Kochen mit Patrick

Auch in der vierzigsten Folge dieser Kolumne dreht sich noch einmal alles um ASP.NET, insbesondere um dessen Kompilierungsmodell und wie man es sich geschickt zunutze macht. Das monatliche Rezept kommt diesmal ohne Fleisch aus, lassen Sie sich überraschen!

dnpCode
A0906Kochen

Patrick A. Lorenz ist Geschäftsführer der PGK GmbH, einem auf .NET spezialisierten Technologiedienstleister. Daneben ist er als Autor tätig. Sein neuestes Buch „ASP.NET 3.5 mit AJAX" beschreibt die Neuerungen in .NET 3.5 für Webentwickler. In seiner Freizeit ist Patrick Hobbykoch. Sie erreichen ihn unter www.pgk.de oder lorenz@pgk.de.

Als ich die Dateien für die Kolumne vorbereitet habe, musste ich erst einmal mein Wissen über römische Zahlen auffrischen. Damit sind meine Word-Dokumente nämlich nummeriert, diesmal mit XL, der römischen Zahl für 40. So viele Kolumnen sind bereits erschienen. Unter [1] bin ich übrigens fündig geworden. Dort gibt es eine Online-Umrechnung – direkt neben der Beschreibung der schriftlichen Addition für Schüler der ersten bis vierten Klasse. Ein Mathematiker wäre wohl nicht aus mir geworden.

Das ASP.NET-Kompilierungsmodell

Was Sie schon immer über ASP.NET wissen wollten, aber nie zu fragen wagten – das könnte für das Modell der Projekt- und Seitenkompilierung von ASP.NET gelten. Unstrittig ist, dass es ein solches Modell geben muss. Schließlich ist gerade die Kompilierung ein gemeinhin bekannter Vorteil gegenüber anderen Frameworks wie dem klassischen ASP. Dieses interpretiert Seiten bei jedem Aufruf; das ist langsam und skaliert entsprechend schlecht. Ebenfalls bekannt ist, dass ASP.NET-Seiten je nach Art des Deployments beim jeweils ersten Aufruf etwas länger benötigen als bei allen Folgeaufrufen.

Das Modell besteht aus zwei Phasen. Die erste Kompilierung findet statt, wenn Sie oder Ihr Build-Server das Web-Application-Projekt explizit kompilieren. Die erzeugte Assembly landet im *bin*-Verzeichnis der Applikation.

Die viel interessantere Phase zwei findet implizit statt, spätestens wenn der erste Benutzer eine Seite aufruft. Dann nämlich wird die *.aspx*-Datei, mit der Deklaration von Web-Controls et cetera, selbst in eine Quelltextdatei umgewandelt und dynamisch im Hintergrund kompiliert. Die so erstellte Klasse leitet sich von der Code-behind-Klasse aus Ihrem Webprojekt ab.

Abbildung 1 zeigt das Vorgehen anhand von zwei nacheinander eintreffenden Seitenanforderungen. Beim ersten Aufruf (1) wird die physikalische Seite (2) durch die ASP.NET-Engine in eine Klasse kompiliert (3 und 4), die sich indirekt von Page ableitet. Diese wird instanziert, ausgeführt und das Ergebnis an den Client geliefert (5 und 6). Für den zweiten Aufruf (7) wird direkt eine neue Instanz der Klasse erzeugt (8). Ein neuerliches Parsen der physikalischen *.aspx*-Seite wird so vollständig umgangen.

Der Ansatz klingt abstrakt, lässt sich jedoch mit einfachen Mitteln sehr transparent nachvollziehen. Listing 1 zeigt eine sehr einfache Seite bestehend aus einigen statischen HTML-Fragmenten sowie drei Web-Controls. In die *TextBox* eingegebene Werte werden per Klick auf den *Button* in ein *Label* kopiert. Die Verarbeitung übernimmt die Ereignisbehandlung *BT_Click*, die in der Code-behind-Klasse der Seite implementiert wurde.

Stellen Sie sicher, dass Sie den Debug-Modus in der *web.config* oder der Seite aktiviert haben, und bringen Sie das Beispiel zur Ausführung. Orientieren Sie sich anschließend ein wenig im folgenden Ordner:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.
50727\Temporary ASP.NET Files
```

Unterhalb von *root* sollten Sie hier mindestens einen aktuellen Ordner erkennen, der in einer recht kryptischen Struktur weitere Ordner und schließlich nicht minder kryptische Dateien enthält. Die Namen ergeben sich aus Hash-Codes, informativer sind die Dateiendungen, darunter *.cs* und *.dll*. Klicken Sie sich durch die C#-Dateien, bis Sie eine finden, in deren ersten Zeilen sich der Pfad zur *default.aspx* findet, zum Beispiel so:

```
#pragma checksum
"C:\...\Compilation1\Default.aspx"
"{406ea660-64cf-4c82-b6f0-42d48172a799}"
"16EE0D19072629E660B592D71B436D42"
```

Die Datei enthält neben vielen scheinbar sehr speziellen Compiler-Anweisungen im Kern die für die *default.aspx* durch die ASP.NET-Engine generierte Quelltextklasse, die sich von Ihrer in Visual Studio angelegten Code-behind-Klasse ableitet:

```
public class default_aspx :
  global::Compilation1._Default,
  System.Web.SessionState.
  IRequiresSessionState,
  System.Web.IHttpHandler {
```

Die Klasse weist für jedes deklarativ notierte Server-Control eine eigene Build-Methode auf, zum Beispiel für den Button (gekürzt):

```
[System.Diagnostics.
   DebuggerNonUserCodeAttribute()]
private global::System.Web.UI.WebControls
  .Button @__BuildControlBT()
{
  global::System.Web.UI.WebControls
    .Button @__ctrl;
  @__ctrl = new global::System.Web.UI
   .WebControls.Button();
  this.BT = @__ctrl;
  @__ctrl.ApplyStyleSheetSkin(this);

  @__ctrl.ID = "BT";
  @__ctrl.Text = "OK";
  @__ctrl.Click -= new
    System.EventHandler(this.BT_Click);

  @__ctrl.Click += new
    System.EventHandler(this.BT_Click);
  return @__ctrl;
}
```

Interessant ist, dass auch die statischen HTML-Elemente und Texte in Form von Server-Controls integriert werden, und zwar immer alle Inhalte zusammen, die zwischen zwei „echten" Controls liegen:

```
@__parser.AddParsedSubObject(new
  System.Web.UI.LiteralControl(
  "\r\n\r\n<p>\r\nEingabe:\r\n"));

global::System.Web.UI.WebControls.
  TextBox @__ctrl1;
@__ctrl1 = this.@__BuildControlTB();
@__parser.AddParsedSubObject(@__ctrl1);
@__parser.AddParsedSubObject(new
  System.Web.UI.LiteralControl("\r\n"));
```

In der Summe der Methoden ergibt sich eine vollständige Abbildung der deklarativen ASP.NET-Seite, die folgerichtig während der Laufzeit nicht mehr direkt benötigt wird. Doch vorher muss die Klasse noch kompiliert werden. Dies übernimmt völlig unbemerkt der C#-Compiler in Form der Konsolenapplikation *csc.exe*, die übrigens auch von Visual Studio selbst verwendet wird. Der vollständige Aufruf des Compilers lässt sich in der *.out*-Datei im Verzeichnis erkennen. Die Details sind einen Abdruck an dieser Stelle nicht wert, aber grob sieht es so aus:

```
C:\WINDOWS\Microsoft.NET\Framework\v3.5\csc
.exe /t:library /utf8output
/R...<Referenzen>
/out:"c:\...\App_Web_op0th0tx.dll"
/D:DEBUG /debug+ /optimize- /w:4
/nowarn:1659;1699;1701 /warnaserror-
"c:\...\App_Web_op0th0tx.0.cs"
```

Etwaige Fehler werden in einer *.err*-Datei protokolliert, ansonsten steht die kompilierte Assembly im temporären Ordner zur weiteren Verwendung zur Verfügung. Wenn Sie diese mit einem Disassembler betrachten, werden Sie genau Ihre Seitenstruktur wiedererkennen können.

ASP.NET lädt im weiteren Verlauf die Assembly in den Speicher und instanziert bei jeder neuen Anforderung der Seite die generierte Klasse und führt diese mithilfe der ASP.NET-Infrastruktur aus. Dabei wird die ursprüngliche *.aspx*-Datei, wie beschrieben, nicht weiter verwendet – allerdings wird sie beobachtet. Etwaige Änderungen führen zu einer erneuten Kompilierung der Seite.

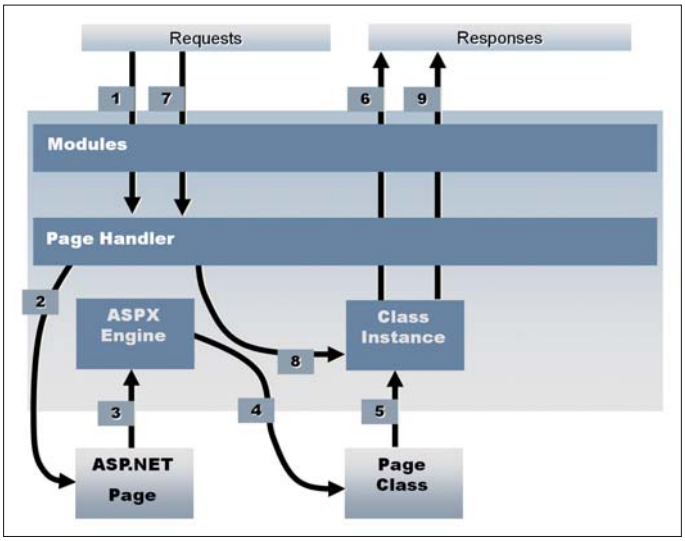
Damit nicht zu viele Assemblies im temporären Ordner herumschwirren, kann ASP.NET mehrere Seiten in einer Assembly bündeln. Und nach einem Neustart der Applikation beginnt das Spiel von Neuem und alle vorherigen Dateien werden gelöscht.

Listing 1

Eine deklarative ASP.NET-Seite.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="Compilation1._Default" %>

<html>
<head runat="server" />
<body>
 <form runat="server">
  <p>
   Eingabe:
   <asp:TextBox id="TB" runat="server" />
   <asp:Button id="BT" runat="server" Text="OK" OnClick="BT_Click" />
  </p>
  <p>
   Ausgabe:
   <asp:Label id="LB" runat="server" ForeColor="Red" Font-Bold="true" />
  </p>
 </form>
</body>
</html>
```



[Abb. 1] Zwei nacheinander eintreffende Seitenanforderungen werden bearbeitet.

ASP.NET Precompilation

Die Frage, was sich hinter der mit ASP.NET 2.0 eingeführten Precompilation auf sich hat und warum diese den Start einer Applikation beschleunigen kann, beantwortet sich nach dem Studium des vorherigen Ablaufs von ganz alleine: Die Generierung einer Klasse pro Seite und deren Kompilierung wird vom Produktivsystem zurück in die Deployment-Phase verlagert.

Eine Precompilation lässt sich zum Beispiel mit dem zu ASP.NET gehörenden Kommandozeilen-Programm *aspnet_compiler.exe* durchführen. Das Tool bietet drei Modi an:

- Bei der In-Place-Compilation wird der zuvor beschriebene Prozess durchgeführt und die generierten Assemblies werden im lokalen ASP.NET Temp-Ordner abgelegt.
- Wird dem Tool ein Zielverzeichnis übergeben (Parameter *targetDir*), wird das Ergebnis der Precompilation sowie die Applikation selbst in diesem Ordner abgelegt. Die *.as?x*-Dateien werden dabei als leere Platzhalter kopiert und sind entsprechend unveränderlich.
- Die dritte Variante gleicht der vorherigen, kopiert die *.as?x*-Dateien jedoch unverändert und erlaubt dadurch Änderungen am deklarativen Teil der Seiten (zusätzlicher Parameter *–u*).

Zwischenzeitlich hat Microsoft auch eine 2008er-Version des Projekttyps Web Deployment [2] sowie des zugehörigen IIS Web Deployment Tools veröffentlicht, welches das gesamte Deployment einer Webapplikation auf einen IIS einschließlich der Precompilation übernehmen kann.

Mit ASP.NET Version 2.0 hat sich im Innern des Frameworks sehr viel getan, und dieses wurde maßgeblich offener gestaltet. Dieser Umstand lässt sich auch ganz interessant an der Precompilation nachvollziehen. Das Kommandozeilen-Programm ist recht „dumm" und leitet die Parameter an die Klasse *ClientBuildManager* aus dem Namespace *System.Web.Compilation* weiter. Statt des Tools können Sie dieses API auch direkt einsetzen.

```
string sourcePath = @"C:\...\Compilation1";
string targetPath =
  @"C:\PrecompilationTest\";

var parameters = new
  ClientBuildManagerParameter
{
  PrecompilationFlags =
    PrecompilationFlags.Clean |
    PrecompilationFlags.OverwriteTarget
};

var bm = new ClientBuildManager("/",
    sourcePath, targetPath, parameters);
bm.PrecompileApplication();
```

Die Klasse *ClientBuildManager* bietet noch eine Reihe weiterer Methoden an, die zur Steuerung der Kompilierung außerhalb der eigentlichen Webapplikation genutzt werden können. Im Unterschied dazu steht die Klasse *BuildManager* aus demselben Namespace, die eine Reihe von sich teilweise überschneidenden statischen Methoden bietet. Diese können aus einer laufenden Webapplikation heraus aufgerufen werden, beispielsweise um eine Seite oder ein User Control zu instanzieren – was durchaus sehr hilfreich sein kann.

Verwendung von Expression Buildern

Expression Builder sind ein weitestgehend unbekanntes und ungenutztes Feature, das mit ASP.NET 2.0 eingeführt wurde. Mithilfe eines offenen Provider-Modells lassen sich innerhalb von ASP.NET-Seiten und ASP.NET-User-Controls Ausdrücke elegant an Control-Eigenschaften binden, die während der Kompilierung transparent in Quelltext umgewandelt werden. Hierzu existiert eine spezielle Syntax:

```
<%$ expressionPrefix:expressionValue %>
```

ASP.NET nutzt diese Möglichkeit beispielsweise zur deklarativen Integration von benannten *ConnectionStrings* in *DataSource*-Controls. Das folgende Beispiel stammt aus der MSDN-Hilfe:

```
<asp:SqlDataSource Runat="server"
 SelectCommand="SELECT * FROM
 [Employees]"
 ConnectionString=
   "<%$ ConnectionStrings:
    NorthwindConnectionString %>">
</asp:SqlDataSource>
```

Das gezeigte Präfix kennzeichnet die Art des Ausdrucks, der dahinter notiert wird. Der Name verweist dabei auf einen registrierten Expression Builder, von denen ASP.NET von Haus aus bereits drei kennt: *ConnectionString* für registrierte Datenbank-Connection-Strings, *AppSettings* für AppSetting-Einträge aus der Konfigurationsdatei *web.config* und *Resources* für Resource-Texte. Das ist ganz nett, der eigentliche Clou ist jedoch die Möglichkeit, eigene Expression Builder zu implementieren und so einen kurzen Weg aus dem deklarativen Seitenmodell in die eigene Infrastruktur zu schaffen. Das folgende Beispiel eines RandomColor-Expression-Builders ist etwas konstruiert, zeigt die Zusammenhänge aber ganz gut. Ziel ist ein Ausdruck, der zufällig aus einer Liste von übergebenen Farben eine auswählt und zurückliefert. Die Notation soll so aussehen:

```
<asp:Label runat="server"
 ForeColor="<%$ RandomColor:
            Red,Blue,Yellow %>"
 Text="Hallo Welt"
/>
```

Der neue Expression Builder wird von der gleichnamigen Basisklasse abgeleitet. Die abstrakte Klasse erzwingt die Implementierung der Methode *GetCodeExpression*, die eine *CodeExpression* aus dem Code DOM (*System.CodeDOM*) zurückzuliefern hat. Dabei handelt es sich um eine sprachenneutrale Darstellung dessen, was in der später von der ASP.NET-Engine generierten Quelltextdatei der Eigenschaft *ForeColor* zugewiesen wird. Die *CodeExpression* wird also später in Quelltext umgewandelt, der von ASP.NET kompiliert und ausgeführt wird. Alles klar? So schaut's aus:

```
public class
  RandomColorExpressionBuilder :
  ExpressionBuilder
{
 public override CodeExpression
  GetCodeExpression(BoundPropertyEntry
  entry, object parsedData,
  ExpressionBuilderContext context)
 {
  return new CodeMethodInvokeExpression(
   new CodeTypeReferenceExpression(
   this.GetType()), "GetRandomColor",
   new[]
   {
    new CodePrimitiveExpression(
    entry.Expression.Trim().ToString())
   }
  );
 }
 ...
}
```

Die Methode erhält den Ausdruck als *object* (Zeichenkette) und stellt die *CodeExpression* zusammen. Die verweist auf die statische Methode *GetRandomColor*, die ebenfalls in der Klasse implementiert ist:

```
public static Color GetRandomColor(
  string values) {
  var colors = values.Split(',');
  var rnd = new Random();
  var color =
    colors[rnd.Next(colors.Length)];
  return Color.FromName(color);
}
```

Nun müssen Sie den neuen Ausdruckstyp nur noch in der *web.config* registrieren, und schon kann er verwendet werden:

Es geht auch ohne Fleisch

So wirklich zum Vegetarier tauge ich nicht, da sprechen die vorangegangenen 39 Ausgaben dieser Kolumne durchaus eine deutliche Sprache. Aber es geht auch ohne Fleisch, und hin und wieder lasse ich mich zu Experimenten überreden, auch wenn dem meist eine Menge Nörgelei meinerseits vorausgeht. Also ...

Chili con Tofu

Der Name ist eindeutig, und es geht wirklich. Es schmeckt sogar. Probieren Sie es also auch einmal aus!

Zunächst zwei große Zwiebeln und einige Zehen Knoblauch fein würfeln. Anschließend drei Möhren und 200 Gramm Knollensellerie schälen. Möhren und Sellerie sowie eine rote und eine grüne Paprika schälen und grob würfeln. Auch 500 Gramm Tofu geht es an den Kragen, denn der muss ebenfalls gewürfelt werden, mit etwa drei Millimetern Kantenlänge.

Lassen Sie anschließend in einem großen Topf etwas Öl heiß werden und schwitzen Sie Zwiebeln und Knoblauch an. Dann kommt der Tofu hinzu. Der hat eines mit Zucchini oder auch Auberginen gemein: Er schmeckt zunächst einmal nach nichts. Daher muss auch der Tofu gut angebraten werden.

Anschließend kommt das Gemüse in den Topf und dazu noch 800 Gramm Tomaten aus der Dose inklusive Saft – alternativ natürlich frische Tomaten. Dazu kommen ein viertel Liter Brühe und ein viertel Liter Rotwein. Hitze runter, mit Salz, Pfeffer und Sambal Oelek kräftig würzen und eine gute Stunde köcheln lassen.

Danach noch eine Dose abgespülter Kidney-Bohnen in den Topf, weitere 15 Minuten köcheln lassen, scharf abschmecken und dampfend auf den Teller. Ob Sie das fehlende Hackfleisch schmecken werden?

```
<compilation debug="true">
 <expressionBuilders>
  <add
   expressionPrefix="RandomColor"
   type="..."/>
 </expressionBuilders>
</compilation>
```

Und was macht ASP.NET später aus diesem Konstrukt? Ein kleines Stückchen Quelltext in der generierten Klassendatei. Und im Ergebnis wechselt das Label tatsächlich mehr oder weniger zufällig die Schriftfarbe.

```
@__ctrl.ForeColor =
 ((System.Drawing.Color)
 (Expressions1.
  RandomColorExpressionBuilder.
  GetRandomColor("Red,Blue,Yellow")));
```

Zugegeben, der Ansatz ist durch die Verwendung des Code DOM etwas abstrakt, aber die Flexibilität liegt auf der Hand. Da der Ausdruck zu Quelltext umgewandelt werden kann, muss dieser nicht zur Laufzeit interpretiert werden und skaliert entsprechend gut.

Der Visual Studio Webserver

Ein Tipp am Rande, um die Kolumne für diesen Monat zu einem würdevollen Abschluss zu bringen: Zum Ausprobieren der hier vorgestellten Beispiele können Sie selbstverständlich direkt die entsprechenden Visual-Studio-Projekte öffnen und ausführen. Ich finde es allerdings ganz nützlich, ASP.NET-Applikationen auch unabhängig von Visual Studio und dem IIS ausführen zu können.

Dazu starte ich den in Visual Studio integrierten Webserver kurzerhand manuell. Das gelingt zum Beispiel mit folgender Batch-Datei:

```
// start_webserver.cmd
start
C:\WINDOWS\Microsoft.NET\Framework\
 v2.0.50727\WebDev.WebServer.EXE
 /port:8084
 /path:"..."
 start http://localhost:8084
```

Der Webserver startet die gewünschte Webapplikation auf dem lokalen Rechner auf einem definierten Port (hier 8084) und ruft die Applikation anschließend im Browser auf. Nächstes Mal geht's hier weiter, denn es gibt noch eine ganze Menge nicht so bekannter Details und Funktionen von ASP.NET. **[bl]**

[1] Umrechnung arabischer und römischer Zahlen, dnpLink SL0906Kochstudio1

[2] Visual Studio 2008 Web Deployment Project, dnpLink SL0906Kochstudio2