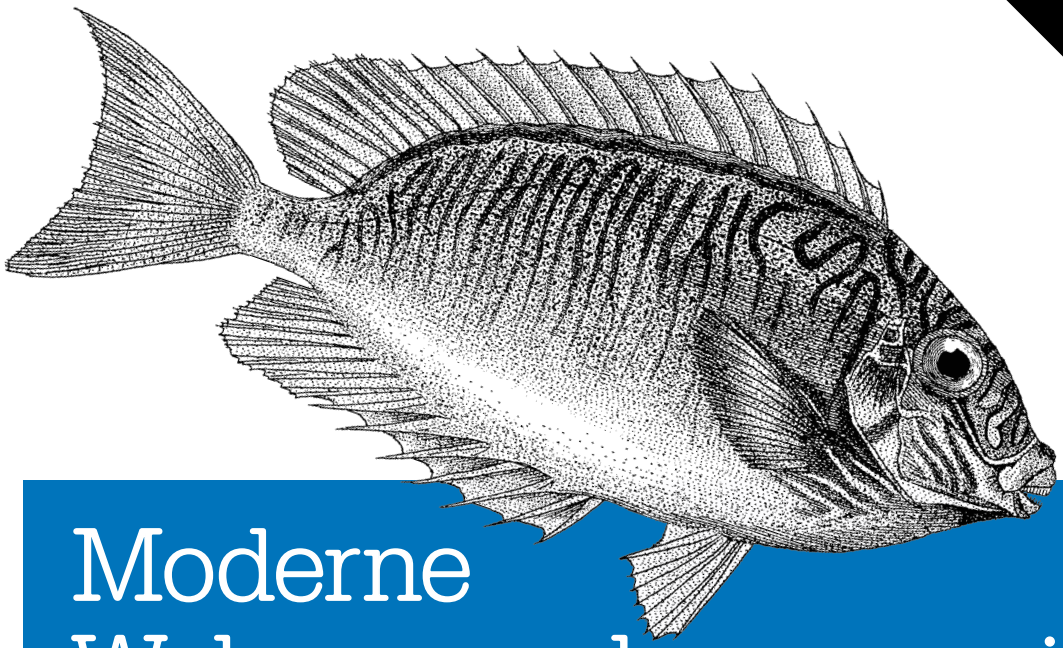
O'REILLY®

2. Auflage



Moderne Webanwendungen mit ASP.NET MVC & JavaScript

ASP.NET MVC IM ZUSAMMENSPIEL MIT WEB APIS UND JAVASCRIPT-FRAMEWORKS

Manfred Steyer
Holger Schwichtenberg

2. Auflage

Moderne Webanwendungen mit ASP.NET MVC und JavaScript

ASP.NET MVC im Zusammenspiel mit Web APIs und JavaScript-Frameworks

Manfred Steyer & Holger Schwichtenberg

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

2. Auflage 2014

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Lektorat: Ariane Hesse, Köln
Korrektorat: Dorothee Klein, Karin Baeyens, Siegen
Satz: mediaService, Siegen, *www.mediaservice.tv*
Umschlaggestaltung: Michael Oreal, Köln
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; *www.koeselbuch.de*

ISBN: 978-3-95561-740-0

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Vorwort **9**

1 ASP.NET MVC **15**
Architektur 15
Erste Schritte mit ASP.NET MVC 17
Controller 33
Views 40
Models 51
Globalisierung 57
Areas 60
Filter 63

2 ASP.NET Web API **69**
REST, WebAPIs und HTTP-Services 69
Einen einfachen HTTP-Service erstellen 70
Mehr Kontrolle über HTTP-Nachrichten 75
HTTP-Services über HttpClient konsumieren 80
Routen 83
Weiterführende Schritte mit der Web-API 86
Querschnittsfunktionen implementieren 92
Filterüberschreibungen 101
Benutzerdefinierte Formate unterstützen 101
Serialisierung beeinflussen 104
Web-API und HTML-Formulare 108
Fortschritt ermitteln 111
Feingranulare Konfiguration 112

3 JavaScript-Frameworks . . . 115
JavaScript als Multiparadigmen-Sprache . . . 116
JavaScript debuggen . . . 133
jQuery . . . 134
ASP.NET MVC-Modelle mit jQuery Validate validieren . . . 139
jQuery UI . . . 141
jQuery Mobile . . . 143
Twitter Bootstrap . . . 155
Offlinefähige Webanwendungen mit HTML 5 . . . 173
Asynchronität und Hintergrundprozesse . . . 187
Internationalisierung mit Globalize . . . 196
modernizr . . . 198
TypeScript . . . 199

4 AngularJS . . . 211
AngularJS herunterladen und einbinden . . . 211
MVC, MVP und MVVM mit AngularJS . . . 211
Erste Schritte mit AngularJS . . . 212
AngularJS näher betrachtet . . . 218
HTTP-Services via AngularJS konsumieren . . . 223
Angular-Services bereitstellen und konsumieren . . . 226
Filter in AngularJS . . . 227
Mit Formularen arbeiten . . . 230
Logische Seiten und Routing . . . 241
AngularJS-Anwendungen testen . . . 250
Benutzerdefinierte Direktiven . . . 262

5 ASP.NET SignalR . . . 279
Long-Polling . . . 279
Web-Sockets . . . 280
Überblick über ASP.NET SignalR . . . 281
PersistentConnection . . . 281
Hubs . . . 286
Pipeline-Module für Querschnittsfunktionen . . . 294
SignalR konfigurieren . . . 296
Cross Origin Resource Sharing (CORS) . . . 296
SignalR skalieren . . . 296

6 Datenzugriff mit Entity Framework . . . 303
Überblick . . . 303
Mit dem Entity Data Model arbeiten . . . 304
Daten abfragen . . . 311
Entitäten verwalten . . . 320
Erweiterte Mapping-Szenarien . . . 329
Mit gespeicherten Prozeduren arbeiten . . . 341
Mit nativem SQL arbeiten . . . 345
Codegenerierung anpassen . . . 346
Code First . . . 347
Datenbasierte Dienste mit dem Entity Framework, ASP.NET Web API und OData . . . 364

7 Basisdienste im ASP.NET-Umfeld . . . 377
Open Web Interface for .NET (OWIN) und Katana . . . 377
Direkt mit HTTP interagieren . . . 386
Zustandsverwaltung auf Sitzungsebene . . . 391
Caching . . . 397

8 Sicherheit . . . 405
Gesicherte Übertragung mit SSL/TLS . . . 405
Zugang zu Action-Methoden beschränken . . . 409
Windows-Sicherheit unter Verwendung von HTTP-basierter Authentifizierung . . . 411
Mit Clientzertifikaten arbeiten . . . 414
Sicherheitszenarien mit ASP.NET Identity und Katana . . . 417
Benutzerdefinierte Authentifizierungs-Middleware-Komponenten mit Katana entwickeln . . . 434
Single-Sign-On und weiterführende Szenarien mit OAuth 2.0, OpenID Connect und Katana . . . 446
Single-Sign-On mit WIF . . . 496

9 ASP.NET MVC und ASP.NET Web API erweitern . . . 501
ASP.NET MVC erweitern . . . 501
ASP.NET Web API erweitern . . . 529

10 Testbare Systeme mit Dependency-Injection . . . 537
Fallbeispiel ohne Dependency-Injection . . . 537
Fallbeispiel mit Dependency-Injection . . . 541
Zusammenfassung und Fazit . . . 547

Index . . . 549

Vorwort

Das Web hat sich verändert. Während für gute Webanwendungen vor rund 10 Jahren noch die Regel galt, so viele Aufgaben wie möglich auf dem Server zu erledigen, machen moderne Web-Systeme exzessiv von clientseitigen Techniken, allen voran JavaScript, Gebrauch, um die Benutzerfreundlichkeit zu steigern und sich so von Konkurrenz-Produkten abzuheben. Dank HTML 5 können nun auch offlinefähige Webanwendungen entwickelt werden, sodass Webtechniken eine ernstzunehmende Alternative zu klassischen Plattformen für die Entwicklung von Desktop-Clients darstellen. Darüber hinaus werden moderne Webanwendungen den klassischen mehrschichtigen Anwendungen, die sich auf bereitgestellte Services abstützen und von ihnen gezielt Informationen beziehen, immer ähnlicher.

Auch im Bereich mobiler Endgeräte spielen moderne Websysteme eine wichtige Rolle, zumal sie derzeit sowie in absehbarer Zukunft die einzige Technologie zur Entwicklung plattformübergreifender Lösungen darstellen: Egal ob auf dem iPhone, auf der Android-Plattform oder unter Windows (Phone) 8: Webtechnologien werden überall unterstützt.

Das vorliegende Buch präsentiert einen Technologie-Mix, mit dem .NET-Entwickler solche modernen Webanwendungen entwickeln können. Dabei wird auf serverseitige Technologien wie ASP.NET MVC, ASP.NET Web API oder das Entity Framework ebenso eingegangen wie auf clientseitige Konzepte, ohne die die beschriebene Art von Anwendungen nicht möglich wäre. Zu diesen clientseitigen Technologien zählen jQuery, jQuery Mobile, TypeScript sowie HTML-5-APIs für offinefähige Webanwendungen. Daneben widmet dieses Buch dem populären JavaScript-Framework AngularJS, das die Erstellung JavaScript-getriebener Anwendungen erleichtert und ein großes Augenmerk auf die Aspekte Testbarkeit und Wartbarkeit legt, ein eigenes Kapitel. Auch das Thema Sicherheit wird umfangreich behandelt, indem neben klassischen Szenarien auch auf erweiterte Ansätze wie Single-Sign-On und die Delegation von Rechten eingegangen wird. Hierzu geht das vorliegende Werk nicht nur auf moderne und weit verbreitete Protokolle wie OAuth 2.0 und Open Id Connect ein, sondern zeigt auch auf, wie diese Protokolle mit den jüngsten Neuerungen im Bereich der ASP.NET-Familie implementiert werden können. Eine Diskussion über Dependency Injection zur Steigerung der Testbarkeit am Ende des Buchs rundet den vermittelten Stoff ab.

Für wen dieses Buch gedacht ist

Das Buch richtet sich an Softwareentwickler, die bereits grundlegende Erfahrung mit .NET und C# sowie mit Webtechnologien, allen voran JavaScript, HTML und HTTP, haben und moderne Web-Anwendungen entwickeln wollen.

Zielsetzung des Buchs

Zielsetzung dieses Buchs ist es, dem Leser zu zeigen, wie die eingangs erwähnten Technologien eingesetzt werden können, um moderne Web-Anwendungen zu schaffen. Der Fokus liegt dabei auf den Konzepten, die hinter diesen Technologien stehen. Dazu spielt dieses Buch eine Vielzahl an Beispielen durch.

Es ist nicht das Ziel dieses Buchs, ein erschöpfendes Nachschlagewerk für die Vielzahl der von den genannten Frameworks bereitgestellten Klassen und Methoden zu sein. Für eine erschöpfende Auflistung sämtlicher Funktionen wird, gerade bei den clientseitigen Technologien, auf die umfangreichen frei verfügbaren Online-Referenzen verwiesen. Darüber hinaus gehen die Autoren auch davon aus, dass der Leser mit wenig Aufwand herausfinden kann, in welchem Namensraum bzw. in welcher Assembly sich eine erwähnte Klasse befindet, sofern diese nicht ohnehin durch die in Visual Studio verwendeten Vorlagen eingebunden wurden.

Unterschiede zur ersten Auflage

Das Buch wurde in Hinblick auf die nachfolgend genannten Framework-Versionen aktualisiert und erweitert. Die Beschreibung des JavaScript-Frameworks *knockout.js* wurde durch ein umfangreiches Kapitel, welches auf das mächtigere und weiter verbreitete Framework *AngularJS* eingeht, ersetzt. Der überwiegende Teil des Kapitels zum Thema Sicherheit wurde neu geschrieben, sodass es sich am neu eingeführten Framework *ASP.NET Identity* und dem Projekt Katana (*Microsoft.Owin*) orientiert. Auch die Möglichkeiten im Hinblick auf Self-Hosting-Szenarien werden nun vor dem Hintergrund von *Katana* beschrieben. Einige kleinere Kapitel wurden zusammengelegt und Leser-Feedback eingearbeitet. Daneben wurde das Kapitel zu JavaScript erweitert, sodass es nun unter anderem auf *Twitter Bootstrap*, *IndexedDb*, *Promises* und *Globalize* eingeht. Aus Platzgründen findet sich das Kapitel zu den Internet-Informationsdiensten (IIS) nicht mehr in dieser zweiten Auflage, zumal es auch stärker in den Bereich der Administration fällt. Es steht den Lesern jedoch über den Leserservice – weitere Informationen finden Sie am Ende des Vorworts – zum Download zur Verfügung.

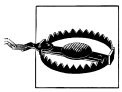
Typografische Konventionen

In diesem Buch haben wir die folgenden typografischen Konventionen verwendet:

Kursiv:
: Steht für Internet-Adressen wie zum Beispiel Domain-Namen und URLs sowie für neue Begriffe, wenn sie definiert werden.

`Nichtproportionalschrift`
: Um die Lesbarkeit des Textes zu verbessern, werden Dateinamen, Funktionen, Methoden, Eigenschaften u.a. in Nichtproportionalschrift abgesetzt.

Dieses Symbol weist auf einen Tipp, einen Vorschlag oder eine allgemeine Anmerkung hin.



Dieses Symbol weist auf eine Warnung hin.

Behandelte Versionen

Das Buch behandelt Microsoft .NET Framework 4.5.1, ASP.NET MVC 5.1, ASP.NET Web API 2, ASP.NET SignalR 2 sowie Entity Framework 6.1 und AngularJS 1.2. Die verwendete Entwicklungsumgebung ist Visual Studio 2013.

Express-Editionen

Microsoft bietet neben Visual Studio auch noch eine Produktfamilie unter dem Titel »Visual Studio Express Editionen« an. Bei den Express-Editionen handelt es sich um größere Bausteine, die aus Visual Studio herausgebrochen wurden und jeweils für einen speziellen Anwendungsfall bereitstehen. Diese Express-Editionen sind kostenlos auf der Microsoft-Website (*http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products*) beziehbar. Sie können ohne Gebühren unbegrenzt verwendet werden. Die Express-Editionen reichen für die wichtigsten Anwendungstypen aus und die meisten der hier beschriebenen Aspekte können damit realisiert werden. Aus Platzgründen geht dieses Buch jedoch nicht auf die Unterschiede zwischen der verwendeten Version, welche oben genannt wurde, und den kostenfreien Express-Editionen ein.

Verwendete Programmiersprache

Das vorliegende Buch verwendet serverseitig die Programmiersprache C# sowie clientseitig JavaScript.

Alle Leser, die serverseitig lieber mit Visual Basic .NET arbeiten, können die abgedruckten Beispiele sehr einfach mit kostenlosen Werkzeugen nach Visual Basic .NET konvertieren. Informationen dazu findet man unter *http://www.dotnetframework.de/tools.aspx*.

Sprachversion

Dieses Buch beschreibt die englische Version von Visual Studio, weil inzwischen viele deutsche Entwickler (einschließlich der Autoren) die englische Version der Software bevorzugen, zumal die Übersetzungen ins Deutsche oft holprig sind und die Fehlermeldungen nur schwerer verständlich machen. Anwender einer anderen Sprachversion können über den Befehl `Tools | Options | Environment | International Settings` weitere Sprachpakete herunterladen und einrichten. Weiterhin sei noch darauf hingewiesen, dass die Anordnung der Menüs und auch einige Tastaturkürzel von den gewählten Einstellungen in Visual Studio abhängen. Alle Ausführungen in diesem Buch beziehen sich auf die Umgebungseinstellung *Common Settings*, die bei der Installation des Produkts ausgewählt werden kann.

Leserservice

Den Lesern dieses Buchs werden von den Autoren folgende Serviceleistungen im Rahmen einer zugangsbeschränkten Website angeboten.

Der URL für den Zugang zum Leser-Portal lautet: *http://www.dotnetframework.de/leser*. Bei der Anmeldung müssen Sie das Kennwort *Into Darkness* angeben.

Downloads: Sie können alle in diesem Buch vorgestellten Codebeispiele hier herunterladen. Hier finden Sie außerdem ein Bonuskapitel zu den Internet-Informationsdiensten.

Diskussionsrunde: Ein webbasiertes Forum bietet die Möglichkeit, Fragen an die Autoren zu stellen. Bitte beachten Sie jedoch, dass dies eine freiwillige Leistung der Autoren ist und kein Anspruch auf eine kostenlose Betreuung besteht.

Newsletter: Alle registrierten Leser erhalten zwei- bis viermal jährlich einen Newsletter mit aktuellen Terminen und Publikationshinweisen.

Leserbewertung: Vergeben Sie Noten für dieses Buch und lesen Sie nach, was andere Leser von diesem Buch halten.

Errata: Trotz eines jahrelang erprobten Vorgehensmodells und der dreifachen Qualitätskontrolle (Co-Autor, Fachlektor, Verlag) ist es möglich, dass sich einzelne Fehler in dieses Buch eingeschlichen haben. Im Webportal können Sie nachlesen, welche Fehler gefunden wurden. Sie können hier auch selbst Fehler melden, die Ihnen auffallen.

Danksagungen

Unseren Dank für ihre Mitwirkung an diesem Buch möchten wir aussprechen an

- unsere Familienangehörigen, allen voran und in alphabetischer Reihenfolge *Felix*, *Heidi*, *Kerstin* und *Maja*, die uns neben unserem Hauptberuf das Umfeld geschaffen haben, auch an manchen Abenden und Wochenenden an diesem Buch zu arbeiten
- *Hans-Peter Grahsl* von der FH CAMPUS 02, der wertvolles Feedback zum Kapitel über AngularJS gegeben hat, und *Arno Hollosi* von der FH CAMPUS 02, der regelmäßig im Zuge von Diskussionen wichtige Aspekte und Sichtweisen einbringt
- die Microsoft Press-Lektoren *Thomas Braun-Wiesholler* und *René Majer*, die die erste Auflage dieses Buchs von der Verlagsseite aus betreut haben
- die O'Reilly-Lektorin *Ariane Hesse*, die die vorliegende zweite Auflage dieses Buchs betreut hat
- den Fachlektor *Uwe Thiemann*, der alle Texte inhaltlich geprüft hat
- die Korrektorinnen *Dorothee Klein* und *Karin Baeyens*, die das Buch sprachlich verbessert haben und
- *Gerhard Alfes* von mediaService, der sich um die optischen Aspekte des Buchs gesorgt hat

Graz & Essen, im März 2014

Manfred Steyer & Holger Schwichtenberg

Über www.IT-Visions.de

Die beiden Autoren arbeiten bei der Firma *www.IT-Visions.de* als Softwarearchitekten, Softwareentwickler, Trainer und Berater für .NET-Techniken. www.IT-Visions.de ist ein Verbund der deutschen Top-Experten im Bereich der Microsoft-Produkte und -Technologien, insbesondere .NET. Unter Leitung und Mitwirkung des bekannten .NET-Experten Dr. Holger Schwichtenberg bietet *www.IT-Visions.de*:

- Strategische und technische Beratung
- Konzepte, Machbarkeitsstudien und Reviews
- Coaching bei Entwicklungsprojekten
- Technischen Support vor Ort und via Telefon, E-Mail oder Web-Konferenz
- Individuell zugeschnittene technische Vor-Ort-Schulungen und anforderungsorientierte Workshops
- Öffentliche Seminare (in Kooperation mit dem Heise-Verlag), siehe *www.dotnet-akademie.de*

Die Schwestergesellschaft *5Minds IT-Solutions GmbH & Co. KG* bietet Softwareentwicklung (Prototypen und komplette Lösungen) sowie den Verleih von Softwareentwicklern.

Zu den Kunden gehören neben vielen mittelständischen Unternehmen auch Großunternehmen wie z.B. E.ON, Bertelsmann, EADS, Siemens, MAN, Bayer, VW, Bosch, ThyssenKrupp, Merkle, Fuji, Festo, Deutsche Post, Deutsche Telekom, Fielmann, Roche, HP, Jenoptik, Hugo Boss, Zeiss, IKEA, diverse Banken und Versicherungen sowie mehrere Landesregierungen.

Firmenwebsites: *http://www.IT-Visions.de* und *http://www.5minds.de*

Über den Autor Manfred Steyer

Manfred Steyer ist Trainer und Berater bei *www.IT-Visions.de* sowie verantwortlich für den Fachbereich Software Engineering der Studienrichtung IT und Wirtschaftsinformatik an der FH CAMPUS 02 in Graz.

Er schreibt für das windows.developer magazin (vormals dot.net magazin) sowie Heise Developer und ist Buchautor bei O'Reilly, Microsoft Press sowie Carl Hanser. Manfred hat berufsbegleitend IT und IT-Marketing in Graz sowie Computer Science in Hagen studiert und kann auf mehr als 15 Jahre an Erfahrung in der Planung und Umsetzung von großen Applikationen zurückblicken. Er ist ausgebildeter Trainer für den Bereich der Erwachsenenbildung und spricht regelmäßig auf Fachkonferenzen.

In der Vergangenheit war Manfred Steyer mehrere Jahre für ein großes österreichisches Systemhaus tätig. In der Rolle als Bereichsleiter hat er gemeinsam mit seinem Team Geschäftsanwendungen konzipiert und umgesetzt.

Sein Weblog erreichen Sie unter *www.softwarearchitect.at*.

Seine E-Mail-Adresse lautet *m.steyer@IT-Visions.at*.

Über den Autor Dr. Holger Schwichtenberg

- Studienabschluss Diplom-Wirtschaftsinformatik an der Universität Essen
- Promotion an der Universität Essen im Gebiet komponentenbasierter Softwareentwicklung
- Seit 1996 selbstständig als unabhängiger Berater, Dozent, Softwarearchitekt und Fachjournalist
- Leiter des Berater- und Dozententeams bei *www.IT-Visions.de*
- Leitung der Softwareentwicklung im Bereich Microsoft/.NET bei der 5minds IT-Solutions GmbH & Co. KG (*www.5minds.de*)
- 60 Fachbücher bei Microsoft Press, Addison-Wesley und dem Carl Hanser-Verlag und mehr als 700 Beiträge in Fachzeitschriften
- Gutachter in den Wettbewerbsverfahren der EU gegen Microsoft (2006–2009)
- Ständiger Mitarbeiter der Zeitschriften iX (seit 1999), dotnetpro (seit 2000) und Windows Developer (seit 2010) sowie beim Online-Portal *heise.de* (seit 2008)
- Regelmäßiger Sprecher auf nationalen und internationalen Fachkonferenzen (z.B. TechEd, Microsoft IT Forum, BASTA, BASTA-on-Tour, Advanced Developers Conference, .NET-Entwicklerkonferenz, OOP, VS One, Wirtschaftsinformatik, Net.Object Days, Windows Forum, DOTNET-Konferenz, XML-in-Action)
- Zertifikate und Auszeichnungen von Microsoft:
  - Microsoft Most Valuable Professional (MVP)
  - Microsoft Certified Solution Developer (MCSD)
- Thematische Schwerpunkte:
  - Microsoft .NET Framework, Visual Studio, C#, Visual Basic
  - .NET-Architektur/Auswahl von .NET-Technologien
  - Einführung von .NET Framework und Visual Studio/Migration auf .NET
  - Webanwendungsentwicklung mit IIS, ASP.NET und AJAX
  - Enterprise .NET, verteilte Systeme/Webservices mit .NET
  - Relationale Datenbanken, XML, Datenzugriffsstrategien
  - Objektrelationales Mapping (ORM), insbesondere ADO.NET Entity Framework
  - Windows PowerShell (WPS) und Windows Management Instrumentation (WMI)
- Ehrenamtliche Community-Tätigkeiten:
  - Vortragender für die International .NET Association (INETA)
  - Betrieb diverser Community-Websites *www.dotnetframework.de*, *www.entwickler-lexikon.de*, *www.windows-scripting.de*, *www.aspnetdev.de* u.a.
- Weblog: *http://www.dotnet-doktor.de*
- Kontakt: *hs@IT-Visions.de* sowie Telefon 0201 649590-0

KAPITEL 1

ASP.NET MVC

ASP.NET MVC ist ein Framework aus der ASP.NET-Familie, welches zum einen das weit verbreitete MVC-Muster implementiert und dem Entwickler zum anderen die volle Kontrolle über die gerenderten Webseiten gibt. Es erlaubt das dynamische Rendern von Ressourcen wie z.B. HTML-Seiten auf der Serverseite. Dieses Kapitel zeigt, wie dieses Framework eingesetzt wird.

Architektur

Bevor die ersten Codebeispiele folgen, beschäftigt sich dieser Abschnitt mit verschiedenen Überlegungen zum MVC-Muster. Darüber hinaus wird auf das MVVM-Muster (Model-View-ViewModel, auch als Model-View-Presenter bekannt) eingegangen, da dieses heutzutage häufig in Kombination mit MVC eingesetzt wird.

Model-View-Controller (MVC)

Die Abkürzung MVC steht für Model-View-Controller, ein Pattern, das ursprünglich bei Xerox für die Trennung von Logik und Präsentation entwickelt wurde. Es sieht vor, dass eine Applikation in drei Teile aufgeteilt wird: Model, View und Controller (Abbildung 1-1).

Das Modell entsprach dabei ursprünglich den fachlichen Daten sowie den darauf operierenden Routinen. Da diese beiden Aspekte heutzutage in der Regel voneinander getrennt werden, wird das Modell häufig lediglich mit den Daten der Applikation assoziiert und die Operationen für diese Daten, wie Laden, Speichern oder das Durchführen von Berechnungen, in eigene Klassen ausgelagert. Die Aufgabe der View ist das Anzeigen von Models sowie das Entgegennehmen von Benutzereingaben. Der Controller stellt das Bindeglied zwischen Modell und View dar: Er nimmt Anfragen sowie Benutzereingaben entgegen und wählt zur Abarbeitung der Anfrage eine passende Routine aus. Anschließend wird eine View ausgewählt sowie die anzuzeigenden Daten in Form eines Models an diese übergeben.

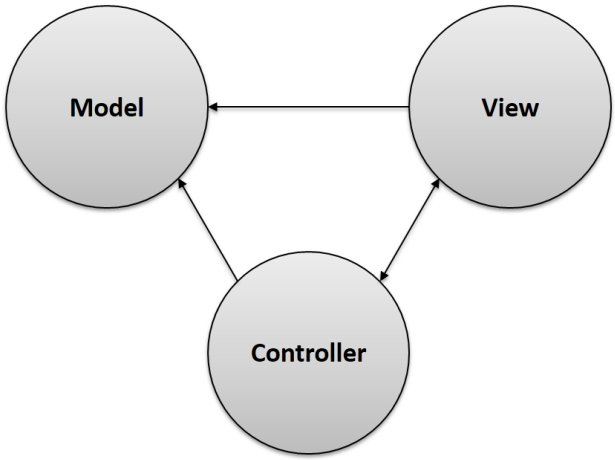


Abbildung 1-1: Das MVC-Muster

Durch diese Trennung können die einzelnen Teile separat wiederverwendet werden. Beispielsweise müssten bei einem Produkt, welches an das Design verschiedener Kunden anzupassen ist, lediglich die Views ausgetauscht bzw. modifiziert werden. Daneben erleichtert es das gleichzeitige Unterstützen verschiedener Benutzerschnittstellen – zum Beispiel eine für Mitarbeiter, eine weitere für Kunden und eine für mobile Endgeräte. Durch die Verteilung der einzelnen Aufgaben auf die Komponenten Modell, View und Controller wird auch eine eventuell gewünschte Arbeitsteilung vereinfacht. Webdesigner könnten sich beispielsweise um die View kümmern, Entwickler um den Controller sowie um die von ihm angestoßenen Routinen und Datenbankexperten um das Modell, welches sich ggf. auf einen O/R-Mapper, wie das ADO.NET Entity Framework, stützt. Da die gesamte Logik durch den Controller widergespiegelt wird, wird auch das Testen sowie das Automatisieren von Tests erleichtert.

Überblick über MVVM (Model-View-ViewModel)

Da dieselben fachlichen Daten in unterschiedlichen Views häufig unterschiedlich angezeigt werden, sieht das Muster *MVVM* (*Model-View-ViewModel*) vor, dass jede View ein eigenes Modell erhält, welches als *ViewModel* bezeichnet wird. Dieses basiert auf einem oder mehreren Models und bereitet deren Daten für die Verwendung innerhalb der View auf. Zusätzlich kann es auch Berechnungen durchführen oder benachbarte Objekte in einer »flachen« Struktur anbieten.

Beispielsweise könnte so für eine Rechnung auch die Anzahl der stattgefundenen Mahnungen über eine Eigenschaft angeboten oder die von der View zu verwendende Hintergrundfarbe in Hinblick auf das Hervorheben mehrfach gemahnter Rechnungen ermittelt werden. Darüber hinaus ist es auch nicht unüblich, im ViewModel Methoden zu hinterlegen, welche sich um die Verarbeitung der Daten (Validieren, Speichern, Laden, Berechnungen) kümmern oder die damit verbundenen Verarbeitungsvorgänge zumindest anstoßen, indem sie an die entsprechenden Klassen weiterdelegieren.

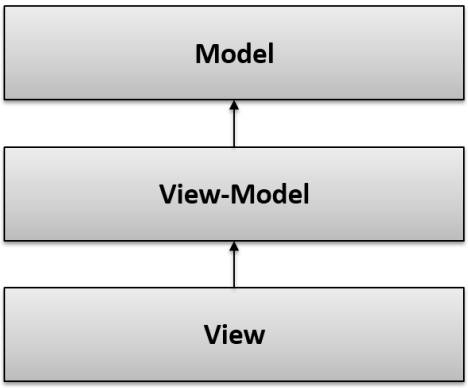


Abbildung 1-2: Das Muster MVVM (Model-View-ViewModel)

Beim Modell im Sinne von MVC handelt es sich somit, zumindest bei größeren Anwendungen, häufig um ein View-Modell, welches Daten für eine bestimmte View oder eine Gruppe von Views zur Verfügung stellt.

Erste Schritte mit ASP.NET MVC

Zur Einführung in ASP.NET MVC zeigt dieser Abschnitt, wie damit eine einfache Webanwendung zur Verwaltung von Hotels erstellt werden kann. Damit Sie einen guten Überblick erhalten, werden viele Themen, die in den folgenden Abschnitten detailliert erläutert werden, an dieser Stelle nur gestreift. Zur Vereinfachung kommt keine Datenbank, sondern lediglich eine statische Liste zum Einsatz. Diese erlaubt es Ihnen, sich an dieser Stelle voll und ganz auf ASP.NET MVC zu konzentrieren.

ASP.NET MVC-Projekt anlegen

Um eine ASP.NET-MVC-Anwendung zu erzeugen, wählt der Entwickler in Visual Studio 2013 die Projektvorlage *Web | ASP.NET Web Application*. Daraufhin wird er aufgefordert, sich für ein Framework aus der ASP.NET-Familie zu entscheiden. Die Wahl fällt dabei auf MVC. Im unteren Bereich kann der Entwickler weitere Frameworks auswählen, die im Projekt zusätzlich zu ASP.NET MVC zum Einsatz kommen sollen. Davon muss, um den Ausführungen in diesem Kapitel folgen zu können, jedoch kein Gebrauch gemacht werden. Nach dem Bestätigen dieser Auswahl erzeugt Visual Studio ein neues ASP.NET MVC-Projekt. Die Struktur eines solchen Projekts ist in Abbildung 1-3 dargestellt.

Dabei fallen drei Ordner auf: *Models*, *Views* und *Controllers*. Der Ordner *Controller* beinhaltet die einzelnen Controller-Klassen, deren Namen per Definition auf `Controller` enden. Beispielsweise stellt die Projektvorlage *Internet Application* einen *HomeController* für die Startseite sowie einen *AccountController* für das Registrieren und Anmelden von Benutzern zur Verfügung. Der Ordner *Models* beherbergt die einzelnen Modellimplementierungen. Dabei handelt es sich in der Regel um Klassen, welche Konzepte aus dem Bereich der Anwendung repräsentieren, zum Beispiel Hotels bei einer Anwendung zur Verwaltung von Hotels.

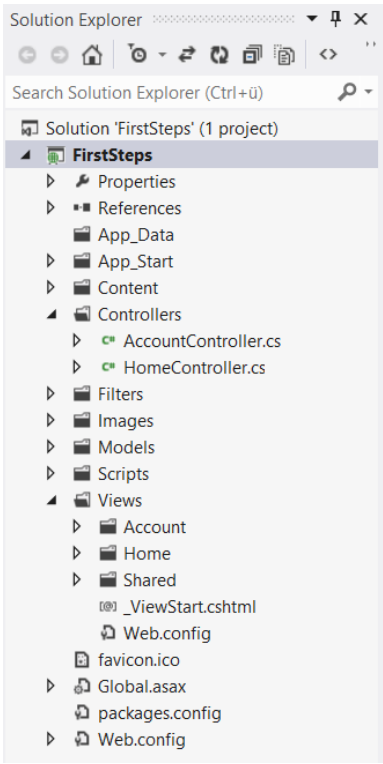


Abbildung 1-3: Verzeichnisstruktur eines ASP.NET MVC-Projekts

Der Ordner `Views` ist in weitere Ordner unterteilt, wobei jeder Ordner die Views eines bestimmten Controllers beinhaltet und auch dessen Namen (ohne die Endung Controller) übernimmt. Der Ordner `Views/Home` beinhaltet zum Beispiel die Views, an die der `HomeController` delegiert; `Views/Account` jene, die der `AccountController` nutzt. Die einzige Ausnahme ist der Ordner `Shared` – er beinhaltet Views, welche von sämtlichen Controllern verwendet werden können, darunter allgemeine Fehlerseiten oder Layoutseiten, die für eine einheitliche Darstellung der einzelnen Views sorgen.

Um einen neuen Controller hinzuzufügen, kann der Befehl `Add | Controller` aus dem Kontextmenü des Ordners `Controllers` verwendet werden; um eine neue View hinzuzufügen, der Befehl `Add | View` aus dem Kontextmenü des Ordners `Views` bzw. eines darunter liegenden Ordners. Zum Anlegen eines neuen Models existiert hingegen kein eigener Befehl, da es sich hierbei in der Regel um herkömmliche Klassen handelt, die keine gemeinsamen Charakteristika aufweisen, wie z.B. gemeinsame Basisklassen.

Modell anlegen

Für die hier beschriebene erste Beispielanwendung wird im Ordner `Models` eine Klasse `Hotel` (Beispiel 1-1) angelegt. Wenn auch die Projektvorlage diesen Ordner für Models vorsieht, kann der Entwickler einen beliebigen Ordner hierfür verwenden. Es wäre auch möglich, Models in anderen Namensräumen oder gar Assemblies zu hinterlegen.

Beispiel 1-1: Einfaches Model

```
public class Hotel
{
    public int HotelId { get; set; }
    public string Bezeichnung { get; set; }
    public int Sterne { get; set; }
}
```

Zum Laden und Speichern von Hotels wird darüber hinaus unterhalb des Projektordners ein Ordner `Data` mit einer `HotelRepository`-Klasse eingerichtet (Beispiel 1-2). Um das Beispiel einfach zu halten, verwaltet es Hotels in einer statischen Liste anstatt in einer Datenbank, sodass sich der Leser auf ASP.NET MVC konzentrieren kann und sich nicht darüber hinaus noch mit Aspekten des Datenbankzugriffs, welche in Kapitel 6 behandelt werden, belasten muss.

Beispiel 1-2: Repository für Hotels

```
public class HotelRepository
{
    private static List<Hotel> hotels = new List<Hotel>();

    static HotelRepository()
    {
        hotels.Add(new Hotel { HotelId = 1, Bezeichnung = "Hotel zur Post", Sterne = 2 });
        hotels.Add(new Hotel { HotelId = 2, Bezeichnung = "Hotel Rebstock", Sterne = 3 });
        hotels.Add(new Hotel { HotelId = 3, Bezeichnung = "Wellness Hotel", Sterne = 4 });
    }

    public List<Hotel> FindAll()
    {
        return hotels;
    }

    public Hotel FindById(int id)
    {
        return hotels.Where(h => h.HotelId == id).FirstOrDefault();
    }

    public void Delete(int hotelId)
    {
        var hotel = FindById(hotelId);
        hotels.Remove(hotel);
    }

    public void Save(Hotel h)
    {
        if (h.HotelId == 0)
        {
            // Neues Hotel einfügen
            hotels.Add(h);
            h.HotelId = hotels.Max(htl => htl.HotelId) + 1;
        }
        else
        {
            // Bestehendes Hotel aktualisieren
            var hotel = FindById(h.HotelId);
            hotel.Bezeichnung = h.Bezeichnung;
```

```
                hotel.Sterne = h.Sterne;
            }
        }
    }
```

Controller anlegen

Zum Anlegen des Controllers für das hier betrachtete Beispiel wählt der Entwickler aus dem Kontext-Menü des Ordners `Controllers` den Befehl `Add | Controller`. Für die Erstellung der hier beschriebenen Beispielanwendung wird die Vorlage `Empty MVC Controller` gewählt, sodass der Entwickler gezwungen ist, die benötigten Methoden selbst einzurichten.

Den Controller-Namen legt er passend zum Modell auf `HotelController` fest. Nachdem er das Dialogfeld bestätigt hat, richtet Visual Studio eine neue Controller-Klasse ein, welche von der Basisklasse `Controller` erbt. Alternativ dazu könnte der Entwickler diese Klasse auch von `ControllerBase` erben lassen oder stattdessen das Interface `IController` implementieren. In der Praxis wird dies selten gemacht, da dieses Verhalten mit dem Verlust jenes Komforts, der durch die Basisklasse `Controller` geboten wird, einhergeht.

Die öffentlichen Methoden der Controller-Klassen nennen sich Action-Methoden. Sie werden auf einen URL abgebildet und liefern per Definition eine Instanz von `ActionResult` zurück. Der URL, auf den sie standardmäßig abgebildet werden, lautet auf `/controller/action`, wobei `controller` für den Namen des Controllers (ohne die Endung `Controller`) sowie `action` für den Namen der Action-Methode steht. Die Methode `Index` in Beispiel 1-3 wird demnach angestoßen, wenn der Anwender auf den URL */hotel/index* zugreift. Da es sich bei Index um den standardmäßig verwendeten Methodennamen handelt, könnte der Benutzer alternativ dazu auch nur auf */hotel* zugreifen. Der standardmäßige Controller-Name lautet im Übrigen auf `Home`, was auch der Grund dafür ist, dass ohne Angabe eines URLs die Action-Methode `Index` im Controller `HomeController` angestoßen wird, welche zur Anzeige der Startseite führt.

Soll eine Methode nicht als Action-Methode herangezogen werden, kann diese mit dem Attribut *NonAction* annotiert werden. Soll innerhalb des URLs eine vom Methodennamen abweichende Bezeichnung verwendet werden, kann diese über das Attribut *ActionName*, mit dem die jeweilige Action-Methode zu markieren ist, festgelegt werden.

Die Action-Methode `Index` im betrachteten Controller lädt sämtliche Hotels über das eingerichtete `HotelRepository` und erzeugt anschließend mit der von `Controller` geerbten Hilfsmethode `View` ein `ActionResult`, welches ASP.NET MVC anweist, eine View zu rendern. Den Namen dieser View übergibt sie als erstes Argument, das von der View anzuzeigende Modell – eine Liste mit Hotels – als zweites Argument. Da sie als Namen der View `Index` angibt, sucht ASP.NET MVC im `View`-Ordner des Controllers, der den Namen `/Views/Hotel` trägt, nach dieser View. Wird das Framework dort nicht fündig, wird die Suche nach der View `Index` im Ordner `/Views/Shared` fortgesetzt. Bleibt auch dieser Anlauf erfolglos, erhält der Aufrufer eine Fehlermeldung.

Da der Name der gewünschten View dem Namen der Action-Methode gleicht (beide nennen sich `Index`), kann der Entwickler beim Aufruf der Methode `View` auch den Namen der View weglassen. Das Ergebnis dieses Aufrufs, welcher durch den Kommentar am Ende der Methode `Index` angedeutet wird, wäre dasselbe: ASP.NET MVC würde trotzdem nach einer View mit dem Namen `Index` Ausschau halten.

Beispiel 1-3: Einfacher Controller zum Abfragen von Hotels

```
public class HotelController : Controller
{
    // Mapping auf: /Hotel/Index
    //              /Hotel
    public ActionResult Index()
    {
        var rep = new HotelRepository();
        var hotels = rep.FindAll();
        return View("Index", hotels);
        // return View(hotels);
    }
}
```

View anlegen

Um die in Beispiel 1-3 referenzierte View nur für den gezeigten Controller bereitzustellen, müssen Sie sie im Verzeichnis `/Views/Hotels` einrichten. Um sich hierbei von Visual Studio unterstützen zu lassen, führen Sie einen Rechtsklick auf dem Inhalt der Action-Methode aus und wählen anschließend den Befehl `Add View` (Abbildung 1-4). Daraufhin zeigt Visual Studio ein Dialogfeld an, über das der Entwickler Eckdaten zur anzulegenden View bekannt geben kann (Abbildung 1-5). Abgesehen vom Namen kann nun festgelegt werden, welche Vorlage für die neue View zu verwenden ist. Beispielsweise existieren Vorlagen für Übersichtslisten und Detailansichten. Stützt sich eine Vorlage auf ein Model, muss der Entwickler auch eine Model-Klasse auswählen.

Die Option `Create as partial view` gibt an, ob die neue View als sogenannte partielle View, welche in andere Views eingebettet wird, einzurichten ist. Mit der Option `Use a layout page` gibt der Entwickler an, dass das Ergebnis der View in eine Layoutseite, welche immer wiederkehrende Elemente wie Menüeinträge oder eine Fußzeile beinhaltet, eingebettet werden soll. Wählt der Entwickler diese Option aus, ohne eine Layout-Seite im sich darunter befindlichen Eingabefeld anzugeben, verwendet Visual Studio eine standardmäßige Layoutseite. Weitere Infos zu Layoutseiten finden Sie im Abschnitt »Layoutseiten«.

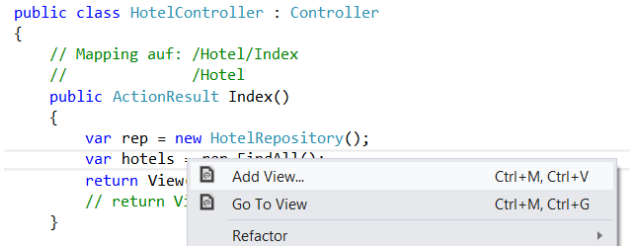


Abbildung 1-4: Kontextmenübefehl zum Hinzufügen einer neuen View

Um eine View für die zuvor beschriebene Action-Methode `Index` aus dem eingerichteten `HotelController` anzulegen, vergibt der Entwickler den Namen `Index`. Daneben wählt er die Vorlage *List* sowie das Model `Hotel` aus. Nach dem Bestätigen der Eingaben richtet Visual Studio im Verzeichnis `/Views/Hotel` eine View mit dem Namen `Index.cshtml` ein. Würde Visual Basic verwendet, hätte die View den Namen `Index.vbhtml`. Beispiel 1-4 zeigt den Inhalt dieser View.

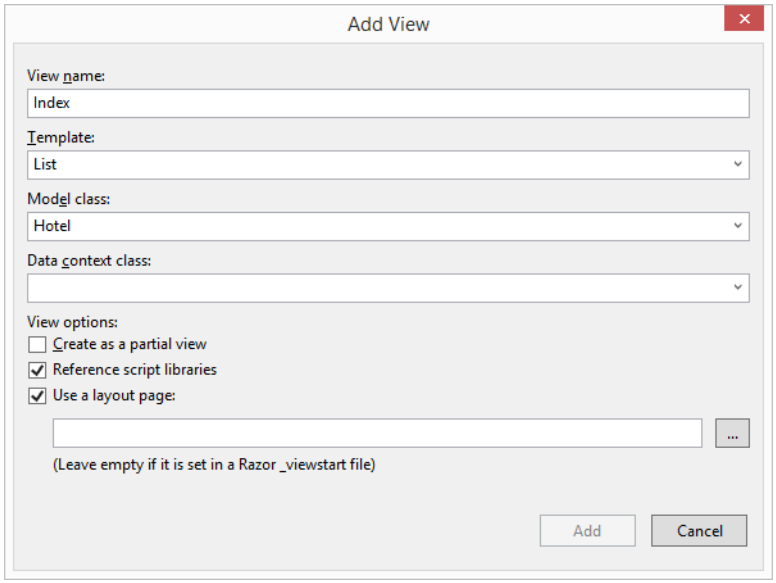


Abbildung 1-5: Neue View einfügen

Zum Rendern einer View greift ASP.NET MVC auf eine sogenannte View-Engine zurück. Die standardmäßig verwendete View-Engine, welche sich im Lieferumfang von ASP.NET MVC befindet, nennt sich Razor. Anweisungen, die mit einem `@` beginnen, sowie Anweisungen, die sich innerhalb von `@{ … }` befinden, führt Razor im Zuge des Renderings aus. Alle anderen Zeilen werden in Form von HTML-Markup direkt ausgegeben.

Die einzelnen von Visual Studio verwendeten Vorlagen finden sich unter *C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates*, sofern Visual Studio auf Laufwerk C: installiert ist. Die dort vorhandenen Ordner gruppieren die Templates nach Anwendungsfall. Beispielsweise erwartet Visual Studio im Ordner *MvcView* die Vorlagen, welche eine View für ein bestimmtes Model erzeugen. Ein Beispiel dafür ist die zuvor beschriebene Vorlage *List*, welche eine Übersichtsliste für eine Auflistung mit Model-Instanzen erzeugt. Daneben beinhaltet der Ordner *MvcViewWithoutModel* Vorlagen für Views, deren Aufbau sich nicht an einem Model orientiert. Ordner, die mit dem Präfix *ApiController* beginnen, enthalten Vorlagen für Controller. Die Vorlagen stehen standardmäßig in einer Version für C# sowie in einer Version für VB.NET zur Verfügung. Die C#-Versionen weisen die Endung *cs.t4* auf; die VB.NET-Versionen verwenden hingegen die Endung *vb.t4*. Das Kürzel *t4* steht für die gleichnamige Technologie, die Microsoft schon seit einigen Jahren für die Erzeugung von Code-Vorlagen einsetzt.

Um diese Vorlagen für ein eigenes Projekt zu erweitern oder abzuändern, erzeugt der Entwickler innerhalb des Projekts den Ordner *CodeTemplates* und kopiert den Inhalt des oben genannten Ordners dort hinein. Anschließend kann er die Vorlagen abändern und erweitern. Legt er zum Beispiel im Ordner *CodeTemplates\MvcView* eine Kopie von *Edit.cs.t4* mit dem Namen *Edit2.cs.t4* an, steht ihm beim Erzeugen einer neuen View auch dieses Template zur Verfügung.

Die Anweisung `@model` legt den Typ des darzustellenden Modells auf `IEnumerable<FirstSteps.Models.Hotel>` fest. Die View erwartet demnach als Modell eine Auflistung mit `Hotel`-Objekten. Dies macht die

View zu einer streng typisierten View und erlaubt es dem Entwickler, über die Eigenschaft `Model` sämtliche übergebenen `Hotel`-Instanzen in diesem Modell zu iterieren. Beispielsweise könnte er durch die folgende Anweisung für jedes Hotel im übergebenen Modell einen Absatz bestehend aus der Eigenschaft Bezeichnung und Sterne ausgeben lassen, wobei die beiden Eigenschaften in der Ausgabe durch ein Komma getrennt dargestellt würden.

```
@foreach(var h in Model) { <p>@h.Bezeichnung, @h.Sterne</p> }
```

Da der Typ des Models bekannt ist, unterstützt Visual Studio das Formulieren dieser Anweisung mittels IntelliSense. Wie von Geisterhand scheint Razor die Grenzen zwischen den Codefragmenten, die serverseitig im Zuge des Renderings zur Ausführung gebracht werden, und jenen, die direkt in Form von HTML-Markup auszugeben sind, zu erkennen. Beispielsweise erkennt Razor, dass die schließende geschweifte Klammer am Ende zur serverseitigen `foreach`-Anweisung gehört, oder dass das Komma, welches die Bezeichnung von den Sternen trennt, direkt in das Markup auszugeben ist. Dass Razor hierzu klare Regeln verwendet, die man glücklicherweise in den meisten Fällen nicht beachten muss, wird später im Abschnitt »Razor« aufgezeigt.

Legt der Entwickler mit `@model` den Typ des Models nicht fest, wird `Model` zur dynamischen Eigenschaft, für die der Entwickler natürlich von Visual Studio auch kein IntelliSense erhält. Wie bei dynamischen Eigenschaften üblich, kann auch der Compiler nicht prüfen, ob durchgeführte Zugriffe möglich sind. Vielmehr winkt er alle Zugriffe durch und Fehler werden erst zur Laufzeit durch eine Ausnahme abgemahnt.

Die Anweisung `ViewBag.Title` legt den Seitentitel, den die Layoutseite im Kopfbereich der HTML-Seite auszugeben hat, fest. Die Hilfsmethode `@Html.ActionLink` erzeugt einen Link, der die Beschriftung `Create New` trägt und auf die Action-Methode `Create` des zuletzt verwendeten Controllers verweist. Über eine Überladung dieser Methode könnte der Entwickler auch einen anderen Controller referenzieren.

Standardmäßig lautet der URI, der von dieser Methode verwendet wird, `/Hotel/Create`, weswegen der Entwickler in Versuchung kommen könnte, den entsprechenden HTML-Befehl für einen solchen Link direkt in der View zu platzieren. Da man allerdings über die Konfiguration die Art und Weise, wie URLs auf Action-Methoden abzubilden sind, ändern kann, erscheint es sinnvoller, diese Hilfsmethode, welche den tatsächlichen URL aufgrund dieser Konfigurationseinstellungen ermittelt, heranzuziehen.

`@Html.DisplayNameFor` rendert den Namen einer Eigenschaft, der in Form eines Lambda-Ausdrucks übergeben wird. Dieser Lambda-Ausdruck bildet eine Instanz des Typs `Hotel` auf die gewünschte Eigenschaft ab. Bei `model` handelt es sich dabei um den frei wählbaren Parameternamen dieses Lambda-Ausdrucks.

Den anzuzeigenden Namen könnte man zwar an dieser Stelle auch direkt hinterlegen, allerdings gibt ASP.NET MVC dem Entwickler die Möglichkeit, direkt im Modell Metadaten, wie eben die anzuzeigenden Eigenschaftsnamen, zu platzieren. Und genau diese Bezeichnung würde `DisplayNameFor` in Erfahrung bringen und ausgeben. Somit könnte der Entwickler beispielsweise festlegen, dass das Feld Sterne in der GUI als Hotelkategorie bezeichnet werden soll und `DisplayNameFor` würde hierfür einheitlich diese Bezeichnung zu Tage fördern.

Die betrachtete View iteriert mit einer `foreach`-Schleife durch das Modell, wobei es sich um eine Auflistung mit `Hotel`-Objekten handelt. Pro Hotel gibt sie die Eigenschaften `Bezeichnung` und `Sterne` aus. Hierzu würden die Anweisungen `@item.Bezeichnung` bzw. `@item.Sterne` reichen. Stattdessen verwendet

die View jedoch die Hilfsmethode `DisplayFor`, an die die auszugebende Eigenschaft als Lambda-Ausdruck übergeben wird. Der Grund dafür liegt in der Tatsache, dass der Entwickler pro `Model`-Eigenschaft eine Vorlage definieren kann, mit der die Eigenschaft angezeigt werden soll. Existiert für eine Eigenschaft keine solche Vorlage, prüft ASP.NET MVC, ob für den Datentyp der Eigenschaft eine Vorlage vorliegt. Standardmäßig existiert zum Beispiel eine Vorlage für boolesche Felder, die diese in Form eines nicht editierbaren Kontrollkästchens (ausgefüllt entspricht `true`, leer entspricht `false`) anzeigt. Weitere Informationen hierzu finden Sie im Abschnitt »Vorlagen für Felder und Models«.

Am Ende rendert die betrachtete View auch noch drei Links, welche auf die (noch nicht existierenden) Controller-Methoden `Edit`, `Details` und `Delete` verweisen und dieselbe Beschriftung tragen. Die Eigenschaften des übergebenen anonymen Objekts werden in die generierten Links in Form von URL-Parametern integriert. Eigenschaften mit der Bezeichnung `id` kommt dabei jedoch eine Sonderstellung zu: Sie werden standardmäßig in den URL eingefügt. Der Action-Link mit der Beschriftung `Edit` würde somit auf den URL */Hotel/Edit/7* verweisen, sofern die Eigenschaft `id` den Wert 7 aufwiese. Dies ist aus Sicht von ASP.NET MVC (beim Einsatz der Standardeinstellungen) gleichbedeutend mit */Hotel/Edit?id=7*. Da die erste Variante von Suchmaschinen bevorzugt wird – zumal HTTP davon ausgeht, dass jede Ressource ihren eigenen URL besitzt – nimmt ASP.NET MVC damit vorlieb. Alle anderen Parameter werden standardmäßig jedoch immer an den URL angehängt. Hieße der zu übergebende Parameter zum Beispiel `HotelNr`, würde die Hilfsmethode `ActionLink` hierfür den folgenden URL rendern: */Hotel/Edit?HotelNr=7*.

Beispiel 1-4: Einfache View zum Anzeigen von Hotels

```
@model IEnumerable<FirstSteps.Models.Hotel>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Bezeichnung)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Sterne)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Bezeichnung)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Sterne)
        </td>
        <td>
```

```
            @Html.ActionLink("Edit", "Edit", new { id=item.HotelId }) |
            @Html.ActionLink("Details", "Details", new { id=item.HotelId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.HotelId })
        </td>
    </tr>
}

</table>
```

Webanwendung testen

Um die erzeugte Webseite zu testen, führt der Entwickler das Projekt aus. Dies bewirkt, dass Visual Studio einen Webserver startet. In Visual Studio 2012 handelt es sich dabei um IIS Express. Visual Studio öffnet auch ein Browserfenster mit der Starseite der Anwendung. Indem der Entwickler an den URL im Browser */hotel/index* (bzw. alternativ dazu nur */hotel*, da `index` die standardmäßig verwendete Action-Bezeichnung ist) anhängt, gelangt er zur soeben entwickelten Hotelseite (Abbildung 1-6). Die einzelnen Links führen zwar noch ins Nirvana, aber dieses Problem lässt sich durch das Bereitstellen weiterer Action-Methoden beheben.

Bei Betrachtung der Seite fällt auch auf, dass das Ergebnis der View in eine Layoutseite eingebettet wurde. Diese definiert zum Beispiel die Kopfzeile mit einem Menü oder die Fußzeile mit einem beispielhaften Copyright-Hinweis. Technisch gesehen basiert diese Layoutseite auf dem Projekt Twitter Bootstrap (*http://getbootstrap.com/*). Dabei handelt es sich um eine Sammlung von Scripts und CSS-Layouts, die die Erstellung von Webanwendungen unterstützt, welche sich flexibel an unterschiedliche Bildschirmauflösungen anpassen. Dies wird unter anderem ersichtlich, wenn der Benutzer das Browserfenster, wie in Abbildung 1-7 gezeigt, verkleinert. In diesem Fall ersetzt Bootstrap das Menü durch eine links oben angezeigte Schaltfläche, mit der das Menü bei Bedarf eingeblendet wird. Weitere Informationen zu Twitter Bootstrap finden Sie im dritten Kapitel.

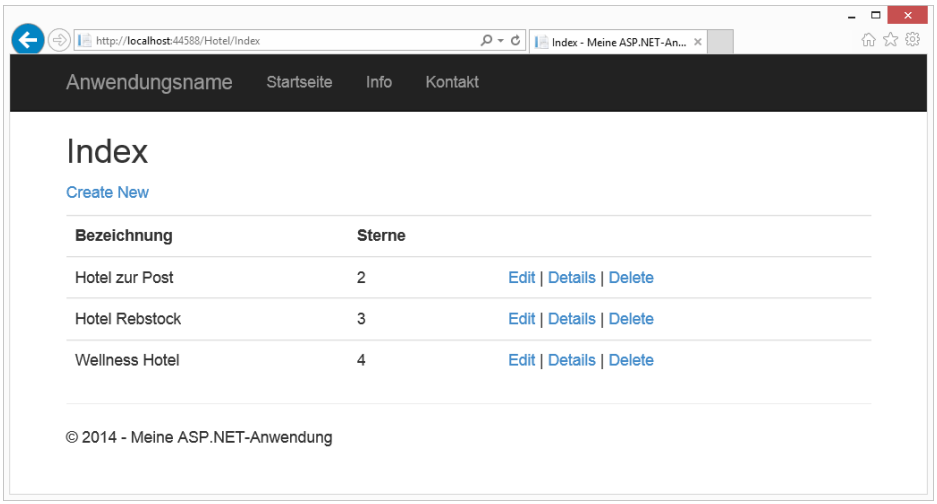


Abbildung 1-6: Demoanwendung in Aktion

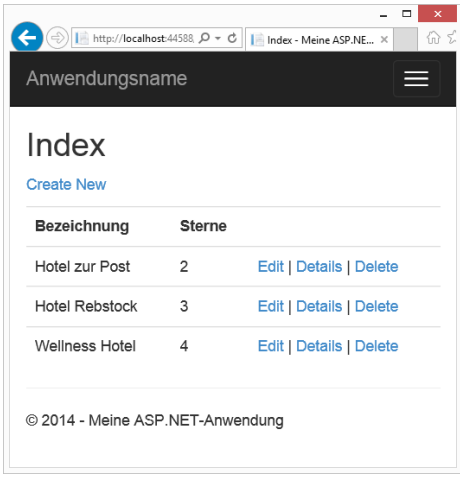


Abbildung 1-7: Automatische Anpassung an Bildschirmauflösung dank Bootstrap

Einträge editieren

Zum Abschluss dieses ersten Rundgangs durch ASP.NET MVC soll noch gezeigt werden, wie die Möglichkeit zum Editieren von Hotels zur hier betrachteten Beispielanwendung hinzugefügt werden kann. Dazu wird zunächst der Controller um zwei Action-Methoden erweitert: Die eine lädt das zu editierende Hotel und bringt es mit einer View innerhalb eines HTML-Formulars zur Anzeige. Die zweite nimmt die Daten aus dem Formular entgegen und aktualisiert damit das Hotel in der statischen Liste.

Beispiel 1-5 zeigt diese beiden Methoden neben der bereits besprochenen Methode `Index`. Beide tragen den Namen `Edit`. Die erste nimmt als Parameter die ID des zu editierenden Hotels entgegen. ASP.NET MVC weist diesem Parameter die ID aus dem URL zu. Dies funktioniert im Übrigen für sämtliche URL-Parameter, deren Namen den Parametern der Action-Methode gleichen. Anschließend lädt die Action-Methode anhand dieser ID das Hotel mithilfe von `HotelRepository` und delegiert anschließend unter Verwendung der geerbten Hilfsmethode `View` an eine View weiter, welche dieses Hotel als Modell übergeben bekommt. Da beim Aufruf der Methode `View` kein Viewname angegeben wird, verwendet ASP.NET MVC den Namen der aktuellen Action-Methode, `Edit`, als Viewname. Somit erwartet das Framework eine View namens `Edit` im Ordner `/Views/Hotel` bzw., falls es dort nicht fündig wird, in `/Views/Shared`.

Die zweite Action-Methode, welche ebenfalls den Namen `Edit` trägt, nimmt eine Instanz von `Hotel` entgegen. Wird diese Methode von ASP.NET MVC angestoßen, versucht es, dieses Hotel aus den übersendeten Parametern, deren Ursprung im HTML-Formular liegt, zusammenzusetzen. Der Entwickler muss sich somit nicht mehr um die wenig spannende, jedoch umso fehleranfälligere Aufgabe des Auslesens und Konvertierens von Parametern kümmern.

Die betrachtete Action-Methode speichert das übergebene Hotel mithilfe von `HotelRepository` und bringt es anschließend erneut zur Anzeige. Über die Eigenschaft `ViewBag.Message` lässt sie der View darüber hinaus eine anzuzeigende Nachricht zukommen.

Bei der Eigenschaft ViewBag handelt es sich um ein dynamisches Objekt, welches der Entwickler nach Belieben mit eigenen Eigenschaften bestücken kann. Verwendung findet es immer dann, wenn neben dem eigentlichen Modell noch weitere Informationen, wie zum Beispiel Vorschlagswerte oder eben Nachrichten, an die View zu übermitteln sind. Als dynamisches Objekt bringt ViewBag sämtliche Vor- und Nachteile der dynamischen Programmierung mit sich. Der Vorteil liegt in der gewonnenen Flexibilität. Der Nachteil liegt in der Umgehung des Typsystems.

Alternativ zur Verwendung der Eigenschaft ViewBag könnte der Entwickler im betrachteten Fall auch ein eigenes View-Modell einrichten, das mit einer Eigenschaft auf das Hotel und mit einer anderen Eigenschaft auf eine eventuell anzuzeigende Nachricht verweisen würde. Generell scheint es eine gute Idee zu sein, sich am Beginn eines Projekts mit seinen Kollegen abzustimmen, ob bzw. wie der Ansichtsbehälter (ViewBag) verwendet werden soll.

Damit ASP.NET MVC entscheiden kann, wann welche der beiden Edit-Methoden aufzurufen ist, wurde die zweite mit dem Attribut HttpPost annotiert. Das bedeutet, dass diese nur dann herangezogen werden soll, wenn der Aufruf über das HTTP-Verb POST erfolgt, was zum Beispiel der Fall ist, wenn Formulardaten übersendet werden. Ruft der Anwender die Seite über GET auf, zum Beispiel indem er einem Link folgt oder die Adresse der Seite direkt eingibt, kommt die andere Methode zur Ausführung. Neben HttpPost stehen analoge Attribute für GET, PUT und DELETE zur Verfügung. Sie nennen sich entsprechend HttpGet, HttpPut bzw. HttpDelete.

Beispiel 1-5: Einfache View zum Anzeigen von Hotels

```
public class HotelController : Controller
{
    // Mapping auf: /Hotel/Index
    //              /Hotel
    public ActionResult Index()
    {
        var rep = new HotelRepository();
        var hotels = rep.FindAll();
        return View("Index", hotels);
        // return View(hotels);
    }
    // Zum Editieren abrufen
    public ActionResult Edit(int id)
    {
        var rep = new HotelRepository();
        var hotel = rep.FindById(id);
        return View(hotel);
    }

    // Zum Speichern des Hotels
    [HttpPost]
    public ActionResult Edit(Hotel h)
    {
        var rep = new HotelRepository();
        rep.Save(h);
        ViewBag.Message = "Hotel wurde gespeichert!";
        return View(h);
    }

}
```

Um die noch fehlende View zu ergänzen, klickt der Entwickler abermals mit der rechten Maustaste in eine der beiden `Edit`-Methoden und wählt anschließend den Kontextmenü-Befehl `Add View`. Als Viewnamen hinterlegt er nun `Edit`, als Modellklasse abermals `Hotel` und als Template wählt er nun `Edit`, sodass das Grundgerüst für ein Formular zum Editieren von Hotels generiert wird.

Nachdem der Entwickler das Dialogfeld bestätigt hat, richtet Visual Studio unter */Views/Hotel* eine View mit dem Namen `Edit.cshtml` ein (Beispiel 1-6). Mit der `model`-Direktive zu Beginn gibt die View an, dass sie als Modell eine `Hotel`-Instanz erwartet. Mit der Hilfsmethode `Html.BeginForm` wird ein `form`-Tag gerendert. Da keine Parameter angegeben werden, erzeugt diese Methode ein `form`-Tag, welches im Zuge des Absendens die erfassten Daten mittels `POST` an jenen URL sendet, von dem aus das Formular ursprünglich abgerufen wurde. Im betrachteten Fall handelt es sich hierbei um den URL */Hotel/Edit*.

Der Entwickler kann durch Überladungen dieser Methode angeben, dass die Daten an einen anderen URL zu senden sind bzw. dass ein anderes HTTP-Verb heranzuziehen ist. Dazu kann er zum Beispiel den Namen eines Controllers bzw. den Namen einer Action-Methode dieses Controllers angeben.

Darüber hinaus besteht die Möglichkeit, URL-Parameter in Form eines dynamischen Objekts anzuhängen. Generell empfiehlt sich ein Blick auf die unterschiedlichen Überladungen der zur Verfügung stehenden Hilfsmethoden. Da `Html.BeginForm` innerhalb einer `using`-Anweisung Verwendung findet, wird am Ende dieser das schließende `form`-Tag gerendert. Alternativ dazu könnte der Entwickler an der gewünschten Stelle auch auf die Methode `Html.EndForm` zurückgreifen.

Innerhalb des Formulars rendert die View mithilfe der Methode `Html.ValidationSummary` eine eventuelle Validierungsfehlermeldung. Erfasst der Anwender zum Beispiel für die Eigenschaft Sterne einen Wert, der nicht nach `int` konvertiert werden kann, ergibt sich ein Validierungsfehler, der von der betrachteten Methode ausgegeben wird. Indem die View `true` übergibt, zeigt sie an, dass sie lediglich Fehlermeldungen anzeigen soll, die sich auf das gesamte Modell, nicht aber nur auf einzelne Eigenschaften beziehen. Für die Ausgabe von feldbezogenen Fehlermeldungen kommt hingegen unmittelbar neben den einzelnen Eingabefeldern die Hilfsmethode `Html.ValidationMessageFor` zum Einsatz. Weitere Informationen zum Validieren von Eingaben finden sich im Abschnitt »Validieren von Benutzereingaben«.

Mit `Html.HiddenFor` rendert die View ein verstecktes Feld, in dem die `HotelId` platziert wird. Somit kann die `HotelId` zwar nicht editiert werden, jedoch wird sie beim Absenden des Formulars mit den anderen Parametern an die adressierte `Action`-Methode übertragen, sodass diese weiß, welches Hotel zu aktualisieren ist. `Html.LabelFor` rendert Beschriftungsfelder und `Html.EditorFor` Eingabefelder für die einzelnen Eigenschaften. Standardmäßig erzeugt `Html.EditorFor` Kontrollkästchen für boolesche Eigenschaften sowie Eingabefelder in allen anderen Fällen. Allerdings kann der Entwickler für bestimmte Eigenschaften oder auch nur für bestimmte Datentypen Vorlagen definieren, die `EditorFor` verwendet. Somit kann er sicherstellen, dass dieselben Felder immer auf dieselbe Weise editiert werden können. Mehr dazu findet sich im Abschnitt »Vorlagen für Felder und Models«.

Neben `EditorFor` stehen noch weitere Hilfsmethoden zur Verfügung, mit denen der Entwickler den Feldtyp des zu rendernden Felds bestimmten kann. Neben der bereits beschriebenen Methode `HiddenFor` sind dies `TextBoxFor`, `TextAreaFor`, `CheckBoxFor`, `RadioButtonFor`, `DropDownListFor` und `ListBoxFor`. Die Namen dieser Methoden korrelieren mit jenen HTML-Tags, welche sie rendern. Darüber hinaus existieren auch Varianten dieser Methoden ohne die Endung `For`. Diese bieten dieselben Möglichkei-

ten, erwarten jedoch die anzuzeigende Modelleigenschaft in Form eines Strings. Da in diesem Fall der Compiler nicht zur Prüfung verwendet werden kann, werden diese Varianten in der Regel gemieden.

Steht der Wert eines Eingabefelds sowohl über das Model als auch in Form eines via POST übersendeten Parameters zur Verfügung, wählen die HTML-Helper zum Rendern von Eingabefeldern letztere Option. Dies ist häufig das gewünschte Verhalten, vor allem dann, wenn der Benutzer ein Formular absendet und seine Eingaben daraufhin erneut anzuzeigen sind. Darüber hinaus erlaubt dieses Verhalten in Kombination mit der Validierung von Eingaben, dass der Benutzer jene Daten, die er vor dem Absenden des Formulars eingetragen hat und die nicht korrekt validiert werden konnten, nochmals zur Korrektur präsentiert bekommt.

Ist der Entwickler mit diesem Verhalten nicht einverstanden, steht es ihm frei, das Eingabefeld manuell zu rendern:

```
<input name="x" value="@Model.Bezeichnung" />
```

Am Ende definiert die View einen Submit-Button, einen Link, der zurück zur Übersicht aller Hotels führt, sowie eine Sektion, die den Namen `Scripts` trägt. Sektionen sind benannte Seitenfragmente, die innerhalb der Layoutseite an einer wohldefinierten Stelle eingefügt werden.

Die standardmäßig von der Projektvorlage `Internet-Application` eingerichtete Layoutseite rendert zum Beispiel am Ende die Sektion `Scripts`, welche per Definition JavaScript-Anweisungen beinhaltet. Mit `@Scripts.Render` wird ein `Script`-Tag erzeugt, welches verschiedene JavaScript-Dateien einbindet. Im betrachteten Fall dienen diese der Validierung von Eingaben.

Beispiel 1-6: View zum Editieren eines Hotels

```
@model FirstSteps.Models.Hotel

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Hotel</legend>

        @Html.HiddenFor(model => model.HotelId)

        <div class="editor-label">
            @Html.LabelFor(model => model.Bezeichnung)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Bezeichnung)
            @Html.ValidationMessageFor(model => model.Bezeichnung)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Sterne)
        </div>
        <div class="editor-field">
```

```
            @Html.EditorFor(model => model.Sterne)
            @Html.ValidationMessageFor(model => model.Sterne)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Um die über `ViewBag` übertragene Nachricht in der View anzuzeigen, fügt der Entwickler noch das Quellcodefragment aus Beispiel 1-7 an der gewünschten Position ein.

Beispiel 1-7: Zugriff auf ViewBag über View

```
@if (ViewBag.Message != null)
{
    <p>
        <b>@ViewBag.Message</b>
    </p>
}
```

Die meisten HTML-Hilfsmethoden bieten eine Überladung, welche ein Dictionary mit HTML-Attributen und deren Werten entgegennimmt. Diese HTML-Attribute werden samt ihrer Werte in das generierte Markup übernommen. Um den Schreibaufwand zu reduzieren, bieten diese Hilfsmethoden auch noch eine weitere Überladung, welche anstatt des Dictionaries ein Objekt entgegennimmt. Die Idee dahinter ist, dass der Entwickler an diese Parameter ein anonymes Objekt übergibt, wobei die Eigenschaften dieses Objekts in Form von HTML-Attributen gerendert werden.

Beispielsweise führt der Aufruf von

```
@Html.LabelFor(model => model.Sterne, new { style="color:red" })
```

dazu, dass `LabelFor` den folgenden Markup rendert:

```
<label for="Sterne" style="color:red">Sterne</label>
```

Vorschlagswerte über Dropdown-Listenfelder anbieten

Dieser letzte Abschnitt dieses ersten Streifzugs durch die Möglichkeiten von ASP.NET MVC zeigt, wie Vorschlagswerte über Dropdown-Listenfelder bereitgestellt werden können. Im Zuge dessen wird das Textfeld zum Erfassen der Eigenschaft `Sterne` durch ein Dropdown-Listenfeld ersetzt.

Damit ASP.NET MVC beim Rendern von Dropdown-Listenfeldern unterstützen kann, muss der Entwickler die Vorschlagswerte in Form einer Auflistung mit `SelectListItem`-Instanzen bereitstellen. Beispiel 1-8 verwendet zum Aufbereiten dieser Auflistung die private Methode `CreateSterne`. Darüber hinaus wurden beide `Edit`-Methoden erweitert, sodass diese Vorschlagswerte über `ViewBag` an die View übermittelt werden. Dabei ist darauf zu achten, dass die Eigenschaft im Ansichtsbehälter nicht densel-