



Webanwendungen schreiben
mit ASP.NET MVC

Unter Aufsicht

Holger Schwichtenberg

Microsofts Entwicklungs-Framework für Webanwendungen – ASP.NET – existiert mittlerweile in verschiedenen Ausprägungen. Neben den ASP.NET Webforms etabliert sich zunehmend ASP.NET MVC, das mit einer besseren Kontrolle über HTML-Tags, guter Testbarkeit und einem modellgetriebenen Ansatz punktet.

Bereits bei Erscheinen des .NET Framework 1.0 gehörte ASP.NET Webforms zu den herausragenden Bestandteilen der Entwicklungsplattform von Microsoft. An vielen Stellen nivelliert Webforms die Eigenarten von HTTP, HTML, CSS und JavaScript. Leistungsfähige objektorientierte Serversteuerelemente generieren Tags und Skriptcode, etwa für Eingabemasken sowie Tabellen- und Kalenderansichten. Rapid Application Development für Websites war geboren.

In einer Welt mit vielen verschiedenen Betriebssystemen, Browsern und Bildschirmgrößen erweist sich solch ein abstrahierendes Framework, bei dem sich die einzelnen Tags nur schwerlich kontrollieren lassen, jedoch als hinderlich. Im Jahr 2009 hat Microsoft daher mit ASP.NET Model View Controller (ASP.NET MVC) ein hausinternes Konkurrenzprodukt zu den Webforms geschaffen. Hier gibt es keine Steuerelemente, der Entwickler arbeitet durchweg wieder auf der Ebene von HTML-Tags. Zudem erlaubt ASP.NET MVC eine bessere Trennung zwischen Bedienoberfläche und ihrer Steuerungsebene mit dem erklärten Ziel, das auto-

matisierte Testen von Anwendungen zu vereinfachen.

Eigenständiges Open-Source-Projekt

ASP.NET MVC basiert zwar auf dem .NET Framework, ist jedoch nicht Teil des .NET Framework Redistributable, sondern ein eigenständiges Open-Source-Projekt (alle relevanten Webinfos liegen unter „Alle Links“). Es unterstützt hauptsächlich C# und Visual Basic .NET. Andere .NET-Sprachen wie F# kann man

ebenfalls einsetzen, allerdings ist der Aufwand größer. Als Entwicklungsumgebung gibt es neben den kostenpflichtigen Varianten von Visual Studio ein kostenfreies Visual Studio Express for Web.

Derzeit liegt MVC in Version 5.1 vor. Laut BuiltWith.com hat sich das Web-Framework einen Platz unter den ersten zehn Konkurrenten erobert (Stand Dezember 2013). Mit 2,78 % Marktanteil rangiert es knapp hinter Ruby on Rails (3,01 %). ASP.NET Webforms liegt mit 22,64 % weiterhin auf Platz zwei deutlich hinter PHP (39,6 %). Die Zahlen beziehen sich auf die größten 10 000 Websites welt-



- ASP.NET MVC ist ein Open-Source-Webentwicklungs-Framework auf Basis von Microsoft .NET.
- Anders als bei ASP.NET Webforms hat der Webentwickler die volle Kontrolle über die auszugebenden HTML-Tags.
- Neben der expliziten Deklaration der Oberflächensteuerelemente unterstützt Microsoft einen modellgetriebenen Ansatz, der dem Entwickler jedoch einige Mühe beim Erstellen der entsprechenden Modelle und Vorlagen bereitet.

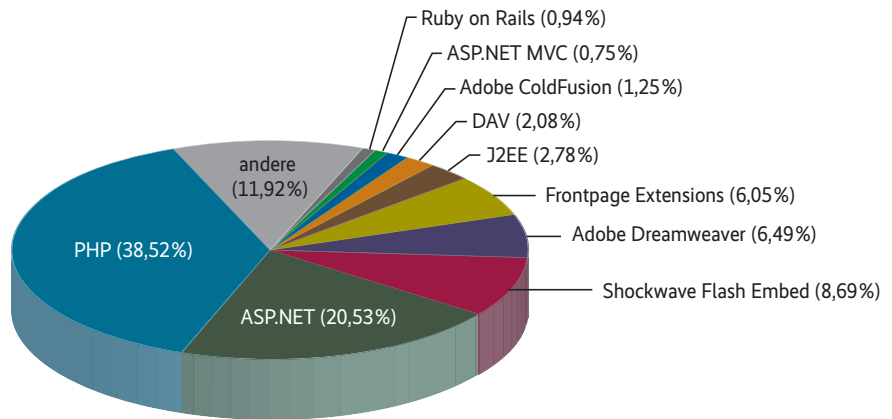
weit (Stand Dezember 2013). Legt man die erste Million der größten Websites zugrunde, kommt MVC nur auf 0,75 % Marktanteil (Abbildung 1). Dies ist der Tatsache geschuldet, dass man mit MVC in datenlastigen Anwendungen weit weniger produktiv arbeitet als mit Webforms. Seit einigen Jahren verliert Webforms jedoch Anteile zugunsten von MVC.

Während Entwickler in früheren .NET- und Visual-Studio-Versionen Webforms und MVC nicht gleichzeitig in einem Projekt einsetzen konnten, teilen beide seit .NET Framework 4.5.1 und Visual Studio 2013 eine gemeinsame technische Basis. Beim Anlegen eines Projekts muss sich der Entwickler allerdings immer noch entscheiden, ob er das in den Projektvorlagen enthaltene Grundgefüge seiner Website mit Webforms oder MVC gestalten will. Er kann jedoch das jeweils andere Framework später ebenfalls verwenden. Das Gerüst basiert seit Visual Studio 2013 auf dem CSS-Layout von Twitter Bootstrap 3.0 und umfasst neben ein paar statischen Seiten (*Index*, *About*, *Contact*) eine komplette datenbankbasierte Benutzerverwaltung inklusive Registrierungs- und Anmeldeseite. Alternativ ist eine Authentifizierung mit Identifikations-Providern wie Google, Twitter, Facebook und Microsoft möglich.

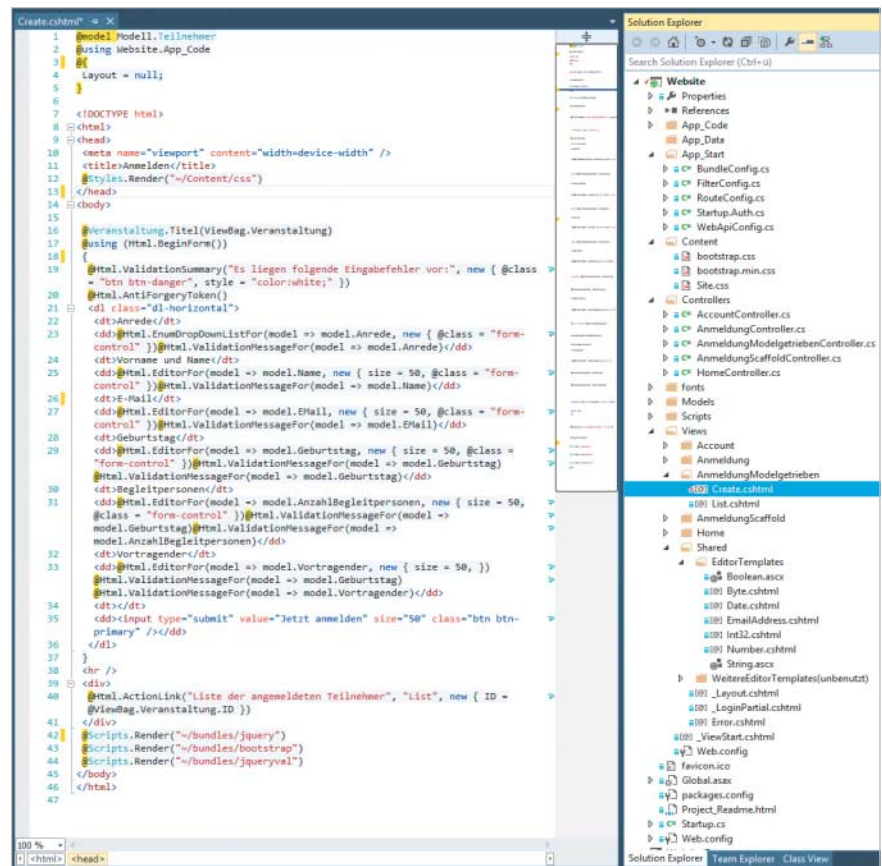
Klare Vorgaben zum Strukturieren

Das Grundgerüst macht mit den Ordnern *Models*, *Views* und *Controllers* klare Vorgaben zum Strukturieren der Dateien (Abbildung 2). Als Beispiel für das Programmieren mit MVC soll die Registrierung für eine Veranstaltung dienen (Abbildung 3). Eine Liste der angemeldeten Personen kann man einsehen (Abbildung 4).

Listing 1 zeigt das zugehörige Objektmodell, das aus den zwei Klassen *Veranstaltung* und *Teilnehmer* sowie einem Aufzählungstyp für die Anrede besteht. Listing 2 enthält die Minimalversion eines MVC-Controllers mit zwei Aktionen (*Create*, *List*). Für beide gibt es jeweils eine Methode, die einen Parameter *id* für die Veranstaltungsnummer aus dem Query String extrahiert. MVC bildet für die Aktionen automatisch URLs nach dem Prinzip *Controllername/Aktion/ID*. Per Konvention muss die Controller-Klasse mit dem Wort *Controller* enden. In der URL darf dieser Zusatz nicht auftauchen, sodass */Anmeldung/Create/123* zur Anmeldeseite für Veranstaltung 123 führt. Die klassische



BuiltWith.com zeigt die Verbreitung von Web-Frameworks unter den eine Million größten Websites weltweit (Abb. 1).



Rechts sieht man die ASP.NET-MVC-Projektstruktur in Visual Studio, links den Inhalt des View *Create.cshtml* zum Anlegen einer Person (Abb. 2).

Schreibweise */Anmeldung/Create?id=123* ist ebenfalls erlaubt.

Für die *Create()*-Methode gibt es noch eine zweite überladene Methode, die per *HTTP-POST* statt *HTTP-GET* gerufen wird, wenn der Benutzer das Formular abschickt. Sie empfängt ein Teilnehmerobjekt, das MVC automatisch mit den im Formular erfassten Daten bestückt, sofern die dortigen Feldnamen den Attributnamen der Teilnehmerklasse entsprechen. Ist diese Konvention nicht erfüllbar, kann der Entwickler die Bindung auf verschiedene Arten selbst definieren. Außerdem empfängt die Methode über ein Session-Objekt die Veranstaltungsnummer. Das Objekt ist das gleiche wie in Webforms.

Beide Aktionen nutzen für das Speichern der Daten Entity Framework [1]. Am Ende der Verarbeitung erzeugt die jeweilige Aktion ein View-Objekt und übergibt ihm ein Datenobjekt mit den anzuzeigenden Informationen. Die Wahl des aufzurufenden View erfolgt ebenfalls per Namenskonvention: Zur Aktion *List* in *AnmeldungController* gehört der View *Views/Anmeldung/List*. Alternativ kann der Entwickler dem View-Objekt einen anderen View-Namen übergeben. Für jede der beiden Aktionen gibt es also jeweils einen View (Listing 3 und 4). Den Gesamtaufbau des Beispiels zeigt Abbildung 5. In automatisierten Unit Tests lassen sich die Aktionen der Controller auf-

Anmeldung zur Veranstaltung

Expertenrunde ASP.NET MVC am 15.02.2014

Anrede:

Vorname und Name:

E-Mail:

Geburtsdag:

Begleitpersonen:

Vortragender: ☐

[Liste der angemeldeten Teilnehmer](#)

So sieht der unter <http://server/Anmeldung/Create/1> erreichbare View aus Abbildung 2 zur Laufzeit im Browser aus (Abb. 3).

Anmeldung zur Veranstaltung

Expertenrunde ASP.NET MVC am 15.02.2014

[Neuen Teilnehmer anmelden](#)

Anrede	Name	E-Mail
HerrDr	Holger Schwichtenberg	H.Schwichtenberg@IT-Visions.de
Herr	Manfred Steyer	M.Steyer@IT-Visions.de
Herr	Jörg Krause	J.Krause@IT-Visions.de
Herr	Christian Wenz	C.Wenz@IT-Visions.de

Diese Informationen zeigt der Browser, wenn man im Fallbeispiel den List-View unter <http://server/Anmeldung/List/1> aufruft (Abb. 4).

rufen und prüfen, ob er die erwarteten Views mit den passenden Daten zurückliefert. Ob die Views eine korrekte Darstellung erzeugen würden, lässt sich jedoch nicht auf dieser Ebene prüfen, sondern nur mit zusätzlichen Oberflächentests.

Mehr Komfort für den Benutzer

Zu Beginn eines View kann der Entwickler durch die Nennung einer .NET-Klasse den Typ der anzuzeigenden Daten bestimmen. So legt der View *List* nach *@model*

fest, dass er eine Liste von Teilnehmerobjekten verarbeitet, *Create* verwendet hingegen einen einzelnen Teilnehmer. Solche typisierten Views haben den Vorteil, dass Visual Studio eine IntelliSense-Eingabeunterstützung bieten kann, selbst wenn das Objekt noch Unterobjekte besitzt. Die Herausforderung: Es darf nur eine einzige Klasse für den View benannt werden. Oftmals, so auch in diesem Beispiel, benötigt man jedoch Daten aus verschiedenen unabhängigen Klassen. In diesem Fall muss der Entwickler entweder eine neue Klassendefinition eines sogenannten View-Modells schaffen, das die verschiedenen Objekte zusammenfasst, oder sich des dy-

namischen Objekts *ViewBag* bedienen, um mit Controller und View über die deklarierten Klassen hinaus beliebige Zusatzdaten austauschen zu können. Für ein dynamisches Objekt gibt es freilich keine Eingabeunterstützung. Das Beispiel zeigt die kürzere *ViewBag*-Variante.

Views bieten zwei Syntaxformen: Die klassische Schreibweise `<%=wert%>` verwendete Microsoft bereits Mitte der 90er-Jahre in den COM-basierten Active Server Pages. Die Dateinamenerweiterung ist dann *.aspx*. Prägnanter zeigt sich die in Listing 3 und 4 verwendete *Razor*-Syntax, die ein Codefragment mit „@“ einleitet und bei der die View Engine eigenständig erkennt, wo ein Befehl endet und HTML-Markup beginnt. Für Razor Views lautet die Dateinamenerweiterung *.cshtml* (C#) beziehungsweise *.vbhtml* (Visual Basic .NET). Anders als bei Webforms gibt es bei MVC leider keinen WYSIWYG-Designer für die Benutzeroberfläche; der Entwickler arbeitet nur auf der Ebene der Tags (Abbildung 2).

Es kann noch schöner werden

Innerhalb des View steht die Klasse *Html* zur Verfügung, die einige Hilfsmethoden zum Generieren von HTML anbietet, etwa *Html.ActionLink()* zum Erzeugen eines `<a>`-Tag zu einem Controller. Auch zum Erzeugen von Eingabefeldern existieren Hilfsmethoden wie *TextBox()*, *TextArea()*, *DropDownList()*, *CheckBox()* und *RadioButton()*. All diese Methoden gibt es zusätzlich in einer Variante mit angehängtem *For*, mit der man direkt über einen

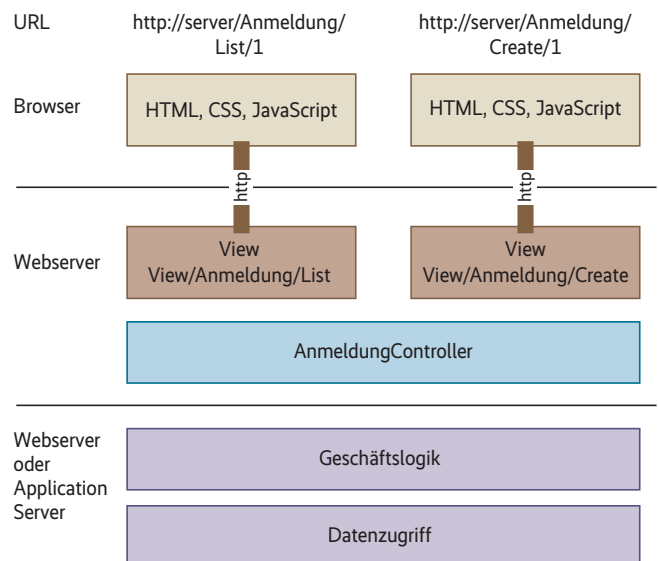
Listing 1: Modellklassen

```
using System;
using System.Collections.Generic;

namespace Modell
{
    public class Veranstaltung
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public DateTime Datum { get; set; }
        public virtual List<Teilnehmer> Teilnehmerliste { get; set; }
    }

    public enum Anrede
    {
        Herr, HerrDr, Frau, FrauDr
    }

    public class Teilnehmer
    {
        public int ID { get; set; }
        public Anrede Anrede { get; set; }
        public string Name { get; set; }
        public string EMail { get; set; }
        public Nullable<DateTime> Geburtsdag { get; set; }
        public byte AnzahlBegleitpersonen { get; set; }
        public bool Vortragender { get; set; }
        public virtual Veranstaltung Veranstaltung { get; set; }
    }
}
```



Im Fallbeispiel gibt es einen *AnmeldungController* mit den zwei Views *List* und *Create* (Abb. 5).

Listing 2: Controller-Klasse *AnmeldungController*

```
using Datenzugriff;
using Modell;
using System.Linq;
using System.Web.ModelBinding;
using System.Web.Mvc;

namespace Website.Controllers
{
    public class AnmeldungController : Controller
    {
        // Entity Framework-Kontext für Datenzugriff
        private EFKontext db = new EFKontext();

        // GET: /Anmeldung/List/ID
        public ActionResult List([QueryString] int id=1)
        {
            ViewBag.Veranstaltung = db.VeranstaltungsSet.Find(id);
            return View(db.TeilnehmerSet.ToList());
        }

        // GET: /Anmeldung/Create/ID
        public ActionResult Create([QueryString] int id)
        {
            ViewBag.Veranstaltung = db.VeranstaltungsSet.Find(id);
            Session["ID"] = id;
            return View(new Teilnehmer());
        }

        // POST: /Anmeldung/Create
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Create(Teilnehmer teilnehmer)
        {
            if (ModelState.IsValid)
            {
                db.TeilnehmerSet.Add(teilnehmer);
                db.SaveChanges();
                return RedirectToAction("List");
            }
            ViewBag.Veranstaltung = db.VeranstaltungsSet.Find(Session["ID"]);
            ViewBag.Status = "Unvollständige Eingaben!";
            return View(teilnehmer);
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                db.Dispose();
            }
            base.Dispose(disposing);
        }
    }
}
```

Lambda-Ausdruck Bezug auf eine Eigenschaft des deklarierten Modells nehmen kann. So könnte der Entwickler die Eingabefelder in Listing 4 auch wie folgt generieren:

```
@Html.TextBoxFor(model => model.Name,
    new { size = 50, @class = "form-control" })
```

Dabei entsteht für den zweiten Parameter ein anonymes .NET-Objekt mit zusätzlichen Attributen für das HTML-Tag. Das „@“-Zeichen vor *class* ist notwendig, da *class* nicht nur ein HTML-Attribut, sondern auch ein Schlüsselwort von C# ist.

Das Anwendungsbeispiel funktioniert, lässt jedoch noch Wünsche hinsichtlich Eingabekomfort und der Eingabevalidierung offen. So unterstützen längst nicht alle aktuellen Browser die in Listing 4 verwendeten HTML5-Eingabefeldtypen wie *Email*, *Number* und *Date*. Internet Explorer 10 und 11 zeigen bei *Number* und bei *Date* nur ein normales Texteingabefeld. Listing 4 verwendet zudem die CSS-Klasse *bg-danger*, die erst mit Bootstrap 3.1 eingeführt wurde. Wer den roten Hintergrund sehen will, muss ein mit Visual Studio 2013 angelegtes Projekt manuell von Bootstrap 3.0 auf Version 3.1 aktualisieren, indem er die Dateien *Bootstrap.css* und *Bootstrap.js* austauscht.

Neben der manuellen Validierung in der *POST*-Aktion unterstützt MVC eine elegantere modellbasierte Form. Dabei setzt Microsoft sogenannte Data Annotations ein, die sich in den Namensräumen *System.ComponentModel* und *System.ComponentModel.DataAnnotations* der .NET-Klassenbibliothek befinden. Sie können Prüfbedingungen wie *[Required]*, *[StringLength]*, *[Range]*, *[EmailAddress]* und *[RegularExpression]* oder die Darstellung festlegen, etwa *[Descrip-*

Listing 3: View */Anmeldung/List.cshtml*

```
@model IEnumerable<Modell.Teilnehmer>
<!DOCTYPE html>
<html><head></head><body>
<h4>Anmeldung zur Veranstaltung</h4>
<h3>@ViewBag.Veranstaltung.Name
    am @ViewBag.Veranstaltung.Datum.
    ToShortDateString()</h3>

<p>
    @Html.ActionLink("Neuen Teilnehmer
        anmelden", "Create",
        new { ID = @ViewBag.Veranstaltung.ID })
</p>

<table class="table table-striped">
<tr>
<th>Anrede</th>
<th>Name</th>
<th>E-Mail</th>
...
</tr>
@foreach (var item in Model)
{
<tr>
<td>@item.Anrede</td>
<td>@item.Name</td>
<td>@item.E-Mail</td>
...
</tr>
}
</table>
</body>
</html>
```

tion) und *[DisplayFormat]*. Die Angabe einer Fehlermeldung in der Data Annotation ist optional. Neben einem statischen Text kann der Entwickler hier auf eine XML-Ressourcendatei verweisen und so Mehrsprachigkeit umsetzen. Wenn er hier nichts erfasst, generiert die Data Annotation selbst einen Fehlertext. Listing 5 zeigt das mit Data Annotations angereicherte Objektmodell.

Schon auf dem Client prüfen

Der Server prüft die Data Annotations automatisch, wenn eine Controller-Aktion die Eigenschaft *ModelState.IsValid* abfragt. Allerdings fehlt dann noch die Ausgabe der Fehlermeldung auf dem Bildschirm. In Listing 3 erfolgte das manuell über *ViewBag.Status*. Im modellgetriebenen Ansatz geht es eleganter: Der Entwickler bestimmt für jedes Feld einzeln mit *@Html.ValidationMessageFor(model => model.Name)* und/oder alle Felder zusammenfassend mit *@Html.ValidationSummary* („Es liegen folgende Eingabefehler vor“) die Ausgabeposi-

tion der Fehlermeldungen aus den Data Annotations.

Das Nutzen von Metadaten geht noch weiter: Deklariert der Entwickler die Eingabefelder nicht selbst, sondern erzeugt sie mit Hilfsmethoden wie *TextBoxFor()*, nutzt MVC im Browser das jQuery-Plugin *jquery.validate.js* (in Verbindung mit der von Microsoft geschaffenen *jquery.validate.unobtrusive.js*), um viele Prüfbedingungen bereits clientseitig durchzusetzen. Die erzeugten HTML-Tags erhalten dazu *data-val*-Attribute:

```
<input data-val="true" 7
    data-val-email="Keine gültige E-Mail-Adresse" 7
    data-val-required="E-Mail ist Pflichtfeld!" 7
    id="EMail" name="EMail" placeholder="" 7
    required="required" size="50" 7
    type="email" value="" />
```

Nicht alles lässt sich auf dem Client auswerten. Für diese Fälle bietet MVC eine Remote Validation, für die man in der Controller-Klasse eine Methode schreibt, auf die im Modell mit der Annotation *[Remote]* zu verweisen ist. Um den Ajax-Aufruf der Methode kümmert sich dann MVC. Alle clientseitigen Prüfungen finden aus Sicherheitsgründen zusätzlich auf dem Server statt.

Noch einen großen Abstraktionsschritt weiter kommt man durch den Einsatz des allgemeinen *Html.EditorFor()* anstelle der konkreten Methoden *Html.TextBoxFor()* und *Co.EditorFor()* wählt anhand des deklarierten Datentyps und der Data Annotation ein geeignetes HTML-Steuerelement, etwa ein *CheckBox*-Element für eine *bool*-Eigenschaft. Diese Auswahl ist nicht fest verdrahtet, sondern der Entwickler kann eigene sogenannte Editor Templates im Ordner des View oder global unter *Views/Shared/EditorTemplate* hinterlegen. Editor Templates sind HTML-Fragmente (Dateiendungen *.ascx*, *.cshtml* oder *.vbhtml*) für ein Eingabefeld. Listing 6 zeigt ein solches für die Datumseingabe. Das Editor Template *Date.cshtml* kommt zum Einsatz, wenn eine Eigenschaft im Objekt den Datentyp *DateTime* besitzt und zusätzlich einschränkend mit *[DataType(DataType.Date)]* annotiert ist. Ohne diese zusätzliche Annotation wäre das Editor Template *DateTime.cshtml* zuständig. Zu beachten ist, dass die Vorlagennamen die CLR-Klassennamen und nicht die sprachspezifischen Datentypnamen verwenden müssen. Für eine mit *int* deklarierte Eigenschaft eines C#-Objekts benötigt der Entwickler also *int32.cshtml* und nicht

int.cshtml. Er kann jedoch auch frei Vorlagennamen vergeben und mit der Annotation *[UIHint(„Vorlagename“)]* auf sie verweisen.

Hilfestellung von Templates

Innerhalb der Vorlagen greift er über das *ViewData*-Objekt und seine Unterobjekte auf die Metadaten zu. *ViewData* selbst enthält auch die im anonymen Objekt übergebenen Zusatzdaten, die in Listing 6 als HTML-Attribute verwendet werden. Eine zweite Art dieser Vorlagen sind Display Templates, die man benutzt, wenn man einen Wert nur anzeigen will. Diese Vorlagen setzt man im View mit *Html.DisplayFor()* ein. Mit ihnen lässt sich der View *List* eleganter implementieren. Das Bezeichnungsfeld für eine Eigenschaft wird auf diese modellgetriebene Weise mit *Html.LabelFor()* erzeugt. Abbildung 2 zeigt auf der linken Seite den modellgetriebenen Ansatz für den View *Create.cshtml*. Das komplette Beispiel liegt auf dem FTP-Server der iX (siehe „Alle Links“).

Visual Studio hilft mit sogenannten Scaffolds, Vorlagen, die das Grundgerüst

von Views und/oder Controllern erzeugen, beim Anlegen von Controllern und Views. Wenn mit Entity Framework auf die Daten zugegriffen wird, bewirkt Scaffolding am meisten. Hier wählt man den „Scaffold MVC 5 Controller with views, using Entity Framework“. Daraus entstehen dann fünf metadatengetriebene Views (*Index*, *Details*, *Edit*, *Insert*, *Delete*), die *Html.LabelFor()*, *Html.DisplayFor()* und *Html.EditorFor()* sowie die Validierungsgeneratoren nutzen.

Der Entwickler darf eigene Data Annotations schreiben. Meist genügen dazu wenige Programmzeilen. Listing 7 zeigt exemplarisch eine neue Validierungsannotation *[Choice]*, die für ein Zeichenkettenfeld eine begrenzte Auswahl statischer Texte vorgibt. Solch eine Annotation berücksichtigt der Server automatisch. Damit sie auch clientseitig wirkt, muss der Entwickler eine entsprechende JavaScript-Routine selbst implementieren.

Wer Data Annotations für die Darstellung erzeugt, muss sie natürlich in den Templates auswerten. Jede Vorlage kann das für sich selbst erledigen. Alternativ schreibt man eine eigene Ableitung der Klasse *DataAnnotationsModelMetadataProvider*, extrahiert dort die relevanten Informationen und reicht sie automatisch

Listing 4: View /Anmeldung/Create.cshtml

```
@model Modell.Teilnehmer
<!DOCTYPE html>
<html><head></head><body>
<h4>Anmeldung zur Veranstaltung</h4>
<h3>@ViewBag.Veranstaltung.Name
    am @ViewBag.Veranstaltung.Datum.ToShortDateString()</h3>
@using (Html.BeginForm())
{
    if (!String.IsNullOrEmpty(ViewBag.Status))
    { <div class="bg-danger">@ViewBag.Status</div> }
    @Html.AntiForgeryToken()
    <dl class="dl-horizontal">
        <dt>Anrede</dt>
        <dd>
            <select id="Anrede" class="form-control" name="anrede">
                <option selected="selected" value="Herr">Herr</option>
                <option value="HerrDr">Herr Dr</option>
                <option value="Frau">Frau</option>
                <option value="FrauDr">Frau Dr</option>
            </select>
        </dd>
        <dt>Vorname und Name</dt>
        <dd><input id="Name" name="Name"
            type="text" size="50" class="form-control"
            value="@Model.Name" /></dd>
        <dt>E-Mail</dt>
        <dd><input id="EMail" name="EMail"
            type="EMail" size="50" class="form-control"
            value="@Model.EMail" /></dd>
        <dt>Geburtsstag</dt>
        <dd><input id="Geburtsstag" name="Geburtsstag"
            type="Date" size="50" class="form-control"
            value="@Model.Geburtsstag" /></dd>
        <dt></dt>
        <dd><input type="submit" value="Jetzt anmelden"
            size="50" class="btn btn-primary" /></dd>
    </dl>
    <div>@Html.ActionLink("Liste der angemeldeten Teilnehmer",
        "List", new { ID = @ViewBag.Veranstaltung.ID })</div>
</body>
</html>
```

Listing 5: Mit Data Annotations angereichertes Objektmodell

```
public enum Anrede
{
    Herr,
    [Description("Herr Dr")] HerrDr,
    Frau,
    [Description("Frau Dr")] FrauDr
}

public class Teilnehmer
{
    public int ID { get; set; }
    public Anrede Anrede { get; set; }
    [Required(ErrorMessage = "Name ist Pflichtfeld!")]
    [StringLength(50)]
    [RegularExpression(".* .*", ErrorMessage = "Eingabe muss mindestens aus Vorname, einem Leerzeichen und Nachname bestehen!")]
    public string Name { get; set; }
    [Required(ErrorMessage = "E-Mail ist Pflichtfeld!")]
    [EmailAddress(ErrorMessage = "Keine gültige E-Mail-Adresse")]
    public string EMail { get; set; }
    [DisplayFormat(ApplyFormatInEditMode = true,
        DataFormatString = "{0:dd MMM yyyy}")]
    [DataType(DataType.Date)]
    public Nullable<DateTime> Geburtsstag { get; set; }
    [Range(0,3)]
    [DefaultValue(0)]
    public byte AnzahlBegleitpersonen { get; set; }
    public bool Vortragender { get; set; }
    public virtual Veranstaltung Veranstaltung { get; set; }
}
```

Listing 6: Editor Template *Date.cshtml*

```
@{
    var attributes = ViewData;
    attributes.Add("type", "date");
    if (ViewData.ModelMetadata.IsRequired)
    {
        attributes.Add("required", "required");
    }
}
```


allen Templates über *modelMetadata.AdditionalValues* weiter.

Die Grundstruktur der metadatengetriebenen Entwicklung mit ASP.NET MVC ist flexibel, leider hat Microsoft jedoch in seiner Standardimplementierung vier gravierende Lücken gelassen:

- Für Aufzählungstypen zeigt MVC im Standard nur ein Eingabefeld statt eines Auswahlfeldes an.

- Die Templates werten längst nicht alle im .NET-Framework existierenden Annotationen aus.

- Microsoft hat in *ViewData.ModelMetadata* Eigenschaften wie *Watermark* vorgesehen, für die es allerdings keine Annotation gibt, mit der man sie setzen könnte.

- *Html.EditorFor()* erlaubt zwar genau wie *TextBoxFor()* die Übergabe zusätzlicher HTML-Attribute als anonyme Objekte, die Standard-Templates werten sie aber nicht aus.

Reiche Auswahl an Schablonen

Diese Schwächen muss der Entwickler selbst ausgleichen, indem er die Templates modifiziert oder Erweiterungen der *Html*-Klasse anlegt. Mit der Anpassungsarbeit für die Vorlagen erschwert Microsoft dem Entwickler das Leben unnötig, denn auf die derzeitigen Standard-Templates kann er nicht direkt zugreifen. In der Quellcodeverwaltung des MVC-Frameworks liegen sie nur in reiner C#-Form vor, nicht jedoch als *.ascx* oder *.cshtml*-Datei. In dieser Form gibt es nur eine ältere Version. Ein Blog liefert eine Erklärung.

Auf Nuget.org findet man Vorlagen, die HTML5-Eingabefelder oder Twitter Bootstrap nutzen. Im Beispiel auf dem FTP-Server der *iX* liegen mehrere angepasste Templates sowie Erweiterungen der *Html*-Klasse, etwa *Html.EnumDropDownListFor()*, die automatisch eine Auswahlliste für Aufzählungstypen erstellt und dabei die *[Description]*-Annotationen berücksichtigt, sodass der Benutzer *Frau Dr* statt *FrauDr* auswählen kann (alle weiterführenden Infos dazu unter „Alle Links“).

ASP.NET bietet darüber hinaus mehrere Funktionen zum Aufteilen großer Webseiten in kleinere Portionen und zur Wiederverwendung von Oberflächenbestandteilen. Via Masterpages oder Layoutseiten gibt der Webentwickler den Rahmen für andere Seiten vor. Mit den Hilfsroutinen *@Html.RenderBody()*, *@Html.RenderSection()* und *@Html.RenderPage()* fügt er die Bausteine dann zusammen.

Listing 7: Editor Template *Date.cshtml*

```
/// <summary>
/// Eigene Validierungsklasse
/// Auswahl muss aus einer vorgegebenen Menge von Zeichenketten kommen
/// </summary>
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property, AllowMultiple = false)]
public class ChoiceAttribute : ValidationAttribute
{
    public List<string> Choices { get; set; }
    public ChoiceAttribute(params string[] choices)
    {
        this.Choices = choices.ToList();
    }
    public override bool IsValid(object value)
    {
        return (Choices.Contains(value.ToString()));
    }
}
```

Listing 8: Razor-Helfer-Datei */App_Code/Veranstaltung.cshtml*

```
@helper Titel(Moell.Veranstaltung Veranstaltung)
{
    <h4>Anmeldung zur Veranstaltung</h4>
    <h3>@Veranstaltung.Name am @Veranstaltung.Datum.ToShortDateString()</h3>
}
```

Views kann man in Partial Views trennen, wobei die keinen eigenen Controller besitzen. Einzelne, wiederkehrende HTML-Fragmente lassen sich in Razor-Hilfsroutinen auslagern. Ein guter Kandidat dafür wäre die Ausgabe des Veranstaltungstitels. Eine im Unterverzeichnis */App_Code* liegende Helfer-Datei (Listing 8) lässt sich dann in allen Views mit *@Veranstaltung.Titel(ViewBag.Veranstaltung)* leicht einbinden. Umfangreiche Websites kann der Entwickler mithilfe von *Areas* in kleinere Bereiche trennen. Jeder hat einen eigenen Unterordner für Controller, Views und Models.

Das Laden von Stylesheet- und JavaScript-Dateien unterstützt MVC mit den Hilfsklassen *Styles* und *Scripts*. Die Methode *Render()* fügt entsprechende Tags in die HTML-Seite ein, die auf diese Dateien verweisen. Sofern man vorher in *AppStart/BundleConfig.cs* mehrere *.css*- oder *.js*-Dateien zu einem Bündel zusammengeschnürt hat, lassen sie sich auch zusammen laden. So holt *@Styles.Render(„~/Content/css“)* alle *.css*-Dateien aus dem Content-Ordner in einem Rutsch und auf das Wesentliche komprimiert, also ohne Leerzeichen und Umbrüche.

Schutzfunktionen: Nicht jeder darf alles

Im Bereich der Sicherheitsfunktionen bietet ASP.NET zum einen Schutz gegen Cross-Site-Scripting-Angriffe (*[ValidateAntiForgeryToken]*, Listing 2) sowie *@Html.AntiForgeryToken()* (Listing 3 und 4) und zum anderen Authentifizierungs- und Autorisierungsmechanismen. Der Zusatz *[Authorize(Roles=“Admin“)]* auf einer Aktionsmethode in einem Con-

troller sorgt dafür, dass nur authentifizierte Benutzer in der Administratoren-Rolle diese Aktion nutzen können. Nicht authentifizierte Anwender leitet MVC auf die Anmeldeseite um, die in der Standardprojektvorlage unter */App_Start/StartupAuth.cs* festgelegt ist.

ASP.NET MVC dürfte für eine große Zahl von Webentwicklern interessant sein, weil es ihnen einerseits volle Kontrolle über die HTML-Ausgabe bietet, andererseits mit dem modellgetriebenen Ansatz Potenzial für schnelles Entwickeln durch Standardisierung bietet. Allerdings müssen sie sich viele dieser Standards selbst erarbeiten. Microsoft bietet eine flexible Basisinfrastruktur an, die der Nutzer an etlichen Stellen selbst ausgestalten muss. Für eine Menge konkreter Fälle wären Metadatentypen und Vorlagen wünschenswert. Die herzustellen, erfordert allerdings ein tiefes Verständnis der zugrundeliegenden Technik. Hier dürfte der Grund dafür zu finden sein, dass zahlreiche MVC-Anwender lieber weiterhin klassisch und nicht metadatengetrieben arbeiten. (jd)

Dr. Holger Schwichtenberg

leitet das Expertennetzwerk www.IT-Visions.de, das Beratung, Schulungen und Softwareentwicklung im .NET-Umfeld anbietet. Er hält Vorträge auf Fachkonferenzen und ist Autor zahlreicher Fachbücher.

Literatur

- [1] Holger Schwichtenberg; Datenbanktechnik; Klar abbilden; Objekt-Relationales Mapping mit Microsofts Entity Framework 6.0; *iX* 3/2014, Seite 138

Alle Links: www.ix.de/ix1404128

