestehender Aktionsfilter

Im Namensraum `System.Web.Mvc` sind konkrete Aktionsfilter und grundlegende Typen für die Aktionsfilter vordefiniert. Die konkreten Typen sind gebrauchsfertig und lassen sich beim Erstellen eines Controllers ohne weitere Vorkehrungen anwenden. Tabelle 13.9 zeigt die entsprechende Übersicht.

Typ	Name	Beschreibung
	`IActionFilter`	Definiert die Schnittstelle für die Implementierung eines Aktionsfilters, der die Ausführung der Aktion ummantelt
	`IResultFilter`	Definiert die Schnittstelle für die Implementierung eines Aktionsfilters, der die Erstellung des Resultats der Aktion ummantelt
	`FilterAttribute`	Definiert eine abstrakte Basisklasse für die Erstellung einer eigenen Aktionsfilterklasse. Wenn Sie diese Klasse als Basis verwenden, müssen Sie eine oder beide der obigen Schnittstellen realisieren. Diese Basisklasse stellt die Eigenschaft `Order` zur Verfügung. Mithilfe der Eigenschaft `Order` kann die Reihenfolge der Abarbeitung der Aktionsfilter auf einer bestimmten Aktion definiert werden. Das kann beim Einsatz von mehreren Aktionsfiltern auf ein und derselben Aktion unter Umständen wichtig sein.
	`ActionFilterAttribute`	Definiert eine abstrakte Basisklasse für die Erstellung eines eigenen Aktionsfilters. Diese Klasse verfügt über je eine virtuelle Implementierung der möglichen Methoden der beiden Schnittstellen dieser Tabelle.
	`AuthorizeAttribute`	Dieser Aktionsfilter erlaubt die benutzer- oder rollenbezogene Prüfung der Authentifizierung und Autorisierung für die Ausführung der betroffenen Aktion ▶

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Typ	Name	Beschreibung
	`ChildActionOnlyAttribute`	Dieses Aktionsfilter stellt sicher, dass die Aktion nur als untergeordnete Aktion ausgeführt wird. Eine untergeordnete Aktion rendert eine partielle View und nicht eine gesamte View. Lesen Sie mehr zu partiellen Views im Abschnitt »Herstellen von partiellen Views« in diesem Kapitel.
	`HandleErrorAttribute`	Stellt einen Aktionsfilter für die Behandlung von Fehlern bereit. Dieser Aktionsfilter kann mehrfach auf eine bestimmte Aktion angewendet werden. Jede Anwendung kann sich dabei auf einen anderen Ausnahmetyp beziehen.
	`RequireHttpsAttribute`	Dieser Aktionsfilter erzwingt eine Verbindung auf dem HTTPS-Protokoll
	`ValidateAntiForgeryTokenAttribute`	Mithilfe dieses Aktionfilters kann sichergestellt werden, dass eine manipulierte Anforderung an den Server erkannt wird
	`ValidateInputAttribute`	Markiert eine Aktion für die Prüfung der Eingabe

Tabelle 13.9 Vordefinierte Aktionsfilter im Namensraum `System.Web.Mvc`

Einen Aktionsfilter selbst herstellen

Die Herstellung eines eigenen Aktionsfilters geschieht am einfachsten über den bereits beschriebenen Weg der Vererbung von der abstrakten Basisklasse `ActionFilterAttribute`. Da die von den Schnittstellen her definierten Methoden in der Basisklasse bereits virtuell implementiert sind, können Sie hier die effektiv zu implementierenden Methoden selbst bestimmten.

Der Aktionsfilter in Listing 13.15 ermöglicht die Aufzeichnung der Ausführungen der Aktionen und der Resultaterstellung. Die Anwendung erfolgt auf der Aktion oder auf der Controllerklasse unter Angabe des Dateinamens. Der Aktionsfilter schreibt seine Daten in das Verzeichnis *App_Data* der Anwendung.

```
// Definiert einen Aktionsfilter für die Erstellung eines Logeintrags
public class LogingActionFilterAttribute : ActionFilterAttribute {

    // Liefert oder definiert den Dateinamen der verwendeten Logdatei
    public string LogFileName { get; set; }

    // Liefert den vollqualifizierten Dateinamen für den Logeintrag
    private string GetFullName(HttpContextBase objContext) {
        return Path.Combine(objContext.Request.PhysicalApplicationPath, "App_Data", LogFileName);
    }

    // Schreibt den Logeintrag in die Datei
    private void WriteLogEntry(HttpContextBase objContext,
                               string strController, string strAction, string strText)
    {
        try {
            File.AppendAllText(GetFullName(objContext), string.Format(
                "{0}; {1}; {2}; {3}{4}",
                DateTime.UtcNow.ToLongTimeString(),
                strController, strAction, strText,
                Environment.NewLine));
        }
```

```
        catch {
            // Alle Fehler ignorieren
        }
    }

    // Ereignisbehandlung vor der Ausführung der Aktion
    public override void OnActionExecuting(ActionExecutingContext filterContext) {
        WriteLogEntry(filterContext.HttpContext,
            filterContext.ActionDescriptor.ControllerDescriptor.ControllerType.Name,
            filterContext.ActionDescriptor.ActionName,
            "Executing");
        base.OnActionExecuting(filterContext);
    }

    // Ereignisbehandlung nach der Ausführung der Aktion
    public override void OnActionExecuted(ActionExecutedContext filterContext) { ... }

    // Ereignisbehandlung vor der Bereitstellung des Resultats
    public override void OnResultExecuting(ResultExecutingContext filterContext) { ... }

    // Ereignisbehandlung nach der Bereitstellung des Resultats
    public override void OnResultExecuted(ResultExecutedContext filterContext) { ... }
}
```

Listing 13.15 Eigene Implementierung eines Aktionsfilters

Der Aufbau einer View

Im Abschnitt »Ein erster Entwicklungszyklus« haben wir mithilfe von Visual Studio auch eine erste View generiert. Dabei haben wir das Dialogfeld *Add View* des Assistenten bereits ein erstes Mal angetroffen. Im gleichen Rahmen, in dem wir den Controller später vervollständigt haben, wollen wir uns nun dem Aufbau der restlichen Views mithilfe des Dialogfelds widmen. Nach der Generierung der Gerüste für die Views betrachten wir die Codierungsmöglichkeiten in einer Viewklasse näher.

Grundsatzentscheid: Masterseite oder nicht

ASP.NET MVC ändert gegenüber der althergebrachten Entwicklung von Webanwendungen mit ASP.NET viele Dinge. Unverändert wird dagegen die Möglichkeit des Einsatzes von Masterseiten übernommen. Einzig die Platzierung der Masterseiten erlebt in ASP.NET MVC eine Neudefinition.

Für gemeinsame Dateien der verschiedenen Views sieht die Struktur des ASP.NET MVC-Projekts den Ordner *Shared* innerhalb des Ordners *Views* vor. In diesen Ordner sollten Sie denn auch die Masterseite platzieren. Im Übrigen gelten aufgrund meiner Erfahrungen hinsichtlich der Verwendung von Masterseiten mit ASP.NET MVC die im zwölften Kapitel, Abschnitt »Das Konzept der Masterseiten«, beschriebenen Verhältnisse.

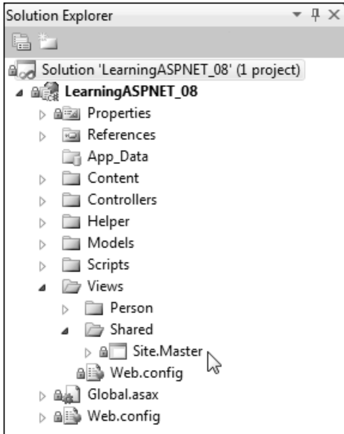

Abbildung 13.9 Platzierung der Masterseite(n) in der Struktur der ASP.NET MVC-Anwendung

Für Masterseiten gelten die gleichen Restriktionen wie für normale Views in ASP.NET MVC. Diesbezüglich soll noch einmal die Tatsache erwähnt werden, dass der Lebenszyklus einer Seite ein anderer ist, und dass aufgrund des fehlenden Postbacks die Verwendung von Webserversteuerelementen kaum möglich ist.

Generierung von Views

Für die Generierung der Views sehen wir zuerst pro Controllerklasse einen Ordner innerhalb des Ordners *View* vor. Der Name des Ordners muss dabei dem Namen des Controllers ohne Suffix `Controller` entsprechen. In unserem Beispiel definiert der Controller `PersonController` den Ordner *Person* im Ordner *Views* (siehe auch Abbildung 13.9).

Nach dieser Vorbereitung sind wir bereit, die benötigten Views zu erzeugen. Zu diesem Zweck starten wir den Assistenten mit dem Kontextmenübefehl *Add/View* auf dem Zielordner der gewünschten View (in unserem Beispiel *Person*). Das Dialogfeld *Add View* erlaubt nun die Definition der zu generierenden View. Verwenden Sie dazu die Eingaben, wie Sie in Tabelle 13.10 in Bezug auf die Abbildung 13.10 beschrieben sind.

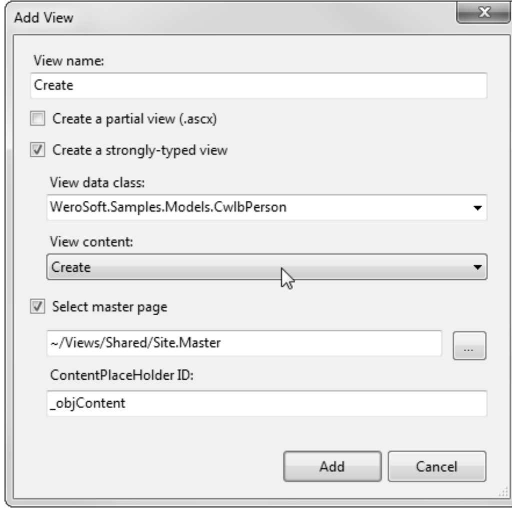

Abbildung 13.10 Dialogfeld für die Generierung einer View

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Feld	Bedeutung
View Name	Der Name der View wird für die Bildung des Dateinamens der View verwendet. Sofern der definierte Name mit dem Namen der Aktion des Controllers übereinstimmt, kann die View aus dem Controller ohne explizite Nennung erzeugt werden. Wird ein abweichender Name definiert, muss im Controller jeweils der Name der zu verwendenden View explizit angegeben werden.
Create a partial view	Wählen Sie diese Option aus, wenn Sie eine partielle View erstellen wollen. Eine partielle View ist eine unvollständige View, die nur innerhalb einer anderen View verschachtelt angewendet werden kann. Lesen Sie mehr zum Umgang mit partiellen Views im Abschnitt »Herstellen von partiellen Views«.
Create a strongly-typed view	Wählen Sie diese Option aus, wenn Sie eine streng typisierte View herstellen wollen. Eine streng typisierte View erlaubt den typsicheren Zugriff auf die Daten in der Inlinecodierung der View.
View data class	Wählen Sie hier die Klasse des Modells aus, für das Sie die View verwenden wollen. Diese Auswahl ist nur möglich, wenn Sie eine streng typisierte View herstellen.
View content	Wählen Sie den zu generierenden Inhalt der neuen View aus. Die hier zur Auswahl stehenden Inhalte entsprechen den CRUD-Operationen plus der Möglichkeit der Generierung einer leeren View. Die Optionen bedeuten: ▪ **`Create`** Erstellt ein Dialogfeld für die Eingaben der Eigenschaften gemäß der ausgewählten Klasse. Die Schaltfläche des generierten Dialogfelds ist mit *Create* beschriftet. ▪ **`Details`** Erstellt ein Dialogfeld für die schreibgeschützte Ausgabe aller Eigenschaften der ausgewählten Klasse. Die Schaltfläche des generierten Dialogfelds ist mit *Edit* beschriftet. ▪ **`Edit`** Erstellt ein Dialogfeld für die Veränderung der Eigenschaften gemäß der ausgewählten Klasse. Die Schaltfläche des generierten Dialogfelds ist mit *Save* beschriftet. ▪ **`Delete`** Erstellt ein Dialogfeld für die schreibgeschützte Ausgabe aller Eigenschaften der ausgewählten Klasse. Die Schaltfläche des generierten Dialogfelds ist mit *Delete* beschriftet. ▪ **`List`** Erstellt eine Liste aller Daten der ausgewählten Klasse. Jede Datenzeile wird mit den Aktionen Edit, Detail und Delete ergänzt. Die Schaltfläche des generierten Dialogfelds ist mit *Create New* beschriftet. ▪ **`Empty`** Erstellt eine leere View Diese Auswahl ist nur möglich, wenn Sie eine streng typisierte View herstellen.
Select Master Page	Wählen Sie das Kontrollkästchen für die Benutzung einer Masterseite und definieren Sie den Dateinamen der zu verwendenden Masterseite
ContentPlaceHolder ID:	Definieren Sie den Namen des Platzhalters der Masterseite, für den der Inhalt der zu generierenden Seite definiert wird

Tabelle 13.10 Bedeutung der Eingaben für das Dialogfeld *Add View*

Nach dem Bestätigen der Generierung mit der Schaltfläche *Add*, erstellt Visual Studio die entsprechende View. Das Resultat für die beschriebene View *Create.aspx* ist in Listing 13.16 abgebildet. Bei näherer Betrachtung fallen folgende Punkte besonders auf:

- Die Seite spezialisiert die Klasse `ViewPage<T>`
- Im Inlinecode wird wiederholt die Eigenschaft `Html` der Klasse `ViewPage` verwendet
- Die generierten Elemente sind in `<div>`-Tags gekapselt
- Die generierten Elemente weisen bereits Vorbereitungen für die Verwendung mit einem CSS auf

Diese Spezialitäten der Klasse `ViewPage`, wie die Eigenschaft `Html` und weitere Details für die Handhabung von Views, werden in den nächsten Abschnitten beleuchtet. Durch die Generierung aller Seiten für *Create*, *Delete*, *Details*, *Edit* und *List* wird die Anwendung bei korrekter Implementierung des Controllers und des Modells bereits benutzbar.

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
                  Inherits="System.Web.Mvc.ViewPage<WeroSoft.Samples.Models.CwlbPerson>" %>

<asp:Content ID="Content2" ContentPlaceHolderID="_objHeader" runat="server">
</asp:Content>

<asp:Content ID="Content1" ContentPlaceHolderID="_objContent" runat="server">
    <h2>Create</h2>
    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>
        <fieldset>
            <legend>Fields</legend>
            <div class="editor-label"><%: Html.LabelFor(model => model.Id) %></div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Id) %>
                <%: Html.ValidationMessageFor(model => model.Id) %>
            </div>
            <div class="editor-label"><%: Html.LabelFor(model => model.Name) %></div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Name) %>
                <%: Html.ValidationMessageFor(model => model.Name) %>
            </div>
            <div class="editor-label"><%: Html.LabelFor(model => model.FirstName) %></div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.FirstName) %>
                <%: Html.ValidationMessageFor(model => model.FirstName) %>
            </div>
            <div class="editor-label"><%: Html.LabelFor(model => model.Address) %></div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Address) %>
                <%: Html.ValidationMessageFor(model => model.Address) %>
            </div>
            <div class="editor-label"><%: Html.LabelFor(model => model.Email) %></div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Email) %>
                <%: Html.ValidationMessageFor(model => model.Email) %>
            </div>
            <div class="editor-label"><%: Html.LabelFor(model => model.Birthdate) %></div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Birthdate) %>
                <%: Html.ValidationMessageFor(model => model.Birthdate) %>
            </div>
            <p><input type="submit" value="Create" /></p>
        </fieldset>
    <% } %>
    <div><%: Html.ActionLink("Back to List", "Index") %></div>
</asp:Content>
```

Listing 13.16 Generierter HTML-Code der Inhaltsseite *Create.aspx*

Abbildung 13.11 zeigt die Optik der generierten View *Create.aspx* im Einsatz.

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Abbildung 13.11 Optik der generierten View *Create.aspx*

Die generierte View anpassen

Die Aufgabe der Viewklasse besteht darin, die vom Controller übergebenen Daten korrekt zu rendern, sodass die Antwort auf eine Anfrage mit den Daten der nächsten Anfrage übereinstimmt. Die Grundlage dazu liefert die Klasse `ViewPage<T>`, die für den Aufbau einer View verantwortlich ist. Die Klasse stellt neben den herkömmlichen Eigenschaften wie `Server`, `Request` und `Response` spezifische ASP.NET MVC-Eigenschaften zur Verfügung (siehe Tabelle 13.11). Der Typparameter `T` wird nur bei einer streng typisierten View definiert. Die streng typisierte Variante erbt von der gleichnamigen nicht streng typisierten Variante. Die Auswirkung der strengen Typisierung bezieht sich dabei auf das Ansprechen der vom Controller übergebenen Daten.

Member	Name	Beschreibung
	`Ajax`	Liefert ein Hilfsobjekt für die Programmierung in Szenarien mit Ajax-Einsatz
	`Application`	Liefert das Anwendungsstatusobjekt
	`Culture`	Liefert oder definiert die eingestellte Kultur der Seite
	`Html`	Liefert ein Hilfsobjekt für den programmatischen Umgang mit HTML-Elementen während des Renderns der Seite
	`Model`	Liefert eine Referenz auf das übergebene Objekt des Models. In der normalen Basisklasse `ViewPage` hat diese Eigenschaft den Typ `object`. In der abgeleiteten, streng typisierten Variante spiegelt die Eigenschaft den Typ des übergebenen Datenobjekts wider.
	`Request`	Liefert das aktuelle Anforderungsobjekt
	`Response`	Liefert das aktuelle Antwortobjekt
	`Server`	Liefert das Objekt für die Abfrage der serverseitigen Variablen und Verhältnisse
	`Session`	Liefert das aktuelle Sitzungsobjekt
	`User`	Liefert die Sicherheitsinformationen der aktuellen Anfrage
	`ViewData`	Liefert eine Auflistung aller Daten, die zwischen Controller und View übergeben werden

Tabelle 13.11 Wichtigste Eigenschaften der Klasse `ViewPage`

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Das Wesen der Viewklassen definiert die Inlinegenerierung der HTML-Elemente ohne die Verwendung der von ASP.NET üblicherweise bekannten Webserversteuerelemente. Um hier nicht einen allzu großen Rückschritt in Sachen Einfachheit vorzunehmen, stellt ASP.NET MVC für die Generierung die Hilfsklassen `HtmlHelper` und `HtmlHelper<T>` zur Verfügung. Mithilfe dieser Klassen ist es relativ einfach möglich, Daten mit ihren HTML-Elementen in die Antwort zu rendern.

Auch hier dient die streng typisierte Variante der Möglichkeit, das übergebene Objekt aus dem Model streng typisiert ansprechen zu können. Tabelle 13.12 zeigt die wichtigsten Eigenschaften und Methoden der Klasse. Die genannten Methoden dienen allesamt der Generierung von HTML-Elementen entsprechend dem Namen der jeweiligen Methode.

Member	Name	Beschreibung
	`ViewContext`	Liefert oder definiert die Kontextinformation der View
	`ViewData`	Liefert die aktuelle Auflistung der Daten in der View
	`ActionLink()`	Ruft die angegebene untergeordnete Aktion auf und gibt das Ergebnis als HTML-Zeichenfolge zurück
	`BeginForm()`, `EndForm()`	Definiert den Anfang respektive das Ende eines HTML-Formulars
	`CheckBox()`	Erstellt ein Kontrollkästchen
	`DropDownList()`	Erstellt eine Liste für die einfache Auswahl
	`Editor()`	Erstellt in Abhängigkeit des Datentyps oder des Attributs `UIHint` ein einzeiliges oder mehrzeiliges Eingabefeld
	`Hidden()`	Erstellt ein verstecktes Feld
	`Label()`	Erstellt eine Beschriftung
	`ListBox()`	Erstellt ein Element für die Mehrfachauswahl
	`Password()`	Erstellt ein Element für die Eingabe eines Passworts
	`RadioButton()`	Erstellt eine Optionsschaltfläche
	`RenderAction()`	Ruft die angegebene untergeordnete Aktionsmethode auf und rendert das Ergebnis inline in der übergeordneten Ansicht
	`RenderPartial()`	Rendert die angegebene Teilansicht mit dem angegebenen HMTL-Hilfsobjekt
	`RouteLink()`	Gibt einen Verweis (<a>-Tag) zurück, der den virtuellen Pfad der angegebenen Aktion enthält
	`TextArea()`	Erstellt ein mehrzeiliges Eingabefeld
	`TextBox()`	Erstellt ein einzeiliges Eingabefeld
	`Validate()`	Ruft die Validierungsmetadaten für das angegebene Modell ab und wendet die einzelnen Regeln auf das Datenfeld an
	`ValidationMessage()`	Zeigt eine Validierungsmeldung an, falls ein Fehler für das angegebene Feld vorliegt
	`ValidationSummary()`	Gibt eine ungeordnete Liste mit Validierungsmeldungen zurück

Tabelle 13.12 Wichtigste Eigenschaften der Klasse `HtmlHelper`

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Die in Tabelle 13.12 beschriebenen Methoden existieren zusätzlich in der Form `XxxxFor()` (Beispiel `TextBoxFor()`). Mithilfe dieser zusätzlichen Methoden kann mittels Lambda-Ausdruck direkt auf ein Modellelement zugegriffen werden.

WICHTIG Sie können den generierten HTML-Code in beliebiger, projektspezifischer Art anpassen. Sie müssen einzig darauf achten, dass der Controller mit den Daten, die er im Formular mittels HTTP-POST erhält, seine Arbeit korrekt ausführen kann.

Das Beispiel passt den generierten Code der View *Create.aspx* so an, dass die Identifikation der neu erfassten Person nicht mehr eingegeben wird, die Adresse in einem mehrzeiligen Feld liegt und das Geburtsdatum mittels jQuery UI-Datumsauswahl unterstützt wird. Zu diesem Zweck müssen folgende Arbeiten erledigt werden:

- Handhabung der Identifikation im Formular eliminieren
- Adresse in einem mehrzeiligen Feld umsetzen
- jQuery-UI in Projekt einbinden
- jQuery-UI in Seite *Create.aspx* einbinden und `Datepicker`-Steuerelement für Datum konfigurieren

Entnehmen Sie die Änderungen dem Listing 13.17.

HINWEIS Beachten Sie, dass Sie die JavaScript-Datei *jquery-ui-1.8.9.custom.min.js* nicht in Visual Studio finden. Sie müssen diese Datei direkt von der Internetseite für jQuery herunterladen. Das gilt ebenso für das verwendete Cascading Style Sheet *jquery-ui-1.8.9.custom.css*. Sie finden die entsprechenden Daten unter *http://jQueryui.com*.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
                  Inherits="System.Web.Mvc.ViewPage<WeroSoft.Samples.Models.CwlbPerson>" %>

<asp:Content ID="_objHead" ContentPlaceHolderID="_objHeader" runat="server">
    <link href="../../Content/jquery-ui-1.8.9.custom.css" rel="stylesheet" type="text/css" />
    <script src="../../Scripts/jquery-1.4.1.min.js" type="text/javascript"></script>
    <script src="../../Scripts/jquery-ui-1.8.9.custom.min.js" type="text/javascript"></script>
    <script type="text/javascript">
        $(function () {
            $('#Birthdate').datepicker(
             {
                 dateformat: 'dd.mm.yyyy',
                 showWeek: true
             });
        });
    </script>
</asp:Content>

<asp:Content ID="_objMain" ContentPlaceHolderID="_objContent" runat="server">

    <h2>Create</h2>

    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>

        <fieldset>
            <legend>Fields</legend>
            <div class="editor-label"><%: Html.LabelFor(model => model.Name) %></div>
 (Weidergabe gekürzt)
```

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
            <div class="editor-label">
                <%: Html.LabelFor(model => model.Address) %>
            </div>
            <div class="editor-field">
                <%: Html.TextAreaFor(model => model.Address, 5, 60, null) %>
                <%: Html.ValidationMessageFor(model => model.Address) %>
            </div>
 (Weidergabe gekürzt)

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Birthdate) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Birthdate)%>
                <%: Html.ValidationMessageFor(model => model.Birthdate) %>
            </div>

            <p><input type="submit" value="Create" /></p>
        </fieldset>
    <% } %>
    <div><%: Html.ActionLink("Back to List", "Index") %></div>
</asp:Content>
```

Listing 13.17 Angepasster HTML-Code für die korrekte Benutzung

Abbildung 13.12 Optik der Seite nach den Anpassungen

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Daten validieren

Selbstverständlich ist die Datenvalidierung auch in ASP.NET MVC ein sehr wichtiges Thema. Allerdings ist auch hier wieder das Problem vorhanden, dass aufgrund der Architektur von ASP.NET MVC die herkömmlichen Mechanismen von ASP.NET ausgehebelt wurden und entsprechend durch neue ersetzt werden mussten.

Der Lösungsansatz für die Validierung in ASP.NET MVC basiert auf der Deklaration von Validierungsregeln auf Stufe der Eigenschaften des Modells. Das erreichen wir mit dem Einsatz von neuen Attributen aus dem Namensraum `System.ComponentModel.DataAnnotations` gemäß Tabelle 13.13.

Typ	Name	Beschreibung
	`CustomValidationAttribute`	Erlaubt die Festlegung einer eigenen Validierung
	`DataTypeAttribute`	Erlaubt die Prüfung von Eingaben gegenüber einer vordefinierten Maske
	`RangeAttribute`	Erlaubt die Prüfung einer Eingabe gegenüber einem Bereich
	`RegularExpressionAttribute`	Erlaubt die Prüfung eines Inhalts gemäß den Regeln eines regulären Ausdrucks
	`RequiredAttribute`	Erlaubt die Prüfung einer Muss-Eingabe
	`StringLengthAttribute`	Erlaubt die Längenprüfung einer Zeichenfolge

Tabelle 13.13 Attribute für die Validierung von Daten

WICHTIG Die in Tabelle 13.13 vorgestellten Attributklassen erlauben den Zugriff auf die Ressourcen des Projekts, sodass die Lokalisierung der definierbaren Fehlermeldungen direkt mithilfe der Attribute vorgenommen werden kann. Beachten Sie dazu auch die Onlinehilfe.

Die Datenvalidierung beginnt mit der Deklaration der Validierungsregeln auf den Modellklassen. Das Beispiel in Listing 13.18 zeigt die Klasse `CwlbPersonen` nach der Einführung der Regeln. Achten Sie beim Festlegen der Regeln darauf, dass nicht erlaubte leere Eingaben explizit über die Regel `Required` geprüft werden. Das führt oft zur Mehrfachbelegung von Regeln, was aber selbstverständlich erlaubt ist.

```
// Repräsentiert die Daten einer Person
public class CwlbPerson {

    // Liefert oder definiert die Identifikation des Datenobjekts
    public string Id { get; set; }

    // Liefert oder definiert den Namen der Person
    [Required(AllowEmptyStrings=false, ErrorMessage="Der Name muss eingegeben werden.")]
    [StringLength(40, ErrorMessage="Der eingegebene Name ist zu lang.")]
    public string Name { get; set; }

    // Liefert oder definiert den Vornamen der Person
    public string FirstName { get; set; }

    // Liefert oder definiert die Adresse der Person
    public string Address { get; set; }

    // Liefert oder definiert die E-Mail-Adresse der Person
    [RegularExpression(@"…<regulärer Ausdruck für Prüfung der E-Mail-Adresse>…",
        ErrorMessage = "Die E-Mail-Adresse ist ungültig.")]
```

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
    [Required(AllowEmptyStrings = false, ErrorMessage = "Die E-Mail-Adresse muss eingegeben werden.")]
    public string Email { get; set; }

    // Liefert oder definiert das Geburtsdatum der Person
    [Range(typeof(DateTime), "1900.01.01", "2050.12.31",
          ErrorMessage="Das Datum muss zwischen 01.01.1900 und 13.12.2050 liegen.")]
    public DateTime Birthdate { get; set; }
}
```

Listing 13.18 Modellklasse nach der Einführung der Deklarationen für die Validierung der Daten

HINWEIS Bei der direkten Verwendung von Klassen von Entity Framework müssen die Prüfungen über eine spezielle Metadatenklasse eingebracht werden. Das ist notwendig, weil andernfalls das Dekorieren der Attribute in der Entitätsklasse vom Entity-Generator überschrieben würde. Mehr dazu lesen Sie in der Onlinehilfe von .NET. Beginnen Sie Ihre Suche hier über die Beschreibung des Typs `MetadataTypeAttribute`. Im Weiteren können Sie an dieser Stelle mit dem Attribut `System.ComponentModel.DisplayName` auch die lokalisierte Version des Bezeichners definieren. Allerdings ist zu bemerken, dass diese Art der Lokalisierung nur dann sinnvoll ist, wenn die Wiedergabe der Benutzeroberfläche in nur einer Sprache erfolgt. Sollten Sie eine dynamische Lokalisierung mit mehreren Sprachen benötigen, empfehle ich Ihnen die Ablage der Beschriftungen als separate Datenelemente.

Nach der Deklaration der Regeln muss der Code des Controllers bei den Eingangsaktionen (HTTP-POST) so verändert werden, dass die Regeln geprüft werden, und nur bei einwandfreiem Ergebnis die reguläre Operation durchgeführt wird, ansonsten sollen die Daten in der selben Form wieder als View gerendert und an den Browser zurückgesendet werden.

Die Validierung der Daten geschieht über die Eigenschaft `ModelState` des Controllers, indem auf dem gelieferten Objekt des Typs `ModelStateDictionary` die Eigenschaft `IsValid` abgefragt wird. Liefert diese Methode `true`, sind alle Regeln erfüllt und die gewünschte Operation kann durchgeführt werden. Liefert die Eigenschaft `false`, ist mindestens eine Regel nicht erfüllt und das Datenobjekt muss zur weiteren Bearbeitung an den Client zurückgesendet werden.

Listing 13.19 zeigt den dazu notwendigen Code für das Erfassen eines neuen Personenobjekts. Beachten Sie, dass der generierte Code der View bereits durch die Generierung für die Anzeige der Fehlermeldungen vorbereitet ist.

```
// Erstellt ein Personenobjekt nach der Erfassung durch den Benutzer
[HttpPost]
public ActionResult Create(CwlbPerson objPerson) {
    try {
        if (ModelState.IsValid) {
            CwlbDataContext.Current.Create(objPerson);
            return RedirectToAction("Index");
        } else {
            return View(objPerson);
        }
    }
    catch {...}
}
```

Listing 13.19 Code für die serverseitige Validierung der Daten

Abbildung 13.13 zeigt das Resultat der Datenvalidierung nachdem versucht wurde, eine neue Person ohne die Eingabe von Daten zu erfassen. Das gleiche Objekt wurde retourniert und die Fehlermeldungen werden angezeigt. Beachten Sie auch, dass die Abbildung nun komplett in Deutsch gehalten ist. Das habe ich zum

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

einen durch die direkte Übersetzung der generierten Literale in der *.aspx*-Datei und durch die Anwendung des Attributs `System.ComponentModel.DisplayName` in der Datenklasse erreicht (siehe auch vorangehender Hinweis).

Person erfassen

Daten
Name:
Der Name muss eingegeben werden.
Vorname:
Adresse:
E-Mail:
Die E-Mail-Adresse muss eingegeben werden.
Geburtstag:
Das Feld "Geburtstag:" ist erforderlich.
Speichern

Abbildung 13.13 Optik der Standarddarstellung von Fehlern

HINWEIS Alternativ zur Einzeldarstellung pro Eingabe, können Sie auch eine summarische Darstellung der Fehler erreichen. Löschen Sie zu diesem Zweck die einzelnen Fehlerausgaben der Felder und verändern Sie den Parameter der Zeile `Html.ValidationSummary()` von `true` auf `false`.

Sollten Sie eine Eingabe auf dem beschriebenen Weg nicht prüfen können, oder ist Ihnen die Erstellung der Prüfmethoden im Rahmen der Anwendung des Typs `CustomValidationAttribute` zu mühevoll, können Sie eine Prüfung im Controller auch jederzeit imperativ erstellen, indem Sie die Prüfung mit C# codieren und im Fall einer Regelverletzung der Fehlerauflistung über die Eigenschaft `ModelState` einen Fehler manuell hinzufügen.

Listing 13.20 zeigt dazu die Prüfung der Eingabe einer Adresse. Listing 13.18 sieht für diese Eigenschaft keine Prüfung vor.

```
// Erstellt ein Personenobjekt nach der Erfassung durch den Benutzer
[HttpPost]
public ActionResult Create(CwlbPerson objPerson) {
     try {
        // Imperative Prüfung von Daten
        if (string.IsNullOrWhiteSpace(objPerson.Address)) {
            ModelState.AddModelError("Address", "Geben Sie mindestens die Ortschaft ein.");
        }

        if (ModelState.IsValid) {
            CwlbDataContext.Current.Create(objPerson);
            return RedirectToAction("Index");
        } else {
            return View(objPerson);
        }
    }
    catch {...}
}
```

Listing 13.20 Imperative Prüfung einer eingegebenen Information

Herstellen von partiellen Views

Bis jetzt sind wir bei den Betrachtungen von ASP.NET MVC immer davon ausgegangen, dass ein Controller eine View als Gesamtes rendert. Der Begriff »Gesamt« ist hier im Sinn von einer selbstständigen HTML-Seite oder einer Inhaltsseite als Teil der Masterseite gemeint. In der Praxis bestehen gerenderte Seiten jedoch oft aus Einzelteilen, die möglichst ohne Redundanz zusammengestellt werden können sollten. Genau diese Anforderung erfüllt ASP.NET MVC mit der Möglichkeit der Herstellung von partiellen Views.

Bei der Generierung einer View über den Assistenten, kann durch das Wählen des Kontrollkästchens *Create a partial view* direkt eine partielle View erzeugt werden (siehe auch Abbildung 13.10 und die dazu gehörende Tabelle 13.10). Das Resultat einer solchen Generierung ist eine *.ascx*-Datei, also ein ASP.NET Benutzersteuerelement. Sollten Sie diesen Begriff hier das erste Mal hören, empfehle ich Ihnen an dieser Stelle das Studium des Abschnitts »Steuerelemente selbst erstellen« im Kapitel 12.

Aus Sicht des Controllers ändert sich bei der Verwendung von partiellen Views zunächst nichts. Sie müssen einzig die Übersicht darüber bewahren, welche Aktion des Controllers welche Daten an welche View weitergibt. Erzeugt eine Aktion eines Controllers in jedem Fall nur eine partielle View, können Sie die direkte Verwendung dieser Aktion durch einen Browser mit dem Aktionsfilter `ChildActionOnlyAttribute` verhindern. Das unterstreicht das Beispiel dieses Abschnitts, indem der erstellte Controller eine möglichst einfache Funktionalität darstellt (siehe Listing 13.21) und sowohl eine Aktion für die gesamte View der Demo (Aktion `Index()`) und zwei partielle Views (Aktionen `HitList()` und `BestOfList()`) zur Verfügung stellt. Beachten Sie, dass diese Aktionen ohne weitere Datenklassen auskommen!

```
// Interaktionslogik für die Demonstration von partiellen Views
public class DemoPartialController : Controller {
    /// Erstellt die Indexseite
    public ActionResult Index(){
        return View();
    }

    // Liefert eine Liste der besten Produkte
    [ChildActionOnly]
    public ActionResult BestOfList(){
        return View(new List<string>
        { "Der Schaumschläger",
          "Der Feldstecher",
          "Die Durchschlagende",
          "Der Erhebende",
          "Der Öffnende"
        });
    }

    // Liefert eine Liste der aktuell am meisten verkauften Produkte
    [ChildActionOnly]
    public ActionResult HitList(){
        return View(new List<string>
        { "Der Sommerhit",
          "Der Zaubernde",
          "Die gute Fee",
          "Das Märchenhafte"
        });
    }
}
```

Listing 13.21 Einfacher Controller mit zwei Unteraktionen

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

Eine partielle View muss letztendlich in eine View eingebunden werden, die eine Gesamtseite für den Browser rendert. In unserem aktuellen Beispiel übernimmt dies die View *Index.aspx*, die über die Aktion `Index()` des Controllers `DemoPartialController` erzeugt wird (siehe Listing 13.22). Innerhalb dieser View werden die beiden Aktionen `HitList()` und `BestOfList()` desselben Controllers aufgerufen. Die entsprechenden Aufrufe werden mit der bereits bekannten Methode `Action()` der Hilfsklasse `HtmlHelper` vorgenommen.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
nherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="_objHeadContent" ContentPlaceHolderID="_objHeader" runat="server">
</asp:Content>

<asp:Content ID="_objMainContent" ContentPlaceHolderID="_objContent" runat="server">
    <h2>
        Index</h2>
    <h3>
        Liste der aktuell meist verkauftenProdukte</h3>
    <div>
        <%: Html.Action("HitList", "DemoPartial") %>
    </div>
    <br />
    <br />
    <h3>
        Liste der besten Produkte</h3>
    <div>
        <%: Html.Action("BestOfList", "DemoPartial") %>
    </div>
</asp:Content>
```

Listing 13.22 Die Seite *View Index.aspx*, die zwei Unteraktionen aufruft und rendert

Die Inhalte der beiden Views *HitList.ascx* und *BestOfList.ascx* sind sehr einfach aufgebaut. Da lohnt sich der Aufwand der Nutzung des Assistenten kaum. Der Code der beiden Klassen rendert je eine Tabelle, die aus den Daten des Modells gewonnen werden können.

Da ich keine Modellklassen für dieses Beispiel verwende, sind auch die beiden Views in Listing 13.23 und Listing 13.24 nicht streng typisiert. Das erkennen Sie am Schlüsselwort `dynamic`, das jeweils als Parametertyp bei der Angabe der Basisklasse verwendet wird. Die vom Controller übergebenen Daten sind somit direkt über die `Model`-Eigenschaft der Klasse zugänglich. Da in beiden Fällen mit einer Liste des Typs `String` gearbeitet wird, kann die `foreach`-Anweisung die Daten problemlos verarbeiten.

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<dynamic>" %>
<table>
    <% foreach (string strHit in Model)
       { %>
    <tr>
        <td>
            <%: strHit %>
        </td>
    </tr>
    <% } %>
</table>
```

Listing 13.23 HTML-Code der partiellen View *HitList.ascx*

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<dynamic>" %>
<table>
    <% foreach (string strBestOf in Model)
       { %>
    <tr>
        <td>
            <%: strBestOf %>
        </td>
    </tr>
    <% } %>
</table>
```

Listing 13.24 HTML-Code der partiellen View *BestOfList.ascx*

Zur Vollständigkeit schauen wir uns noch Abbildung 13.14 an, die das erzeugte Resultat des Aufrufs der Aktion `Index()` für den Controller `DemoPartialController` zeigt.

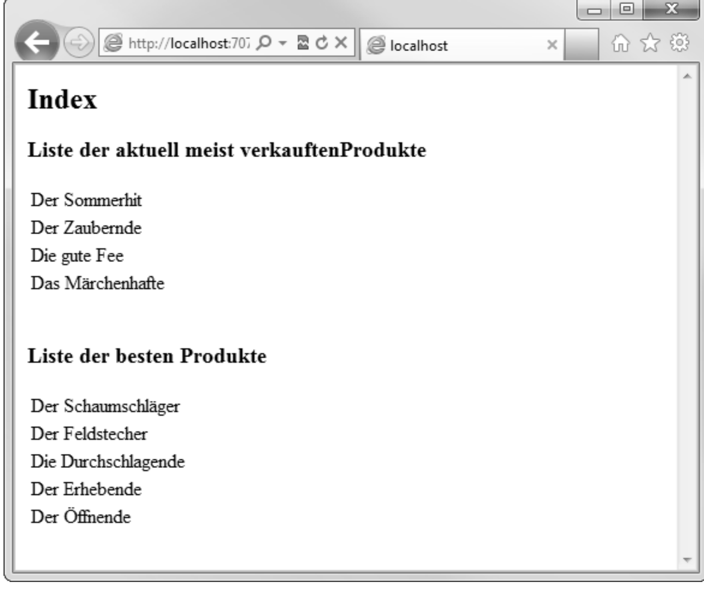

Abbildung 13.14 Resultat der Verwendung von zwei partiellen Views

Rolf Wenger: Handbuch der .NET 4.0-Programmierung, Band 2, Microsoft Press 2011
(ISBN 978-3-86645-439-2)