

## Kapitel 10

# ASP.NET MVC 2

### **In diesem Kapitel:**

Architektur	414
Erste Schritte mit ASP.NET MVC	415
Controller	418
Views	425
Validieren von Benutzereingaben	428
Areas	431
Filter	433
Exkurs: MVC und Dependency Injection	435
Zusammenfassung und Fazit	445

Mit ASP.NET MVC stellt Microsoft eine Alternative zu den ASP.NET WebForms, welche sich am bekannten Muster Model-View-Controller (MVC) orientieren, zur Verfügung. Vor .NET 4.0 konnte dieses Framework über *Codeplex* [COD] bezogen werden; ab .NET 4.0 findet es sich in der Version 2 im Lieferumfang wieder. Da dieses Framework somit erst ab Version 4.0 im Lieferumfang von .NET enthalten ist, bietet dieses Kapitel eine Einführung anstatt lediglich auf die Neuerungen aus Version 2 einzugehen. Leser, welche bereits mit Version 1 vertraut sind, finden die Neuerungen in Version 2 in den folgenden Abschnitten: »Asynchrone Controller«, »Unterstützung beim Anlegen von Views durch Visual Studio«, »Deklaratives Validieren« und »Areas«.

## Architektur

Bevor die ersten Codebeispiele folgen, beschreibt dieser Abschnitt die sich hinter dem Muster MVC verbergenden Überlegungen. Darüber hinaus wird auf das Muster MVVM (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. Das Model entsprach dabei ursprünglich den fachlichen Daten sowie den darauf operierenden Routinen. Da diese beiden Aspekte heutzutage in der Regel getrennt werden, wird das Model 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 Model 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.

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 Model, 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 Model, welches sich ggf. auf einen O/R-Mapper, wie ADO.NET Entity Framework, abstützt. Da die gesamte Logik durch den Controller widerspiegelt wird, wird auch das Testen sowie das Automatisieren von Tests erleichtert.

## Überblick zu MVVM (Model-View-ViewModel)

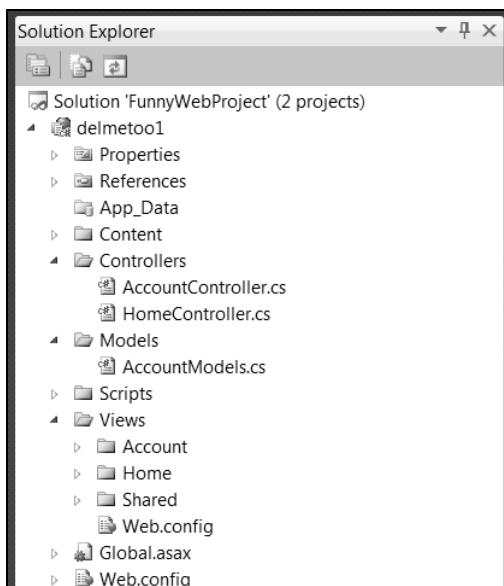
Da dieselben fachlichen Daten in unterschiedlichen Views häufig unterschiedlich angezeigt werden, sieht das Muster *MVVM (Model-View-ViewModel)* vor, dass jede View ein eigenes Model erhält, welches als *ViewModel* bezeichnet wird. Dieses stützt sich auf ein oder mehrere Models ab und bereitet deren 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 das Verarbeiten der Daten (Validieren, Speichern, Laden, Berechnungen) kümmern oder die damit verbundenen Verarbeitungsvorgänge zumindest anstoßen, indem sie an die entsprechenden Klassen weiterdelegieren.

## Erste Schritte mit ASP.NET MVC

Als Einführung in ASP.NET MVC-Framework verwendet dieser Abschnitt ein einfaches Beispiel mit einem Controller, welcher lediglich zwei Zahlen entgegennimmt und addiert sowie einer View, welche das Ergebnis präsentiert. Auf die Verwendung einer View wird in diesem ersten Beispiel zur Vereinfachung verzichtet.

### ASP.NET MVC-Projekt anlegen

Zum Anlegen einer ASP.NET MVC-Applikation kann in Visual Studio 2010 die Vorlage *ASP.NET MVC 2 Web Application* herangezogen werden. Diese beinhaltet bereits ein einfaches Projekt sowie die von ASP.NET MVC standardmäßig erwartete Verzeichnisstruktur, welche in Abbildung 10.1 zu sehen ist.



**Abbildung 10.1** Verzeichnisstruktur eines ASP.NET MVC-Projekts

Dabei fallen drei Ordner auf: Der Ordner *Controller* beinhaltet die einzelnen Controller-Klassen, welche auf *Controller* enden, und *Models* die einzelnen Model-Implementierungen. 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. Die einzige Ausnahme ist der Ordner *Shared* – er beinhaltet Views, welche von sämtlichen Controllern verwendet werden können.

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, welche keine gemeinsamen Charakteristika, wie gemeinsame Basisklassen, aufweisen.

## Controller anlegen

Listing 10.1 zeigt eine einfache Implementierung eines Controllers. Es handelt sich dabei um eine Klasse, welche von *Controller* erbt. Alternativ dazu könnte auch von *ControllerBase* geerbt oder *ApiController* implementiert werden. Die Implementierung beinhaltet eine Methode, welche zwei Parameter vom Typ `int` entgegennimmt. Die Methoden von *Controller* werden als Action-Methoden bezeichnet und auf URLs gemappt. Standardmäßig ist jede Action-Methode über die URL *Controllername/Methodenname* erreichbar, wobei *Controllername* für den Namen der Controllerklasse ohne das Suffix *Controller* und *Methodenname* für den Namen der Actionmethode steht. Die beim Aufruf der URL übergebenen Parameter werden den einzelnen Methodenparametern zugewiesen, sofern Namensgleichheit herrscht. Der zurückgegebene Wert, welcher in der Regel vom Typ `ActionResult` ist, informiert das Framework über Art der Darstellung der ermittelten Daten. Häufig wird auf diese Weise auf eine View verwiesen – es bestehen aber auch andere Möglichkeiten, wie das Rendern eines Objekts nach JSON, das Verweisen auf eine herunterzuladende Datei oder die Angabe eines Strings, welcher das gesamte Ergebnis der Anfrage darstellen soll.

---

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

---

Der betrachtete Controller addiert die übergebenen Werte und hinterlegt diese in einem geerbten Dictionary<sup>1</sup> mit dem Namen `ViewData` unter dem Schlüssel `result`. Da auch die Views auf dieses Dictionary Zugriff haben, kann es zum Weiterreichen von Informationen und somit als nicht typsicherer Ersatz für Models verwendet werden.

Zum Abschluss wird die geerbte Methode `View` verwendet, um ein `ActionResult`, welches auf eine View verweist, zu erzeugen. Da an dieser Stelle kein View-Name übergeben wird, wird eine View mit dem Namen der Action-Methode (hier: `Add`) innerhalb des View-Ordners mit dem Namen des Controllers (hier: `Calc`) erwartet.

---

<sup>1</sup> Genau genommen handelt es sich hierbei um eine Eigenschaft vom Typ `ViewDataDictionary`, welcher die Schnittstelle `IDictionary` implementiert.

```
public class CalcController : Controller
{
    public ActionResult Add(int a, int b)
    {
        int result = a + b;
        ViewData["result"] = result;
        return View();
    }
}
```

**Listing 10.1** Einfacher Controller

## View anlegen

Eine View für den in Listing 10.1 gezeigten Controller findet sich in Listing 10.2. Es handelt sich dabei um eine ASP.NET-Seite, welche von `ViewPage` aus dem Namensraum `System.Web.Mvc` erbt und sich auf eine durch Visual Studio beim Anlegen des Projekts erzeugte Master Page abstützt. Im unteren Teil erfolgt die Ausgabe des vom Controller ermittelten Ergebnisses. Dazu wird auf den im Dictionary `ViewData` hinterlegten Eintrag zugegriffen.

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

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Add
</asp:Content>

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

    <h2>Add</h2>

    Ergebnis: <%=ViewData["result"]%>

</asp:Content>
```

**Listing 10.2** Einfache View

Visual Studio 2010 unterstützt beim Erzeugen sowie Öffnen einer View für eine Action-Methode, indem im Kontextmenü des Editierfensters die Befehle *Add View* und *Go To View* angeboten werden, wenn sich der Cursor innerhalb einer Action-Methode befindet.

---

**HINWEIS** ASP.NET MVC-Neulinge glauben häufig ein Antipattern in Views zu entdecken, da diese mittels `<%=...%>` Informationen direkt in die Seite schreiben. Dies mag daran liegen, dass diese Schreibweise in ASP.NET WebForms verpönt ist, da hier zum einen Steuerelemente zum Rendern der Seiten vorgesehen sind und zum anderen dieses Konstrukt mit dem Einbetten von Geschäftslogik in Websites assoziiert wird, was zulasten der angestrebten Trennung zwischen Logik und Präsentation geht. Bei ASP.NET MVC wird dem allerdings bereits durch die strikte Trennung zwischen Model, View und Controller Rechnung getragen und die diskutierte Schreibweise für Aspekte der Präsentationslogik (nicht Geschäftslogik), wie dem Iterieren und Ausgeben von Daten, verwendet.

---

**HINWEIS** Aufgrund des modularen Designs von ASP.NET MVC besteht auch die Möglichkeit, alternativ zu ASP.NET Websites andere View-Technologien, wie zum Beispiel *NVelocity* [NVE], einzusetzen. In der Praxis findet diese Möglichkeit allerdings nur wenig Verwendung.

## Controller

Nachdem der prinzipielle Aufbau einer ASP.NET MVC-Applikation besprochen wurde, zeigt dieser Abschnitt einige weiterführende Möglichkeiten für die Implementierung von Controllern.

### Models weiterreichen

Im einführenden Beispiel wurde zur Vereinfachung sowie zulasten der Typsicherheit auf die Verwendung eines Models verzichtet. Dies soll nun nachgeholt werden. Listing 10.3 zeigt eine Action-Methode `Edit`, welche das Editieren von Partys erlaubt. Als Parameter wird die Id der zu editierenden Party als nullable int (`int?`) erwartet. Ist dieser Parameter null, wird eine neue Party angelegt, ansonsten wird die Party mit der angegebenen Id geladen. Da die Party als Model an die View weitergereicht werden soll, wird sie zu `ViewData.Model` zugewiesen. Anschließend werden zusätzlich einige Veranstalter von Partys als Vorschlagswerte im Dictionary `ViewData` hinterlegt und an die View weiterdelegiert. Alternativ zur Zuweisung an die Eigenschaft `Model` von `ViewData` besteht auch die Möglichkeit, das Model als Parameter an die Methode `View` zu übergeben.

```
public ActionResult Edit(int? id)
{
    Party party;

    if (id == null)
    {
        party = new Party();
    }
    else
    {
        party = [...]; // Party laden
    }

    ViewData.Model = party;
    ViewData["Veranstalter"] = [...]; // Vorschlagswerte laden

    return View();
    // return View(party);
}
```

**Listing 10.3** Controller, welcher ein Model an eine View weiterreicht

## Models entgegennehmen

Die einzelnen beim Aufruf übergebenen Parameter können verwendet werden, um eine Instanz eines Models zu erzeugen. Möglich wird dies durch den so genannten *Model Binder*. Listing 10.4 demonstriert dies.

```
[HttpPost]
public ActionResult Edit(Party p)
{
    // Party speichern
    return View();
}
```

**Listing 10.4** Controller, welcher ein Model entgegennimmt

Um zu bestimmen, welche Parameter an die Eigenschaften des Models gebunden werden sollen, kann der jeweilige Parameter mit dem Attribut `Bind` annotiert werden. Listing 10.5 demonstriert dessen Verwendung, indem mit der Eigenschaft `Exclude` angegeben wird, dass die `PartyId` beim Binden der Parameter nicht berücksichtigt werden soll. Sollen mehrere Eigenschaften ausgeschossen werden, können diese durch Beistriche getrennt angeführt werden. Als Alternative zu `Exclude` steht auch eine Eigenschaft `Include` zur Verfügung. Wird diese verwendet, werden nur die angeführten Eigenschaften gebunden. Darüber hinaus kann mit `Prefix` angegeben werden, dass die zu bindenden Parameter einen bestimmten Prefix aufweisen müssen.

```
public ActionResult Edit([Bind(Exclude = "PartyId")]Party p) { [...] }
```

**Listing 10.5** Model-Binding beeinflussen

Nicht immer ist es wünschenswert, eine neue Model-Instanz von ASP.NET MVC erstellen zu lassen. In manchen Situationen ist es beispielsweise notwendig, die Model-Instanz aus einer Datenbank zu laden, um sie anschließend mit den übertragenen Parametern zu aktualisieren. In diesen Fällen kann die Auflistung `FormCollection` als Parameter verwendet werden. ASP.NET MVC überträgt sämtliche beim Auftrag übertragenen Parameter in diese Auflistung. In weiterer Folge können diese mit der geerbten Methode `TryUpdateModel` von `FormCollection` in ein beliebiges Objekt übertragen werden (vgl. Listing 10.6). Daneben existieren noch weitere Überladungen von `TryUpdateModel`, welche die Angabe der vom Attribut `Bind` bekannten Parameter `Exclude`, `Include` und `Prefix` erlauben.

```
[HttpPost]
public ActionResult Edit(FormCollection fc)
{
    Party p = [...];
    this.TryUpdateModel(p, fc);
    [...]
}
```

**Listing 10.6** Manuelles Anstoßen des Model Binders

# Action-Methoden überladen

Das Auflösen überladener Action-Methoden wird von ASP.NET MVC nicht unterstützt. Jeder Aufruf muss mit genau einer Methode assoziiert werden können. Die übergebenen Parameter werden bei der Auswahl der zu verwendenden Methode nicht berücksichtigt. Existieren mehrere Überladungen, kann jedoch jede Überladung mit einem HTTP-Verb (GET, POST, PUT, DELETE) assoziiert werden. Dazu stehen entsprechende Attribute zum Annotieren der einzelnen Überladungen zur Verfügung: `HttpGet`, `HttpPost`, `HttpPut`, `HttpDelete`. Dies erlaubt auch die Implementierung von einfachen REST-basierenden Services. Als Beispiel dazu zeigt Listing 10.7 die Rümpfe der Action-Methoden aus Listing 10.3 und Listing 10.4. Beide weisen den Namen `Edit` auf. Die erste erwartet die `Id` der zu editierenden Party, um sie für den Editiervorgang zu laden; die zweite erwartet am Ende des Editiervorgangs die aktualisierte Party zum Ablegen in der Datenbank. Da diese mit `HttpPost` annotiert wurde, wird sie nur bei Verwendung des HTTP-Verbs POST herangezogen. In allen anderen Fällen wird die erste Variante verwendet.

```
public ActionResult Edit(int? id) { [...] }

[HttpPost]
public ActionResult Edit(Party p) { [...] }
```

Listing 10.7    Angabe von HTTP-Verben

# View auswählen

Bis jetzt wurde die Methode `View` verwendet, um ein `ActionResult` zu erzeugen, mit dem angezeigt wurde, dass an die View mit dem Namen der Action-Methode weiterdelegiert werden soll. Soll eine andere View Verwendung finden, kann, wie in Listing 10.8 gezeigt, deren Name als Parameter an `View` übergeben werden. Daneben existiert auch eine Überladung, welche den View-Namen sowie das zu verwendende Model entgegennimmt. Die angegebene View muss sich entweder im View-Ordner des Controllers oder im View-Ordner `Shared` befinden.

```
public ActionResult Create()
{
    [...]
    return View("Details");
}
```

Listing 10.8    Angabe der gewünschten View

Neben der Methode `View` existieren noch weitere Methoden, mit welchen das Ergebnis eines Aufrufs beeinflusst werden kann. All diese haben gemeinsam, dass sie eine Instanz einer Subklasse von `ActionResult`, welche zum gewünschten Verhalten führt, zurückliefern. Tabelle 10.1 informiert über diese Methode.

Methode	Beschreibung
Content	Liefert den angegebenen String als Ergebnis der Anfrage zurück
File	Veranlasst einen Download als Ergebnis
JavaScript	Liefert den angegebenen String als JavaScript-Code, welcher am Client ausgeführt wird, zurück



Methode	Beschreibung
Json	Liefert das angegebene Model JSON-formatiert zurück
PartialView	Verwendet eine partielle View, um einen Teil einer Seite zu rendern
Redirect	Leitet die aktuelle Anfrage an die angegebene URL um
RedirectToAction	Leitet die aktuelle Anfrage an die angegebene Action-Methode um
RedirectToRoute	Leitet die aktuelle Anfrage an jene URL um, welche zur angegebenen Route passt

**Tabelle 10.1** Methoden zum Beeinflussen des Ergebnisses

Es besteht auch die Möglichkeit, für eventuell ausgelöste Ausnahmen eine View zu definieren, welche auf den aufgetretenen Fehler hinweist. Dazu wird, wie in Listing 10.9 demonstriert, mit dem Attribut `HandleError` der Typ einer Ausnahme auf eine View gemappt. Wird dieses Attribut auf eine Controller-Klasse angewandt, gilt es für jede einzelne Action-Methode dieser Klasse.

```
[HandleError(ExceptionType=typeof(ArgumentException), View="Error")]
public ActionResult List() { [...] }
```

**Listing 10.9** Festlegen von Views für Ausnahmen

## URL-Mapping beeinflussen

Action-Methoden werden auf URLs gemappt. Dazu werden in der Datei *global.asax* innerhalb der Methode `RegisterRoutes`, welche von `Application_Start` aufgerufen wird, Routen definiert. Listing 10.10 zeigt die Definition der standardmäßig vorhandenen Route. Beim ersten Parameter handelt es sich um den Namen der Route, beim zweiten um die Route inkl. Platzhalter für die Namen des Controllers und der Action-Methode. Darüber hinaus findet sich am Ende der Route ein Parameter `id`. Der beim Aufruf für diesen Parameter angegebene Wert wird zu einem eventuell vorhandenen gleichnamigen Parameter der Action-Methode zugewiesen. Ein Aufruf von `Party/Edit/7` führt somit beispielsweise dazu, dass die Action-Methode `Edit` eines Controllers mit dem Namen `PartyController` aufgerufen wird, wobei – sofern vorhanden – an den Parameter `id` dieser Action-Methode der Wert `7` zugewiesen wird. Beim dritten Parameter handelt es sich im betrachteten Beispiel um ein anonymes Objekt, welches die Standardwerte der in der Route definierten Parameter angibt. Daraus ist ersichtlich, dass standardmäßig ein Leerstring als `Id`, die Action-Methode `Index` sowie der Controller `Home` angenommen wird, sofern diese Daten beim Aufruf weggelassen werden.

```
public static void RegisterRoutes(RouteCollection routes)
{
    [...]
    routes.MapRoute(
        "Default",                                     // Route name
        "{controller}/{action}/{id}",                  // URL with parameters
        new { controller = "Home", action = "Index", id = "" } // Parameter defaults
    );
}
```

**Listing 10.10** Standard-Route

Ein Beispiel einer benutzerdefinierten Route für die im Einführungsbeispiel verwendete Methode `add` (Listing 10.1) findet sich in Listing 10.11. Sie erlaubt es, die Parameter `a` und `b` direkt in der URL durch Beistriche getrennt anzugeben. Für den Controller sieht diese Route keinen Parameter in der URL vor. Deswegen muss dieser zwingend im Rahmen der Standardwerte angegeben werden. Darüber hinaus werden mit dem vierten Parameter von `MapRoute` die gültigen Werte für die einzelnen Parameter in der URL durch Angabe regulärer Ausdrücke eingeschränkt. Der reguläre Ausdruck `\d+` erwartet beliebig viele Ziffern, jedoch mindestens eine. Ein auf diese Route passender Aufruf wäre zum Beispiel `Calculator/Add/1,2`.

```
routes.MapRoute(
    "CalcRoute",
    "Calculator/{action}/{a},{b}",
    new { controller="Calc" },
    new { a=@"\d+", b=@"\d+" }
);
```

**Listing 10.11** Benutzerdefinierte Route

Ein weiteres Beispiel einer benutzerdefinierten Route findet sich in Listing 10.12. Hier wird ein Parameter nach dem Schema `{*Parametername}` definiert. Das bedeutet, dass die gesamte restliche URL an den definierten Parameter zugewiesen werden soll. Ein Beispiel für einen zu dieser Route passenden Aufruf ist `Calculator/Interpreter/emhoch2/runden`. Dieser Aufruf würde bewirken, dass der Wert `emhoch2/runden` an den Parameter `restlicheUrl` übergeben wird.

```
routes.MapRoute(
    "CalcRoute2",
    "Calculator/Interpreter/{*restlicheUrl}",
    new { controller = "Calc", action = "Interpretiere" }
);
```

**Listing 10.12** Eine weitere benutzerdefinierte Route

In manchen Situationen möchte man den Routingmechanismus ganz umgehen, und die angeforderte Adresse auf die ursprüngliche Weise durch die ASP.NET-Laufzeitumgebung interpretieren lassen. In diesen Fällen kann die Methode `IgnoreRoute` herangezogen werden. Listing 10.13 demonstriert den Einsatz dieser Methode. Der gezeigte Aufruf, welcher sich standardmäßig in der Methode `RegisterRoutes` befindet, bewirkt, dass Aufrufe von Dateien mit der Endung `.axd` nicht geroutet werden.

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

**Listing 10.13** Routen ignorieren

## Zugang zu Action-Methoden beschränken

Da ASP.NET MVC auf der mit ASP.NET geschaffenen Infrastruktur aufsetzt, können auch die bekannten Möglichkeiten in Hinblick auf Security verwendet werden. Darüber hinaus kann mit dem Attribut `Authorize` der Zugang zu einer Action-Methode für bestimmte Benutzerkreise eingeschränkt werden. Ein Beispiel für die Verwendung dieses Attributs findet sich in Listing 10.14. Hier wird der Zugang zur Methode `Decide` auf Benutzer der Rollen `Chef` oder `Boss` eingeschränkt. Alternativ dazu könnte der Zugang auf bestimmte Benut-

zer eingeschränkt werden, indem diese mit der Eigenschaft `Users` angeführt werden. Werden weder Rollen noch Benutzernamen angeführt, kann die Methode von allen möglichen Benutzern, die allerdings bei der Applikation angemeldet sein müssen, aufgerufen werden.

```
[Authorize(Roles = "Chef,Boss")]
public ActionResult Decide()
{
    return Content("Hier könnte Ihre Entscheidung stehen ...");
}
```

**Listing 10.14** Einschränken des Zugriffs auf eine *Action*-Methode

Auf welche Rollen und Benutzernamen sich diese Angaben beziehen, hängt von den konfigurierten Sicherheitseinstellungen ab. Wird beispielsweise Windows-Authorisierung verwendet, so beziehen sich die Angaben auf Benutzernamen und Gruppen aus dem jeweiligen Active Directory bzw. vom jeweiligen Rechner

---

**HINWEIS** Die durch Visual Studio 2010 bei Verwendung der Vorlage *ASP.NET MVC 2 Web Application* generierte Webapplikation beinhaltet einen Controller *AccountController* mit dazugehörigen Models und Views, welche sich auf die Möglichkeiten des Membership-Providers stützen und sich als Ausgangsbasis für eigene Anmelde- und Autorisierungsstrategien eignen.

---

## Asynchrone Controller

Für das Abarbeiten von Anfragen bekommen Applikationen, welche in IIS ausgeführt werden, einen Thread aus einem Threadpool zugewiesen. Um zu verhindern, dass langlaufende Anfragen diese Threads blockieren, besteht die Möglichkeit, einen eigenen Thread zur Abarbeitung der Action-Methode abzuspalten. Nachdem dieser die ihm zugewiesenen Aufgaben vollendet hat, muss das Framework benachrichtigt werden, damit es einen neuen Thread aus dem Threadpool, welcher zum Beispiel eine View rendert und diese als Antwort zum Aufrufer sendet, anfordern kann. Controller, welche solche Methoden anbieten, werden als asynchrone Controller bezeichnet. Diese erben von `AsyncController` und weisen für asynchrone Anfragen ein Methoden-Paar `XXXAsync` und `XXXCompleted` auf, wobei XXX einen Platzhalter für einen frei zu wählenden Namen darstellt und die erste am Beginn der Abarbeitung aufgerufen wird und die Aufgabe hat, einen Thread abzuspalten sowie die letzte nach Abarbeitung der Aufgabe durch den Thread aufrufen wird, um ein geeignetes `ActionResult` auszuwählen. Da lediglich für diese beiden Methoden Threads aus dem Threadpool von IIS herangezogen werden, werden diese von der eigentlichen langlaufenden Aufgabe, welche von einem eigenen Thread ausgeführt wird, nicht blockiert und können somit zum Bedienen weiterer Anfragen verwendet werden.

Damit die Laufzeitumgebung weiß, wann `XXXCompleted` ausgeführt werden kann, inkrementiert `XXXAsync` einen internen Zähler für jeden abgespaltenen Thread. Die einzelnen Threads dekrementieren diesen Zähler nach Vollendung ihrer Aufgaben wieder, sodass dieser den Ursprungswert (0) aufweist, wenn alle Threads beendet wurden. Dies zeigt der Laufzeitumgebung an, dass auch die asynchrone Abarbeitung der Anfrage fertiggestellt wurde, woraufhin `XXXCompleted` angestoßen wird.

Listing 10.15 zeigt die Implementierung eines asynchronen Controllers. `AddAsync` nimmt zwei zu addierende Integer entgegen und erhöht den internen Zähler durch Aufruf von `AsyncManager.OutstandingOperations.Increment` auf den Wert 1. Anschließend werden die beiden Parameter in einem Object-Array verpackt. Dies

ist notwendig, da an Threads lediglich ein einziger Parameter übergeben werden kann. Danach wird ein neuer Thread, welcher die Methode `DoStuff` ausführt, abgespalten. Diese holt die beiden zu addierenden Werte aus dem übergebenen Object-Array, simuliert eine langlaufende Aufgabe durch Aufruf von `Thread.Sleep` und addiert anschließend die beiden Werte. Das Ergebnis wird im Dictionary `AsyncManager.Parameters` abgelegt und der interne Zähler wird wieder dekrementiert. Da der interne Zähler somit wieder den Wert 0 aufweist, führt das Framework die zur aufgerufenen Action-Methode passende Completed-Methode, die im betrachteten Beispiel den Namen `AddCompleted` trägt, aus. Die einzelnen Parameter dieser Methode werden mit den Werten aus dem zuvor verwendeten Dictionary `AsyncManager.Parameters` bestückt.

```
public class AsyncSampleController : AsyncController
{
    public void AddAsync(int a, int b)
    {
        AsyncManager.OutstandingOperations.Increment();
        object parameter = new object[] { a, b };

        Thread t = new Thread(new ParameterizedThreadStart(DoStuff));
        t.Start(parameter);
    }

    public ActionResult AddCompleted(int result)
    {
        ViewData["result"] = result;
        return View();
    }

    public void DoStuff(object o)
    {
        object[] parameter = (object[])o;
        int a = (int)parameter[0];
        int b = (int)parameter[1];

        Thread.Sleep(10000);

        int result = a + b;
        AsyncManager.Parameters["result"] = result;
        AsyncManager.OutstandingOperations.Decrement();
    }
}
```

**Listing 10.15** Implementierung eines asynchronen Controllers

## Caching von Aufrufen

ASP.NET MVC bietet die Möglichkeit, das Ergebnis von Aufrufen zu cachen. Dazu können, wie in Listing 10.16 demonstriert, Action-Methoden mit dem Attribut `OutputCache` annotiert werden. Die an `OutputCache` übergebenen Parameter bestimmen das Caching-Verhalten. Diese entsprechen jenen, die auch *ASP.NET Web Forms* für denselben Zweck anbietet, und werden deshalb hier nicht näher beschrieben.

```
[OutputCache(Duration=5, VaryByParam="")]
public ActionResult List() { [...] }
```

**Listing 10.16** Verwendung von *OutputCache*

# Views

Dieser Abschnitt beschreibt weiterführende Techniken für die Entwicklung von Views.

## Typisierte Views

Damit auf die an Views weitergereichten Models in typischerer Manier zugegriffen werden kann, besteht die Möglichkeit zur Typisierung der Klasse `ViewPage`, welche als Basisklasse für Views dient. Der angegebene Typ wird zur Typisierung der Eigenschaft `Model`, welche das vom Controller definierte Model beinhaltet, verwendet. Listing 10.17 demonstriert dies, indem in der `Pagedirektive` mit dem Attribut `Inherits` angezeigt wird, dass die geerbte `ViewPage` mit dem Typ `IEnumerable<Party>` typisiert werden soll. Durch diese Typisierung kann das Model in weiterer Folge mit einer `ForEach`-Schleife durchlaufen und auf eine typischere Art auf deren Einträge sowie auf die Member dieser zugegriffen werden.

```
<%@ Page Language="C#" MasterPageFile="[" Inherit="System.Web.Mvc.ViewPage<IEnumerable<Party>>" %>
[...
<% foreach (var item in Model) { %>
    <tr>
        <td>
            <%= item.Bezeichnung %>
        </td>
    </tr>
<% } %>
```

**Listing 10.17** Beispiel einer typisierten View

## Unterstützung beim Anlegen von Views durch Visual Studio

Visual Studio unterstützt mit dem in Abbildung 10.2 gezeigten Dialogfeld bei der Erstellung von Views. In den Genuss dieses Dialogfelds kommt man durch Auswahl des Befehls *Add View*, welcher sich im Kontextmenü des Ordners *View* oder einem dessen Unterordnern im *Solution Explorer* befindet. Alternativ dazu wird derselbe Befehl im Kontextmenü des Editierfensters angeboten, sofern sich der Cursor innerhalb einer Action-Methode befindet. Dieses Dialogfeld bietet unter anderem die Möglichkeit anzugeben, dass die zu erzeugende View streng typisiert sein soll (*Create a strongly-typed view*). Wird diese Option aktiviert, kann in weiterer Folge unter *View data class* jene Klasse, welche als Model und somit auch für die Typisierung der View herangezogen werden soll, ausgewählt werden. Zusätzlich kann unter *View Content* angeführt werden, ob die View zum Auflisten von Datensätzen oder zum Anzeigen, Hinzufügen, Editieren oder Löschen eines Datensatzes vom Typ des Models verwendet werden soll. Diese Information wird genutzt, um ein erstes Grundgerüst für die View zu erstellen.

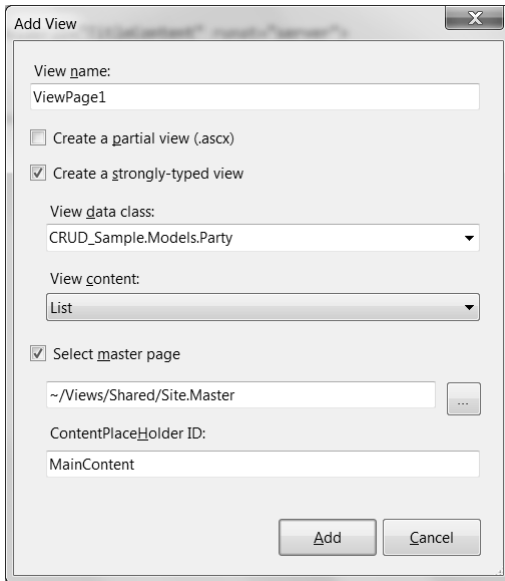


Abbildung 10.2 Hinzufügen einer View in Visual Studio 2010

## Partielle Views

Partielle Views sind Views, welche in anderen Views eingebunden werden können. Zu Zwecken der Parametrisierung wird dabei die Eigenschaft `ViewData` der aktuellen View, welche auch einen Verweis auf das Model enthält, an dieselbe Eigenschaft der partiellen View zugewiesen. Das im Abschnitt »Unterstützung beim Anlegen von Views durch Visual Studio« beschriebene Dialogfeld bietet die Möglichkeit, eine solche partielle View als Benutzersteuerelement (Usercontrol, *ascx*-Datei) anzulegen. Zum Einbinden partieller Views kann innerhalb der rufenden View die Methode `Html.RenderPartial` verwendet werden. Listing 10.18 demonstriert dies, indem eine partielle View mit dem Namen `MyPartialView` eingebunden wird.

```
[...]
<%Html.RenderPartial("MyPartialView");%>
[...]
```

Listing 10.18 Einbinden einer partiellen View

## Html-Helper

Für die Implementierung von Views stehen Hilfsmethoden für wiederkehrende Aufgaben zur Verfügung. Listing 10.19 demonstriert die Verwendung einiger dieser so genannten *Html-Helper*. Die Methode `ActionLink` wird zum Rendern eines Links, welcher auf eine Action-Methode in einem Controller verweist, verwendet. Wird der Controller, wie im betrachteten Beispiel, nicht angeführt, so findet der aktuelle Controller Verwendung. Da sich die URL, welche auf eine Action-Methode verweist, durch eine Modifikation der Routing-Einträge ändern kann, ist die Verwendung dieser Methode dem direkten Hinterlegen von Hyperlinks vorzuziehen.

Die Methode `BeginForm` wird zum Rendern eines Html-Formulars verwendet. Dazu wird diese Methode innerhalb eines `using`-Blocks verwendet, wobei das Formular am Ende des Blocks geschlossen wird. Alternativ zum Einsatz eines `using`-Blocks kann ein Formular auch manuell mit der Methode `EndForm` geschlossen werden. An `BeginForm` können auch der Controller und/oder die Action-Methode, an welche(n) die Formulardaten gesendet werden sollen, angeführt werden. Wird kein Controller angeführt, wird der aktuelle verwendet. Dasselbe gilt für die Action-Methode.

Die Methode `LabelFor` erzeugt ein Beschriftungsfeld und die Methoden `HiddenFor`, `TextBoxFor`, `CheckBoxFor` und `DropDownListFor` erzeugen die von den Methodennamen genannten Steuerelemente für die als Lambda-Ausdruck angegebene Eigenschaft des Models. Die Vorschlagswerte für die `DropDownList` werden an den zweiten Parameter von `DropDownListFor` als `IEnumerable<SelectListItem>` übergeben. Neben diesen im betrachteten Beispiel verwendeten Methoden existiert auch eine Methode `RadioButtonFor`. Damit die Eingaben vor dem letzten Absenden des aktuellen Formulars nicht verloren gehen, geben diese Methoden eventuellen an die Seite übergebenen Parametern, welche für die einzelnen Eigenschaften des Models stehen, den Vorzug gegenüber den tatsächlichen Werten der angegebenen Eigenschaften. Mit einigen Überladungen dieser Methoden kann auch ein anderer Wert, welcher angezeigt werden soll, angegeben werden. Für Fälle, in denen kein Model existiert oder einfach nur Steuerelemente für Werte, die als Parameter oder in Form des Dictionary `ViewData` an die View übergeben wurden, existieren auch Gegenstücke zu diesen Methoden, welche die Angabe des Namens und somit die Angabe des anzuzeigenden Parameters bzw. `ViewData`-Eintrags als String entgegennehmen. Diese lauten auf `Label`, `Hidden`, `TextBox`, `CheckBox`, `DropDownList` und `RadioButton`.

```
<div>
    <%=Html.ActionLink("Back to List", "Index") %>
</div>
<% using (Html.BeginForm()) {%>

    <fieldset>
        <legend>Fields</legend>
        <%= Html.HiddenFor(model => model.PartyId) %>

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

        <div class="editor-label">
            <%= Html.LabelFor(model => model.Abandgarderobe) %>
        </div>
        <div class="editor-field">
            <%= Html.CheckBoxFor(model => model.Abandgarderobe) %>
        </div>

        <div class="editor-label">
            <%= Html.LabelFor(model => model.VeranstalterId) %>
        </div>
        <div class="editor-field">
            <%= Html.DropDownListFor(model => model.VeranstalterId, ViewData["Veranstalter"]
                as IEnumerable<SelectListItem>) %>
        </div>
```

```

        <div>
            <input type="submit" value="Save" />
        </div>
    </fieldset>

<% } %>

```

**Listing 10.19** Ausgewählte *Html-Helper*

## Validieren von Benutzereingaben

Für das Validieren von Benutzereingaben sind zwei Ansätze vorgesehen. Entweder findet die Validierung im Controller statt oder sie erfolgt deklarativ, indem das Model mit Validierungsattributen annotiert wird. Dieser Abschnitt geht auf diese beiden Möglichkeiten ein.

### Manuelles Validieren

Wurde durch benutzerdefinierte Validierungslogiken ein Eingabefehler entdeckt, kann dieser an die Methode `ViewData.ModelState.AddModelError` zur Zwischenspeicherung übergeben werden (Listing 10.20). Als erster Parameter wird der Name des Felds, auf das sich der Fehler bezieht, angegeben; als zweiter Parameter die Fehlermeldung.

```

public ActionResult Edit(Party p)
{
    if (p.Eintritt < 3)
    {
        ViewData.ModelState.AddModelError("Eintritt", "Eintritt darf nicht < 3 sein!");
    }
    [...]
    return View();
}

```

**Listing 10.20** Manuelles Validieren von Benutzereingaben

Zum Anzeigen der hinterlegten Validierungsfehler können die Methoden `ValidationSummary` und `ValidationMessageFor` verwendet werden. `ValidationSummary` gibt sämtliche Fehlermeldungen aus; `ValidationMessageFor` gibt die Fehlermeldung für die über einen Lambda-Ausdruck angegebene Eigenschaft. Für Fälle, in denen die Fehlermeldung nicht über einen Lambda-Ausdruck, sondern lediglich über einen String referenziert werden soll, kann die Methode `ValidationMessage` herangezogen werden. Listing 10.21 demonstriert die Verwendung von `ValidationSummary` und `ValidationMessageFor`. Damit eine eventuelle Fehlermeldung nicht doppelt angezeigt wird, wird mit dem zweiten Parameter von `ValidationMessageFor` angegeben, dass das Vorhandensein einer Fehlermeldung lediglich durch Ausgabe eines Sternchens angezeigt werden soll.

```

<%=Html.ValidationSummary("Die folgenden Fehler sind aufgetreten:") %>
[...]
<div class="editor-field">
    <%= Html.TextBoxFor(model => model.Eintritt) %>
    <%= Html.ValidationMessageFor(model => model.Eintritt, "*") %>
</div>

```

**Listing 10.21** Ausgabe eines Validierungsfehlers



## Deklaratives Validieren

Während in Version 1 von ASP.NET MVC das Validieren von Benutzereingaben manuell implementiert werden musste, bietet Version 2 die Möglichkeit, analog zu ASP.NET Dynamic Data Validierungsregeln für Modelle mittels Attributen festzulegen. Das Framework kümmert sich dabei selbstständig um die Validierung nach den festgelegten Regeln.

### Validierungsattribute

Für die Eigenschaften der in Listing 10.22 dargestellten Model-Klasse wurden mittels Attributen Validierungsregeln definiert. In Fällen, in denen das Modell nicht editiert werden kann, weil es beispielsweise generiert wurde, besteht auch die Möglichkeit, die Validierungsregeln in einer so genannten *Buddy*-Klasse anzugeben. Dabei handelt es sich um eine Klasse, innerhalb welcher die Eigenschaften der Modell-Klasse nachgebildet und mit entsprechenden Attributen versehen werden. Anschließend wird eine zusätzliche partielle Deklaration der Modell-Klasse mit dem Attribut `MetadataType` versehen, welches auf die Buddy-Klasse verweist (siehe Listing 10.23).

```
namespace ValidatorSample.Models
{
    public partial class Superheld
    {
        [Required(ErrorMessage="Name wird benötigt")]
        [StringLength(30, ErrorMessage="Max. 30 Zeichen erlaubt")]
        public string Name { get; set; }

        [Range(1900,2100, ErrorMessage="Muss zwischen 1900 und 2100 sein")]
        public int Geburtsjahr { get; set; }

        [RegularExpression(@"\.+ \#\d+", ErrorMessage = "Falsches Format, XXXX #00 erwartet.
        Beispiel Detective Comics #32")]
        [DisplayName("Titel des ersten Comics")]
        public string ErsterComic { get; set; }

        [Required(ErrorMessage = "Kraftherkunft wird benötigt")]
        public string Kraftherkunft { get; set; }

        public bool KannFliegen { get; set; }
    }
}
```

**Listing 10.22** Verwendung von Validierungsattributen

```
namespace ValidatorSample.Models
{
    [MetadataType(typeof(SuperheldMetaData))]
    public partial class Superheld
    {
    }

    public class SuperheldMetaData
```

```

{
    [Required(ErrorMessage = "'Erster Comic' wird benötigt")]
    public string ErsterComic { get; set; }
}

```

**Listing 10.23** Implementierung einer *Buddy*-Klasse

## Serverseitige deklarative Validierung

Listing 10.24 beinhaltet zwei Action-Methoden eines Controllers, deren Aufgabe das Speichern von Instanzen des Modells aus Listing 10.22 ist. Die erste Methode wird beim ersten Anfordern von `/Create` via *GET* aufgerufen. Im Zuge dessen soll nicht validiert, sondern lediglich ein leeres Formular angezeigt werden. Wird dieses Formular via *POST* abgesendet, wird die zweite Methode aufgerufen. Um festzulegen, dass diese Methode für *POST-Anfragen* gedacht ist, wurde sie mit dem Attribut `HttpPost` versehen. In dieser Methode wird zunächst geprüft, ob die übergebene Model-Instanz, welche die erfassten Daten beinhaltet, den definierten Validierungsregeln genügt. Ist dem nicht so, wird nochmals auf das Eingabeformular verwiesen, wo die entdeckten Validierungsfehler wie im Abschnitt »Manuelles Validieren« beschrieben ausgegeben werden.

```

public ActionResult Create()
{
    Superheld s = new Superheld();
    return View(s);
}

[HttpPost]
public ActionResult Create(Superheld s)
{
    if (!ModelState.IsValid) return View(s);

    Debug.WriteLine("Täusche Speichern vor: " + s.Name);

    return View("Success")
}

```

**Listing 10.24** Reagieren auf Validierungsergebnis

## Clientseitige Validierung

Bis dato wurde die auf diese Art festgelegte Validierung lediglich serverseitig durchgeführt. Es besteht nun die Möglichkeit einer zusätzlichen clientseitigen Validierung, um unnötige Postbacks zu vermeiden. Um in den Genuss dieser Möglichkeit zu kommen, sind die entsprechenden JavaScript-Bibliotheken, welche im Lieferumfang von ASP.NET enthalten sind, einzubinden. Zusätzlich ist dann noch die Methode `Html.EnableClientValidation` aufzurufen (Listing 10.25).

```

<script
    src="/Scripts/MicrosoftAjax.debug.js"
    type="text/javascript" />

<script
    src="/Scripts/MicrosoftMvcValidation.debug.js"

```

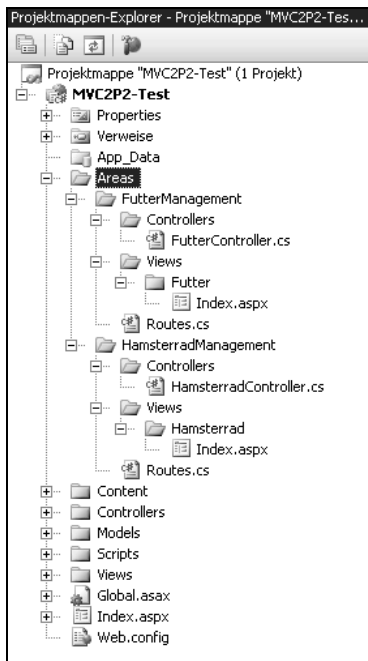
```
type="text/javascript" />  
<% Html.EnableClientValidation(); %>
```

**Listing 10.25** Aktivierung der clientseitigen Validierung

## Areas

Eine Neuerung in ASP.NET MVC 2 ist die Möglichkeit zur Aufteilung eines Projekts auf *Areas*. Eine Area ist ein Teilprojekt, welches möglichst eigenständig entwickelt werden kann. Abbildung 10.3 zeigt eine mögliche Ordnerstruktur für solche Projekte. Die einzelnen Areas befinden sich hierbei unterhalb des Ordners *Areas*, wobei jede Area ähnlich wie ein ASP.NET MVC-Projekt organisiert ist. So finden sich in beiden hier gezeigten Areas beispielsweise die Ordner *Controllers* und *Views* wieder. Zusätzlich existiert eine Standardarea, deren Controller und Views sich, wie bei Area-losen Projekten auch, in den gleichnamigen Ordnern unterhalb des Projektordners befinden.

**HINWEIS** Visual Studio 2010 unterstützt Sie beim Hinzufügen einer neuen Area durch Bereitstellen des Befehls *Add | Area* aus dem Kontextmenü des Projekts im Solution Explorer.



**Abbildung 10.3** Ordnerstruktur eines Projekts mit *Areas*

Jede Area ist angehalten, Informationen über ihre Routen preiszugeben. Dies erfolgt über eine Subklasse von *AreaRegistration*. Listing 10.26 demonstriert dies. Die zu überschreibende Eigenschaft *AreaName* liefert den Namen der Area; die zu überschreibende Methode *RegisterArea* trägt die Routen der jeweiligen Area mittels *MapRoute* in den übergebenen Context ein. Dabei ist zu beachten, dass mit dem letzten Parameter die Namensräume der Controller der jeweiligen Area bekanntgegeben werden.

```

public class Routes : AreaRegistration
{
    public override string AreaName
    {
        get { return "FutterManagement"; }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        context.MapRoute(
            "FutterManagement_default",
            "futter/{controller}/{action}/{id}",
            new {
                controller = "Futter",
                action = "Index",
                id = "" },
            null,
            new string[] { "MVC2P2_Test.Areas.FutterManagement" }
        );
    }
}

```

**Listing 10.26** Bereitstellen von Informationen für eine *Area*

Damit diese Angaben auch Beachtung finden, ist in der Methode `Application_Start` in der `Global.asax` die Methode `AreaRegistration.RegisterAllAreas` aufzurufen (Listing 10.27). Diese durchsucht das Projekt nach Subklassen von `AreaRegistration`, instanziiert diese und erweitert die Routendefinition mit deren Implementierung von `RegisterArea`.

```

protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RegisterRoutes(RouteTable.Routes);
}

```

**Listing 10.27** Registrierung aller *Areas* via *RegisterAllAreas*

Bei Area-übergreifenden Verweisen ist nun auch der Name der Ziel-Area anzugeben. Dabei handelt es sich um jenen String, welcher von der Eigenschaft `AreaName` der jeweiligen Implementierung von `AreaRegistration` zurückgeliefert wird (Listing 10.26). Handelt es sich bei der Ziel-Area um die Standard-Area, so kann dies durch einen Leerstring angegeben werden. Listing 10.28 demonstriert dies. Der erste Link verweist auf die Standard-Area; der zweite auf die Area mit dem Namen `FutterManagement`. Dabei sind jeweils über den letzten Parameter die gewünschten `Html`-Attribute als Instanz von `HtmlAttributes` zu übergeben, damit sich der Compiler für die richtige Überladung entscheidet. Da diese Angabe im betrachteten Beispiel nicht benötigt wird, wird lediglich `null` übergeben.

```

<%= Html.ActionLink(
    "Hamster", "Index", "Hamster",
    new { area = "" }, null)%>

<%= Html.ActionLink(
    "Futter", "Index", "Futter",
    new { area = "FutterManagement" }, null)%>

```

**Listing 10.28** Area-übergreifende Links

# Filter

Kommt es zur Abarbeitung einer ASP.NET MVC-Anfrage werden in der Regel zunächst eine Action-Methode sowie dann eine View, welche das Ergebnis rendert, ausgeführt. *Filter* geben die Möglichkeit, vor und zwischen diesen Schritten benutzerdefinierte Logiken zur Ausführung zu bringen, wobei ein Filter auf beliebig viele Seiten angewandt werden kann. Beispielsweise könnte ein Filter definiert werden, welcher für alle Seiten eines bestimmten Verzeichnisses vor der Ausführung der Action-Methode prüft, ob der aufrufende Benutzer angemeldet ist und falls dem nicht so ist, an ein Login-Formular weiterdelegiert, ohne die Anfrage weiter zu bearbeiten.

ASP.NET MVC bietet vier Arten von Filtern an: Authorization-Filter werden vor Abarbeitung der Anfrage ausgeführt; Action-Filter vor und nach dem Ausführen der Action-Methode; Result-Filter vor und nach dem Ausführen des Action-Results (zum Beispiel vor und nach dem Ausführen der View) und Exception-Filter, nachdem eine Ausnahme ausgelöst wurde. Implementiert wird ein Filter, indem eine Klasse bereitgestellt wird, welche zum einen von `FilterAttribute` erbt und zum anderen mindestens eine Schnittstelle, welche mit einer der vier Filter-Arten assoziiert wird. Durch das Erben von `FilterAttribute` kann der Filter als Attribut verwendet werden, und somit die Action-Methoden, auf welche der Filter angewandt werden soll, damit annotiert werden. Informationen über die Schnittstellen sowie deren Methoden finden sich in Tabelle 10.2.

Schnittstelle	Methode	Beschreibung
<code>IAuthorizationFilter</code>	<code>OnAuthorization</code>	Wird ausgeführt, bevor die Anfrage abgearbeitet wird
<code>IActionFilter</code>	<code>OnActionExecuting</code>	Wird vor der Action-Methode ausgeführt
	<code>OnActionExecuted</code>	Wird nach der Action-Methode ausgeführt
<code>IResultFilter</code>	<code>OnResultExecuting</code>	Wird vor dem Action-Result (z.B. View) ausgeführt
	<code>OnResultExecuted</code>	Wird nach dem Action-Result (z.B. View) ausgeführt
<code>IExceptionHandler</code>	<code>OnException</code>	Wird ausgeführt, wenn eine Ausnahme ausgelöst wurde

**Tabelle 10.2** Für die Implementierung von Filtern bereitgestellte Schnittstellen

Jeder Filter-Methode wird von ASP.NET MVC eine Instanz einer Subklasse von `ControllerContext`, welche Informationen über den aktuellen Stand der Abarbeitung der jeweiligen Anfrage bietet, übergeben. Unter anderem beinhaltet diese Instanz auch eine Eigenschaft `Result` vom Typ `ActionResult`. Durch das Setzen dieser Eigenschaft kann die Filter-Methode das darzustellende Ergebnis beeinflussen. Daneben beinhaltet diese Instanz auch eine Eigenschaft `ExceptionHandled`. Wird diese auf `true` gesetzt, wird angezeigt, dass sich der Filter um eine aufgetretene Ausnahme gekümmert hat und somit die Abarbeitung der Anfrage fortgesetzt werden kann.

Listing 10.29 zeigt eine beispielhafte Filter-Implementierung, welche alle vier Filter-Schnittstellen implementiert und einige Informationen zum aktuellen Zustand der Abarbeitung ausgibt. Die Verwendung der zuvor diskutierten Eigenschaften `Result` und `ExceptionHandled` wird dabei durch Kommentare angedeutet. Damit einfacher ersichtlich ist, welche Methode von welchem der vier Schnittstellen vorgegeben wird, wurden diese explizit implementiert.

```

public class DemoFilter:
    FilterAttribute, IAuthorizationFilter,
    IActionFilter, IResultFilter, IExceptionFilter
{

    void IAuthorizationFilter.OnAuthorization(AuthorizationContext filterContext)
    {
        // filterContext.Result = ...

        Debug.WriteLine("IAuthorizationFilter.OnAuthorization");
        Debug.WriteLine("  User: " + filterContext.HttpContext.User.Identity.Name);
        Debug.WriteLine("  Controller: " + filterContext.Controller.ToString());
        Debug.WriteLine("  Action: " + filterContext.ActionDescriptor.ActionName);
        Debug.WriteLine("");
    }

    void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)
    {
        Debug.WriteLine("IActionFilter.OnActionExecuting");
        Debug.WriteLine("  User: " + filterContext.HttpContext.User.Identity.Name);
        Debug.WriteLine("  Controller: " + filterContext.Controller.ToString());
        Debug.WriteLine("  Action: " + filterContext.ActionDescriptor.ActionName);
        Debug.WriteLine("");
    }

    void IActionFilter.OnActionExecuted(ActionExecutedContext filterContext)
    {
        Debug.WriteLine("IActionFilter.OnActionExecuted");
        Debug.WriteLine("  User: " + filterContext.HttpContext.User.Identity.Name);
        Debug.WriteLine("  Controller: " + filterContext.Controller.ToString());
        Debug.WriteLine("  Action: " + filterContext.ActionDescriptor.ActionName);
        Debug.WriteLine("  Exception: " + filterContext.Exception);
        Debug.WriteLine("  ExceptionHandled: " + filterContext.ExceptionHandled);

        Debug.WriteLine("");
    }

    void IResultFilter.OnResultExecuting(ResultExecutingContext filterContext)
    {
        Debug.WriteLine("IResultFilter.OnResultExecuting");
        Debug.WriteLine("  User: " + filterContext.HttpContext.User.Identity.Name);
        Debug.WriteLine("  Controller: " + filterContext.Controller.ToString());
        Debug.WriteLine("  Result: " + filterContext.Result.ToString());
        Debug.WriteLine("");
    }

    void IResultFilter.OnResultExecuted(ResultExecutedContext filterContext)
    {
        Debug.WriteLine("IResultFilter.OnResultExecuted");
        Debug.WriteLine("  User: " + filterContext.HttpContext.User.Identity.Name);
        Debug.WriteLine("  Controller: " + filterContext.Controller.ToString());
        Debug.WriteLine("  Result: " + filterContext.Result.ToString());
        Debug.WriteLine("  Exception: " + filterContext.Exception);
        Debug.WriteLine("  ExceptionHandled: " + filterContext.ExceptionHandled);
        Debug.WriteLine("");
    }
}

```

```
void IExceptionHandler.OnException(ExceptionContext filterContext)
{
    Debug.WriteLine("IExceptionHandler.OnException");
    Debug.WriteLine("  User: " + filterContext.HttpContext.User.Identity.Name);
    Debug.WriteLine("  Controller: " + filterContext.Controller.ToString());
    Debug.WriteLine("  Result: " + filterContext.Result.ToString());
    Debug.WriteLine("  Exception: " + filterContext.Exception);
    Debug.WriteLine("  ExceptionHandled: " + filterContext.ExceptionHandled);
    Debug.WriteLine("");

    // filterContext.ExceptionHandled = true;
}
```

**Listing 10.29** Filter

Um einen Filter auf eine Action-Methode anzuwenden, ist diese mit der Filter-Implementierung zu annotieren. Listing 10.30 demonstriert dies. Alternativ dazu kann auch eine Controller-Klasse mit einem Action-Filter annotiert werden. Dies bewirkt, dass der Filter für jede einzelne Action-Methode der jeweiligen Controller-Klasse herangezogen wird.

```
[DemoFilter]
public ActionResult Edit(int? id)
{
    [...]
}
```

**Listing 10.30** Anwenden eines Filters

## Exkurs: MVC und Dependency Injection

Die Muster MVC und Dependency Injection haben das Ziel, die Flexibilität und Testbarkeit von Applikationen zu steigern. MVC sieht dazu die Möglichkeit einer sauberen Schichtentrennung vor. Dependency Injection erleichtert das Austauschen und Testen von Abhängigkeiten. Richtig eingesetzt, ergänzen sich diese beiden Muster gegenseitig und führen zu einer qualitativ hochwertigeren Software-Architektur. Dieser Abschnitt zeigt, wie solch eine Architektur mittels ASP.NET MVC und den Dependency Injection Features von Spring.NET geschaffen werden kann.

### Fallbeispiel ohne Dependency Injection

Um den Sinn hinter Dependency Injection zu erläutern, soll zunächst eine Implementierung gezeigt werden, welche ohne dieses Muster auskommt. Im Abschnitt »Fallbeispiel mit Dependency Injection« wird anschließend gezeigt, wie diese Implementierung durch die Verwendung von Dependency Injection wartbarer und vor allem testbarer gestaltet werden kann.

## Implementierung mit ASP.NET MVC

Um den Nutzen des Zusammenspiels der beiden Muster zu demonstrieren, soll zunächst eine simple Webapplikation, welche (vorerst) lediglich dem MVC-Muster folgt, inspiziert werden. Diese Applikation bietet die Möglichkeit einer Kundensuche. Dazu werden zunächst Suchkriterien erfasst (Abbildung 10.4). Ist diese Suche erfolglos, so wird nochmals die Suchmaske angezeigt – jedoch mit einem entsprechenden Hinweis. Falls die Suche mehrere Datensätze liefert, werden diese aufgelistet (Abbildung 10.5). Für jeden Datensatz kann anschließend eine Detailansicht (Abbildung 10.6) angefordert werden. Liefert die Suche genau einen einzigen Datensatz, so wird für diesen direkt die Detailansicht präsentiert – die Auflistung dieses Datensatzes analog zur Auflistung in Abbildung 10.5 wird in diesem Fall also übersprungen.

Listing 10.31 zeigt die für diese Suche zuständige Action-Methode des entsprechenden Controllers. Dabei ist zu beachten, dass der Rückgabewert vom Typ `ActionResult` ist. Instanzen dieser Klasse teilen dem Framework z.B. mit, welche View angezeigt werden soll und welches Model an diese View weitergereicht werden soll. An die Methode werden die erfassten Suchparameter (vgl. Abbildung 10.4) übergeben. Anschließend bedient sie sich eines `CustomerDAOs`, um die Suchabfrage anzustoßen. Das Ergebnis ist eine `List<Customer>`. Die Anzahl der gefundenen Datensätze bestimmt die weitere Vorgehensweise. Eine leere Ergebnismenge führt zur Verwendung des Views `Index` sowie eines `IndexModel`s; eine Ergebnismenge mit einem einzigen Eintrag zur Verwendung von `Detail` und `DetailModel` und bei einer Ergebnismenge mit mehreren Einträgen werden `SearchResult` und `SearchResultModel` herangezogen.

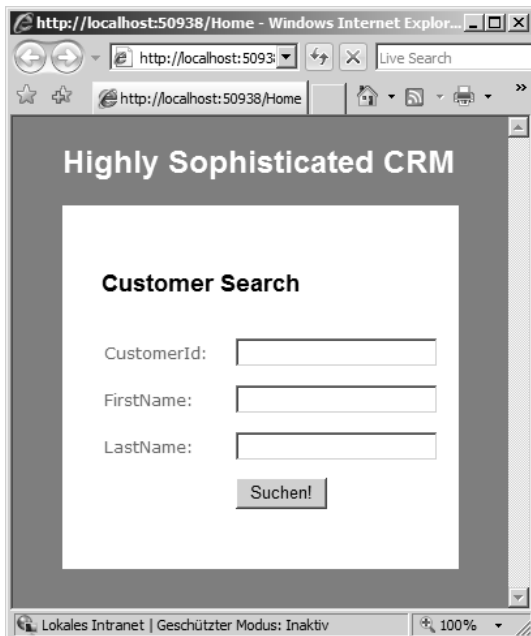


Abbildung 10.4 Suchmaske der Beispielapplikation



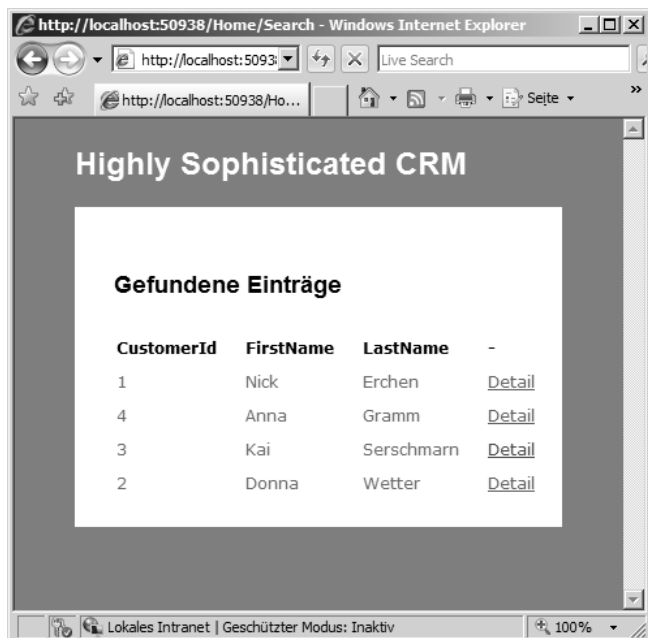


Abbildung 10.5 Auflistung der gefundenen Kunden

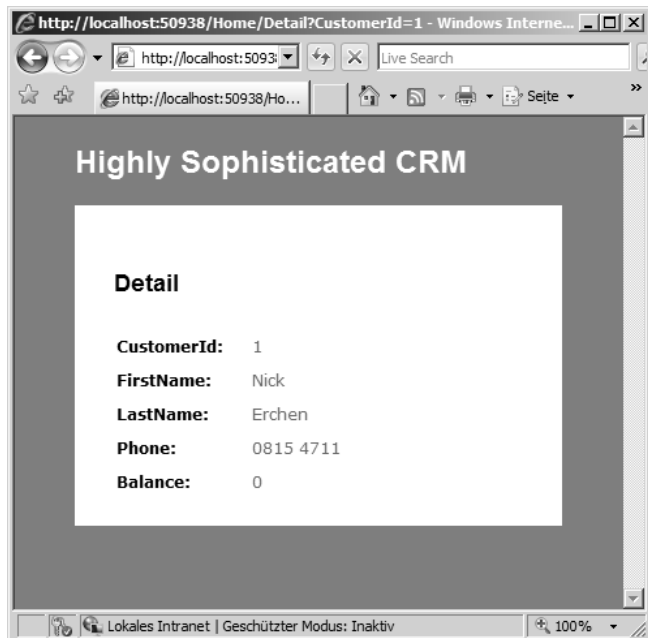


Abbildung 10.6 Detailansicht eines Kunden

```

public ActionResult Search(String customerId, String firstName, String lastName)
{
    int? customerIdsInt;
    CustomerDAO dao = new CustomerDAO();
    List<Customer> customers;

    customerIdsInt = ParseInt(customerId);

    customers = dao.FindCustomersBy(customerIdsInt, firstName, lastName);

    if (customers.Count == 0)
    {
        IndexModel model;
        model = new IndexModel();
        model.message = "Es wurden keine Einträge gefunden!";
        return View("Index", model);
    }
    else if (customers.Count == 1)
    {
        Customer foundCustomer;
        foundCustomer = customers[0];

        DetailModel model;
        model = new DetailModel();
        model.customer = foundCustomer;
        model.accountBalance = dao.CalcAccountBalanceFor(foundCustomer.CustomerId);

        return View("Detail", model);
    }
    else
    {
        SearchResultModel model;
        model = new SearchResultModel();
        model.customers = customers;
        return View("SearchResult", model);
    }
}

```

**Listing 10.31** Action-Methode für Kundensuche

## Unit-Test

Die Tatsache, dass die Action-Methode alle vom Framework benötigten Daten, wie z.B. Name der View oder das Model, zurückliefert, erleichtert das Testen dieser ungemein. Der Controller kann prinzipiell sogar ohne MVC-Framework getestet werden. Dies wird durch den Testfall in Listing 10.32 demonstriert. Dieser Testfall prüft, ob bei einer Suche mit einem einzigen Ergebnis direkt zur Detailansicht umgeleitet wird, sowie ob im Zuge dessen das erwartete Model weitergereicht wird. Dazu werden Variablen mit den Eingangsdaten und den erwünschten Ergebnissen definiert. Die Variable `expectedViewName` beinhaltet z.B. den Namen des erwarteten Views; `expectedModelType` den Typ des erwarteten Models. Nach dem Aufrufen der zu testenden Methode wird das Ergebnis vom Typ `ActionResult` nach `ViewResult` gecastet. Anschließend wird geprüft, ob dieser `ViewResult` sowie das sich im `ViewResult` befindliche Model die erwarteten Werte aufweist.

```
[TestMethod()]
public void SearchTest()
{
    ViewResult result;

    string customerId = "";
    string firstName = "";
    string lastName = "Erchen";

    string expectedViewName = "Detail";
    Type expectedModelType = typeof(DetailModel);
    decimal expectedAccountBalance = 0;
    string expectedLastName = "Erchen";

    HomeController target = new HomeController();

    result = (ViewResult)target.Search(customerId, firstName, lastName);

    Assert.AreEqual(expectedViewName, result.ViewName);
    Assert.IsInstanceOfType(result.ViewData.Model, expectedModelType);

    DetailModel model = (DetailModel)result.ViewData.Model;

    Assert.AreEqual(expectedAccountBalance, model.accountBalance);
    Assert.AreEqual(expectedLastName, model.customer.LastName);
}
```

**Listing 10.32** Testmethode

## Diskussion der betrachteten Lösung

Die vorgestellte Lösung bringt jene Vorteile mit sich, welche von ASP.NET MVC versprochen werden: Zum einen herrscht eine klare Trennung zwischen Präsentation und Logik vor. Dies erleichtert die Wiederverwendung und ermöglicht (prinzipiell) die Weiterentwicklung dieser beiden Schichten durch verschiedene Personen. Zum anderen kann die Logik des Controllers, wie mittels Listing 10.32 demonstriert wurde, relativ einfach getestet sowie testautomatisiert werden.

Ein Nachteil der diskutierten Lösung ist die Tatsache, dass der Controller direkt vom verwendeten Data Access Object (DAO) abhängig ist. Das DAO kann somit zum einen nicht flexibel ausgetauscht werden – zumindest nicht ohne Änderung von bestehenden Codestrecken. Zum anderen – und dieser Punkt ist aus Sicht des Autors viel gravierender – kann der Controller nicht für sich allein getestet werden. Ein Test des Controllers geht immer mit einem indirekten Test des DAOs einher. Ferner muss beachtet werden, dass der Ausgang der Tests von den Testdaten in der Datenbank abhängig ist. Der Test in Listing 10.32 geht beispielsweise davon aus, dass es nur einen einzigen Kunden mit dem Nachnamen *Erchen* gibt. Dies hat zur Folge, dass die Testdatenbank sorgfältig gewartet und zu Beginn eines Tests in einen wohldefinierten Ausgangszustand, z.B. durch Einspielen von SQL-Skripts oder eines Backups, versetzt werden muss.

Das Muster Dependency Injection schlägt genau in diese Kerben. Die Abhängigkeiten, wie z.B. das DAO im betrachteten Fall, werden über den Konstruktor oder über Property zugewiesen – man spricht in diesem Zusammenhang von *injisieren*. Diese Abhängigkeiten müssen jedoch nicht manuell zugewiesen werden. Die Zuweisung erfolgt automatisiert durch ein Framework. Dieses ermittelt über die Konfiguration, welche Abhängigkeiten zu welcher Komponente zugewiesen werden sollen. Allein durch Modifikation der Konfiguration

können somit Abhängigkeiten ausgetauscht werden. Dies ermöglicht z.B. den Austausch eines *SqlServerDAO* gegen ein *OracleDAO*, sofern diese beiden Typen mit jenem der von der Komponente benötigten Abhängigkeit kompatibel sind. In der Regel wird dies erreicht, indem die Komponente gegen eine Schnittstelle programmiert wird und die Abhängigkeiten dieser Schnittstelle implementieren. Im Zuge von Unit-Tests könnte somit auch eine Dummy-Implementierung des benötigten DAOs, welche lediglich die für diesen Test benötigten Daten zurückliefert, zugewiesen werden.

## Fallbeispiel mit Dependency Injection

Nachdem die in diesem Kapitel verwendete Beispielimplementierung nun besprochen wurde, wird in diesem Abschnitt gezeigt, wie durch den zusätzlichen Einsatz von Dependency Injection die Wartbarkeit sowie Testbarkeit gesteigert werden kann.

### Implementierung der Webapplikation

In weiterer Folge wird die vorgestellte Implementierung erweitert und dabei vom Muster Dependency Injection Gebrauch gemacht. Dazu wird an dieser Stelle das freie Framework Spring.NET [SPR] herangezogen. Es handelt sich dabei um die Portierung eines sehr populären und richtungsweisenden JAVA-Framework, welches u.a. Dependency Injection unterstützt.

Dependency Injection sieht, wie oben diskutiert, vor, dass Abhängigkeiten durch ein Framework (hier: Spring.NET) injiziert werden. Dazu wurde dem DAO eine Schnittstelle spendiert (Listing 10.33) und der Controller um einen Konstruktor erweitert, welcher eine Implementierung dieser Schnittstelle entgegennimmt. Wie aus Listing 10.34 ersichtlich, verwendet nun die Action-Methode das auf diesem Wege zugewiesene DAO.

Da Spring.NET die gewünschte Implementierung dieses DAOs automatisch zuweisen soll, müssen auch die dafür nötigen Informationen über ein Konfigurationsfile bereitgestellt werden (Listing 10.35). Über dieses Konfigurationsfile werden die benötigten Klassen registriert. Das Attribut *id* definiert dabei den Namen, über welchen künftig Objekte der jeweiligen Klasse bezogen werden können; das Attribut *type* liefert Auskunft über den voll qualifizierten Namen der Klasse sowie über die Assembly, in welcher dieser Typ zu finden ist. Mittels *singleton="false"* wird definiert, dass Spring.NET immer eine neue Instanz erzeugen soll, wenn eine Instanz der jeweiligen Klasse angefordert wird. Ansonsten würde immer ein und dieselbe Instanz zur Verwendung kommen. Über den Tag *constructor-arg* wird definiert, dass Spring.NET an den Konstruktor von *HomeController* eine Instanz von *CustomerDAO* übergeben soll – oder genauer gesagt: Es wird an den Konstruktor eine Instanz jener Klasse übergeben, welche mit der ID *CustomerDAO* registriert wurde.

Wird nun bei Spring.NET eine Instanz des Controllers angefordert, so erzeugt dieses eine Instanz von *CustomerDAO*, sowie eine Instanz vom Controller und weist dieser das DAO über den Konstruktor zu. Anschließend wird der Controller an den Aufrufer zurückgegeben – der Aufrufer bekommt somit von der *Verdrahtung* zwischen der angeforderten Komponente und seinen Abhängigkeiten nicht das Geringste mit.

```
public interface ICustomerDAO
{
    List<Customer> FindCustomersBy(
        int? customerId, string firstName, string lastName);

    [...]
}
```

**Listing 10.33** Schnittstelle für *CustomerDAO*

```
public class HomeController : Controller
{
    private ICustomerDAO dao;

    public HomeController(ICustomerDAO dao)
    {
        this.dao = dao;
    }

    public ActionResult Search(String customerId, String firstName, String lastName)
    {
        int? customerIdsInt;

        List<Customer> customers;

        customerIdsInt = ParseInt(customerId);

        customers = dao.FindCustomersBy(customerIdsInt, firstName, lastName);

        if (customers.Count == 0)
        {
            IndexModel model;438

            model = new IndexModel();
            model.message = "Es wurden keine Einträge gefunden!";
            return View("Index", model);
        }
        else if (customers.Count == 1)
        {
            Customer foundCustomer;
            foundCustomer = customers[0];

            DetailModel model;
            model = new DetailModel();
            model.customer = foundCustomer;
[...]
```

return View("Detail", model);

```
        }
        else
        {
            SearchResultModel model;
            model = new SearchResultModel();
            model.customers = customers;
            return View("SearchResult", model);
        }
    }
[...]
```

**Listing 10.34** *HomeController*, welcher DAO als Konstruktorargument erwartet

```

<objects xmlns="http://www.springframework.net" >

  <object
    id="CustomerDAO"
    type="MVC_DI.Data.CustomerDAO, MVC_DI_02"
    singleton="false">
  </object>

  <object
    id="HomeController"
    type="MVC_DI.Controllers.HomeController, MVC_DI_02"
    singleton="false">

    <constructor-arg ref="CustomerDAO" />

  </object>
</objects>

```

**Listing 10.35** *Spring.NET-Konfiguration*

## Brückenschlag zwischen ASP.NET MVC und Spring.NET

Da der Controller um einen Konstruktor mit einem Parameter erweitert wurde, kann er nicht mehr von MVC-Framework verwendet werden. Dieses schreibt nämlich vor, dass Controller einen parameterlosen Konstruktor aufweisen müssen. Deswegen wird in weiterer Folge MVC-Framework angewiesen, die benötigten Controller über Spring.NET zu beziehen. Dazu muss man wissen, dass ASP.NET MVC die benötigten Controller über eine Factory anfordert. Diese Factory implementiert die Schnittstelle `IControllerFactory` und kann durch eine benutzerdefinierte Factory, welche ebenfalls diese Schnittstelle implementiert, ersetzt werden. Wie aus Listing 10.36 ersichtlich, definiert diese Schnittstelle (mittlerweile) zwei Methoden: `CreateController` wird vom Framework zur Erzeugung eines Controllers aufgerufen; `DisposeController` wird vom Framework aufgerufen, wenn ein zuvor erzeugter Controller nicht mehr benötigt wird.

Listing 10.37 zeigt eine Implementierung dieser Schnittstelle, welche die benötigten Controller bei Spring.NET anfordert. An den Konstruktor wird der vollständige Name des Spring.NET-Konfigurationsfiles (Listing 10.35) übergeben. Mit dieser Information wird eine Instanz von `XmlApplicationContext` erzeugt. Über diesen Kontext können Komponenten, wie z.B. registrierte Controller, angefordert werden.

`CreateController` nimmt einen `RequestContext` sowie den Namen des gewünschten Controllers entgegen. Aus diesem (internen) Namen wird der Name der Klasse abgeleitet, indem die Endung *Controller* angehängt wird. Anschließend wird beim Kontext eine Instanz dieser Klasse angefordert, woraufhin Spring.NET eine Komponente, zu welcher in der Konfiguration dieser Name über das Attribut `id` (vgl. Listing 10.35) zugewiesen wurde, erzeugt. Diese Instanz stellt auch den Rückgabewert der Methode dar. Konnte der gewünschte Controller über diesen Weg nicht erzeugt werden, so wird dieser Umstand protokolliert und `NULL` zurückgeliefert.

Die Methode `DisposeController` prüft lediglich, ob der nicht mehr benötigte Controller die Schnittstelle `IDisposable` implementiert. Falls dem so ist, wird für diesen die Methode `Dispose` aufgerufen.

Nun muss ASP.NET MVC noch darüber in Kenntnis gesetzt werden, dass diese benutzerdefinierte Factory zur Erzeugung von Controllern verwendet werden soll. Da dies idealerweise gleich beim Start der Webapplikation geschehen soll, bietet sich die Methode `Application_Start` (Listing 10.38) der `global.asax` dafür an: Zunächst wird eine Instanz der benutzerdefinierten Factory erzeugt und der Name der Spring.NET-Konfigurationsdatei an diese übergeben. Anschließend wird die Factory bei MVC-Framework registriert.

```
namespace System.Web.Mvc
{
    public interface IControllerFactory
    {
        IController CreateController(RequestContext context, string controllerName);
        void DisposeController(IController controller);
    }
}
```

**Listing 10.36** Schnittstelle für Controller-Factory aus *System.Web.Mvc*

```
public class SpringControllerFactory: IControllerFactory
{
    private IApplicationContext appContext;

    public SpringControllerFactory(String configFileName)
    {
        appContext = new XmlApplicationContext(configFileName);
    }

    public IController CreateController
        (System.Web.Routing.RequestContext context, string controllerName)
    {
        String fullControllerName = controllerName + "Controller";

        try
        {
            return (IController)appContext.GetObject(fullControllerName);
        }
        catch(Exception e)
        {
            Trace.TraceWarning("Could not create Controller: "
                + fullControllerName + ", " + e.Message + " " + e.StackTrace);
            return null;
        }
    }

    public void DisposeController(IController controller)
    {
        if (controller is IDisposable)
        {
            ((IDisposable)controller).Dispose();
        }
    }
}
```

**Listing 10.37** Implementierung einer benutzerdefinierten Controller-Factory

```
protected void Application_Start()
{
    [...]

    SpringControllerFactory factory;
    factory = new SpringControllerFactory("~/spring.xml");
    ControllerBuilder.Current.SetControllerFactory(factory);
}
```

**Listing 10.38** Registrierung der benutzerdefinierten Controller-Factory in der Datei *Global.asax*

## Testen

Durch die Integration von Spring.NET können Abhängigkeiten einfach(er) über die Konfiguration ausgetauscht werden. Es wird aber auch die Testbarkeit der Applikation erhöht. Für bestimmte Tests können nun eigene Spring.NET-Konfigurationsdateien herangezogen werden. Alternativ dazu kann jeder Test eine Dummy-Implementierung der Abhängigkeiten zuweisen. Im betrachteten Fall wäre dies z.B. eine Dummy-Implementierung der DAO-Schnittstelle, welche genau die für den Test benötigten Daten zurückliefert. Listing 10.39 demonstriert dies unter Verwendung von Mock-Framework NMock [NMO]. NMock bietet die Möglichkeit zur Erzeugung von Attrappen (Mock-Objekten), welche an der Stelle der eigentlichen Komponenten für Testzwecke verwendet werden können. Diese Möglichkeit wird genutzt, um für das DAO eine Attrappe zu erstellen.

Zunächst werden, wie bei Listing 10.32, die zu verwendenden Eingangsdaten sowie die erwarteten Ergebnisse definiert. Anschließend wird jene *List<Customer>* erzeugt, welche von der Attrappe des DAOs zurückgeliefert werden soll – eine *List<Customer>* mit einem bestimmten Kunden. Danach wird eine Attrappe für das DAO bei NMock angefordert sowie festgelegt, dass ein Aufruf der Methode *FindCustomersBy* mit bestimmten Parametern bei dieser Attrappe erwartet wird. Ferner wird festgelegt, dass diese Methode die zuvor erzeugte *List<Customer>* zurückliefern soll. Im Anschluss daran wird der zu testende Controller manuell erzeugt, die Attrappe über den Konstruktor zugewiesen und die Tests durchgeführt. Am Ende wird geprüft, ob alle für die Attrappe erwarteten Methodenaufrufe auch stattgefunden haben.

Auf diese Art und Weise könnte nun für jeden Test eine eigene Attrappe erzeugt werden. Einige Tests werden zum Beispiel eine Attrappe benötigen, welche mehrere Kunden zurückliefert; andere wiederum eine Attrappe, welche eine leere Ergebnismenge liefert. Diese Tests können nun unabhängig von der Datenzugriffsschicht ausgeführt werden. Ferner sind diese nicht mehr von bestimmten Konstellationen in der Test-Datenbank abhängig.

```
[TestMethod()]
public void SearchTest()
{
    // Input
    string customerId = "1";
    int customerIdAsInt = 1;
    string firstName = "";
    string lastName = "";

    // Erwartete Ergebnisse
    string expectedViewName = "Detail";
    Type expectedModelType = typeof(DetailModel);
    decimal expectedAccountBalance = 0;
    string expectedLastName = "Erchen";
```



```

// DAO-Ergebnis erzeugen
Customer fakedCustomer = new Customer();
fakedCustomer.CustomerId = 1;
fakedCustomer.FirstName = "Nick";
fakedCustomer.LastName = "Erchen";

List<Customer> fakedResult = new List<Customer>();
fakedResult.Add(fakedCustomer);

// DAO-Mock-Objekt erzeugen
Mockery mocks = new Mockery();
ICustomerDAO daoMock = mocks.NewMock<ICustomerDAO>();

Expect.Once.On(daoMock).Method("FindCustomersBy").With(customerIdAsInt, firstName,
lastName).Will(Return.Value(fakedResult));

[...]

// HomeController manuell erzeugen und Abhängigkeit (=Mock) zuweisen ...
HomeController target = new HomeController(daoMock);

// Tests durchführen
ViewResult result;
result = (ViewResult)target.Search(customerId, firstName, lastName);

Assert.AreEqual(expectedViewName, result.ViewName);
Assert.IsInstanceOfType(result.ViewData.Model, expectedModelType);

DetailModel model = (DetailModel)result.ViewData.Model;

Assert.AreEqual(expectedAccountBalance, model.accountBalance);
Assert.AreEqual(expectedLastName, model.customer.LastName);

mocks.VerifyAllExpectationsHaveBeenMet();
}

```

**Listing 10.39** Testen unter Injizieren eines *Mock*-Objekts

## Zusammenfassung und Fazit

Durch den Einsatz von ASP.NET MVC kann sowohl die Wartbarkeit als auch die Wiederverwendbarkeit und Testbarkeit einer Applikation erhöht werden sowie lästige Abhängigkeiten zu Test-Datenbanken vermieden werden.

Wie alles im Leben hat auch dieser Architekturansatz seinen Preis, denn »mehr Architektur« bedeutet erfahrungsgemäß auch »mehr Aufwand«. Bei großen Applikationen zahlt sich dieser Mehraufwand aus der Sicht des Autors jedoch allemal aus. Hier stellt die Investition in die Architektur zwar kurzfristig einen Aufwand dar; langfristig jedoch – vor dem Hintergrund der Aspekte Wartbarkeit, Wiederverwendbarkeit und Testbarkeit – einen Gewinn.

