12 Servererweiterungen

Was früher als ISAPI-Filter und ISAPI-Erweiterung mühevoll mit C++ für den IIS programmiert werden musste, ist heute Bestandteil von ASP.NET und einfacher als jemals zuvor. Lesen Sie in diesem Kapitel, wie Sie das HTTP-Handler-Konzept von ASP.NET und eigene Anwendungen erweitern.

12.1 Schnellstart

Dieser Abschnitt gibt einen kompakten Überblick über das Thema und zeigt sinnvolle Verknüpfungen mit ergänzenden und vorbereitenden Kapiteln. Der Wegweiser in die Referenz hilft, die passenden Seiten in der MSDN-Online-Referenz besonders schnell zu finden.

12.1.1 Über dieses Kapitel

Die Erweiterung der Funktionalität vom alten ASP war Sache der Programmierung der ISAPI-Schnittstelle. ISAPI steht für *Internet Services Application Programming Interface* und stellt eine Schnittstelle zum IIS dar, die – vorzugsweise – mit C++ programmiert werden konnte. Dabei gab es zwei Verfahren. ISAPI-Erweiterungen dienten der Behandlung bestimmter, individuell vergebener Datei-Erweiterungen und konnten damit fehlende Funktionen in ASP kompensieren. ISAPI-Filter dagegen wurden in den Datenstrom zwischen Browser und Client »eingehängt« und konnten diesen verändern – in beiden Richtungen.

Die Programmierkenntnisse, die ein ISAPI-Entwickler mitbringen musste, waren erheblich größer als für die ASP-Programmierung. Außerdem war eine effektive Entwicklung nur mit C++ möglich, was vermutlich die von VBScript am weitesten entfernte Sprache

ist. Insofern war die ISAPI-Programmierung nur selten Teil der täglichen Arbeit des ASP-Programmierers, obwohl es sicher zahlreiche Anwendungsfälle gegeben hätte.

Mit .NET hat sich auch die Programmierung der Schnittstellen grundlegend geändert. Da nun eine richtige Programmiersprache und eine umfassende Klassenbibliothek zur Verfügung steht, besteht kein Grund mehr, für derartige Filter oder Handler auf eine andere Sprache auszuweichen.

Auch wenn die Programmierung heute anders aussieht, gibt es dennoch direkte Entsprechungen der beiden ISAPI-Verfahren. Generell werden alle Anfragen aus dem Web von einem so genannten *HttpHandler* beantwortet. Genau diese Module kann man – verhältnismäßig einfach – auch selbst programmieren. Sie stellen dann ungefähr die Leistungen zur Verfügung, wie die alten ISAPI-Erweiterungen. Mehr dazu finden Sie in Abschnitt 12.2, »HTTP-Handler« ab Seite 1024.

Auch für die ISAPI-Filter gibt es Ersatz: *HttpModule*. Damit können sie den Datenstrom – unabhängig von anderen Programmteilen – umleiten. Wie Sie diese Module programmieren und verwenden beschreibt Abschnitt 12.3, »HTTP-Module« ab Seite 1037.

12.2 HTTP-Handler

ASP.NET verwendet generell so genannte Handler zur Abarbeitung der per HTTP eintreffenden Anfragen. Das gilt auch für die fest implementierte Laufzeitumgebung. Durch die Möglichkeit, solche Handler selbst zu programmieren, ist es leicht, ASP.NET zu erweitern.

Die Auswahl des für eine Anforderung benötigten Handlers erfolgt auf Basis des Dateinamens. Zulässig ist die Angabe von Platzhalterzeichen. Damit werden dann praktisch auch die von ASP.NET selbst verarbeiteten Dateiendungen bedient.

12.2.1 Vorhandene HTTP-Handler

Für die vorhandenen Handler ist die Konfiguration in der Datei *machine.config* verantwortlich. Der Abschnitt `<httpHandlers>` enthält die entsprechenden Angaben:

```
<configuration>
   <system.web>
     <httpHandlers>
       <add verb="*" path="trace.axd"
            type="System.Web.Handlers.TraceHandler"/>
       <add verb="*" path="*.aspx"
            type="System.Web.UI.PageHandlerFactory"/>
       <add verb="*" path="*.ashx"
            type="System.Web.UI.SimpleHandlerFactory"/>
       <add verb="*" path="*.asmx"
            type="System.Web.Services.Protocols.↵
                  WebServiceHandlerFactory, System.Web.Services,↵
                  Version=1.0.3300.0, Culture=neutral, ↵
                  PublicKeyToken=b03f5f7f11d50a3a" ↵
            validate="false"/>
       <add verb="*" path="*.rem"
            type="System.Runtime.Remoting.Channels.↵
                  Http.HttpRemotingHandlerFactory, ↵
                  System.Runtime.Remoting, ↵
                  Version=1.0.3300.0, Culture=neutral, ↵
                  PublicKeyToken=b77a5c561934e089" ↵
            validate="false"/>
       <add verb="*" path="*.soap"
            type="System.Runtime.Remoting.Channels.↵
                  Http.HttpRemotingHandlerFactory, ↵
                  System.Runtime.Remoting, ↵
                  Version=1.0.3300.0, Culture=neutral, ↵
                  PublicKeyToken=b77a5c561934e089"
            validate="false"/>
       <add verb="*" path="*.asax"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.ascx"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.config"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.cs"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.csproj"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.vb"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.vbproj"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.webinfo"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.asp"
            type="System.Web.HttpForbiddenHandler"/>
       <add verb="*" path="*.licx"
            type="System.Web.HttpForbiddenHandler"/>
```

```
        <add verb="*" path="*.resx"
             type="System.Web.HttpForbiddenHandler"/>
        <add verb="*" path="*.resources"
             type="System.Web.HttpForbiddenHandler"/>
        <add verb="GET,HEAD" path="*"
             type="System.Web.StaticFileHandler"/>
        <add verb="*" path="*"
             type="System.Web.HttpMethodNotAllowedHandler"/>
      </httpHandlers>
  </system.web>
</configuration>
```

Die Attribute

Der Abschnitt ist relativ leicht lesbar. Verknüpft wird eine Dateierweiterung (Attribut `path`) – optional auch ein konkreter Dateiname mit oder ohne Platzhalterzeichen – mit einer Klasse, die für die Verarbeitung zuständig ist (Attribut `type`). Zusätzlich kann noch mit dem Attribut `verb` bestimmt werden, welche HTTP-Kommandos akzeptiert werden. Außer dem Sternchen, das alle zulässt, können Sie hier GET, POST, DEBUG usw. einsetzen. Informationen zu HTTP finden Sie auch in Abschnitt 1.3.1, »Das Protokoll HTTP« ab Seite 54.

Bemerkenswert ist, dass auch die Abarbeitung von normalen *aspx*-Seiten lediglich durch einen Standard-Handler erfolgt – dem `Page`-Handler. Im Grunde ist der modulare Aufbau also nicht nur ein gutes Entwurfswerkzeug, sondern Basis des gesamten Unterbaus von ASP.NET. Es spricht deshalb vieles dafür, dass die Architektur auf lange Sicht leistungsfähig genug ist, auch große Änderungen am Bedarf der Softwareentwickler mitzumachen.

Verbotene Dateiendungen

Aus der gezeigten Liste erklärt sich auch, warum Dateien mit der Endung *.config* nicht verarbeitet werden können. Hier wird `System.Web.HttpForbiddenHandler` aufgerufen, eine Klasse, die den Zugriff verweigert und eine entsprechende Meldung ausgibt. Diese internen Handler sind allerdings nicht öffentlich implementiert, können also nicht verändert oder konfiguriert werden.

12.2.2 Erweiterung durch eigene Handler

Um nun eigene Handler verwenden zu können, muss die Liste erweitert werden. Selbstverständlich können Sie dies auch »pro Applikation« tun, indem der entsprechende Abschnitt der Datei *web.config* hinzugefügt wird. Normalerweise existiert dieser Eintrag

noch nicht; Sie können ihn dann unterhalb des Zweiges <system.web> anlegen.

Das genügt jedoch noch nicht, denn die Dateierweiterung muss auch im IIS verknüpft werden. Der IIS empfängt die Anforderung vom Browser immer als Erster und entscheidet nun anhand des angeforderten Dateityps, welche Applikation dafür zuständig ist. Die Verknüpfung der von ASP.NET verarbeiteten Typen wurde durch die Installation hergestellt. Andere Einträge müssen von Hand hinzugefügt werden.

12.2.3 Programmierung eines eigenen Handlers

Schaut man sich die Möglichkeiten an, die HTTP-Handler bieten, stellt man sich zuerst die Frage, wozu das zu gebrauchen ist. Generell gilt auch hier: Nicht alles, was .NET bietet, müssen Sie auch verwenden. Es geht am Anfang mehr darum, einen Überblick zu erhalten, was es alles gibt und sich dann im Bedarfsfall daran zu erinnern. Sie machen also nichts falsch, wenn Sie mit HTTP-Handlern nichts anfangen können.

Nichstdestotrotz gibt es natürlich Einsatzmöglichkeiten. Vor den ersten Listings sollten Sie sich ein wenig mit dem Prinzip auseinander setzen.

Das Prinzip der HTTP-Handler-Programmierung

Ausgangspunkt eines eigenen Handlers ist wie immer eine Klasse. Damit Sie nichts falsch machen, muss die Klasse von der Schnittstelle IHttpHandler abgeleitet werden. Sie verlangt, dass eine Eigenschaft und eine Methode zu implementieren sind:

- IsReusable

  Diese Eigenschaft vom Typ bool gibt an, ob jede Anfrage eine neue Instanz des Handlers erfordert oder nicht. Der Standardwert, den diese schreibgeschützte Eigenschaft zurückgibt, ist true. Normalerweise existiert also nur einen Instanz für alle Anfragen. Wenn Sie keinen triftigen Grund haben, das Verhalten zu ändern, belassen Sie es dabei.

- `ProcessRequest`

  Diese Methode erledigt die eigentliche Arbeit. Sie ist immer vom Typ `void`. Als Parameter wird der Kontext der aktuellen Verbindung erwartet, Typ `HttpContext`. Damit haben Sie Zugriff auf `Request`, `Response` usw.

Mehr Informationen zu `Request`, `Response` und `Context` finden Sie im Abschnitt 8.2, »Die Welt der Standardobjekte« ab Seite 753.

Nun muss ASP.NET noch mitgeteilt werden, dass Sie unter bestimmten Umständen die Verwendung des eigenen Handlers wünschen. Die »bestimmten Umstände« können anhand der Verwendung einer bestimmten Dateierweiterung oder auch eines konkreten Dateinamens erkannt werden.

Dateierweiterung

Die Erkennung einer Dateierweiterung setzt voraus, dass diese überhaupt bis zu ASP.NET vordringt. Dazu muss der IIS entsprechend instruiert werden. Gehen Sie dazu folgendermaßen vor:

1. Öffnen Sie die Managementkonsole INTERNET-INFORMATIONSDIENSTE.
2. Auf der Registerkarte VERZEICHNIS klicken Sie auf KONFIGURATION. Falls die Schaltfläche inaktiv ist, müssen Sie erst eine Anwendung erstellen. Besser ist es, eine von Visual Studio .NET erzeugte Anwendung zu verwenden.
3. Sie gelangen nun in den Dialog ANWENDUNGSKONFIGURATION. In der Liste ANWENDUNGSZUORDNUNGEN sehen Sie, wie beispielsweise die Dateierweiterung *.aspx* mit ASP.NET – konkret mit der ISAPI-DLL *aspnet_isapi.dll* – verknüpft ist. Merken Sie sich den Pfad aus dem Eingabefeld AUSFÜHRBARE DATEI des Dialogs BEARBEITEN.
4. Erzeugen Sie nun einen neuen Eintrag. Tragen Sie in dem Eingabefeld AUSFÜHRBARE DATEI den Pfad zu ASP.NET ein und unter Erweiterung *.pix*. Der IIS leitet nun alle Dateien mit der Endung *.pix* an ASP.NET weiter.

   Füllen Sie die übrigen Einstellungen so aus, wie in der folgenden Abbildung gezeigt:

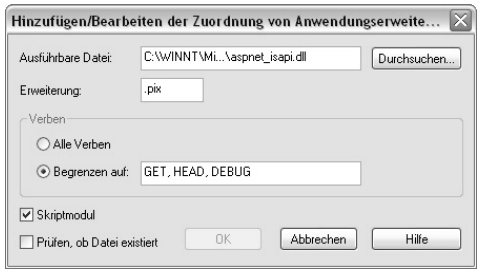

Abbildung 12.1: Dateierweiterung mit ASP.NET verbinden

Für Windows XP-Benutzer: Klicken Sie nach der Auswahl des Pfades nochmals in das Feld AUSFÜHRBARE DATEI*, damit die* OK*-Schaltfläche aktiviert wird. Im .NET-Server und unter Windows 2000 ist das nicht erforderlich.*

5. Speichern Sie alle Einstellungen.

Praktische Programmierung

Bilder dynamisch per Handler erzeugen

Bevor nun der Handler in der Datei *web.config* verknüpft wird, muss die Klasse erstellt werden. Im Beispiel geht es um das dynamische Erzeugen von Bildern. Eine Einführung in dieses Thema wurde bereits im Abschnitt 11.2, »Bilder dynamisch erstellen« ab Seite 988 gezeigt. Es geht nun darum, ein Bild dynamisch zu erstellen, indem als Bildpfad Folgendes verwendet wird:

```
<img src="bild.pix?text=Hallo%20Handler&color=red"/>
```

Dabei kommt es nur auf die Dateierweiterung an, der Name ist beliebig und die aufgerufene Datei muss nicht existieren.

Beachten Sie, dass in Abbildung 12.1 das Kontrollkästchen PRÜFEN, OB DATEI EXISTIERT*, deaktiviert wurde. Sonst funktioniert das mit der nicht existenten Datei nicht.*

Im Projekt *csharpadvanced* ist dieses Prinzip im Beispiel *PictureHandler.aspx* realisiert worden:

```
<body MS_POSITIONING="GridLayout">
<h1>Dynamische Bilder, mit Handler erzeugt</h1>
<img src="bild.pix?w=480&h=80&ff=Verdana&fs=
28&a=center&b=2&bgc=yellow&c=blue&t=Hallo%20Banner"/>
<br/>
<br/>
```

```
<br/>
<img src="bild.pix?w=48&h=48&ff=Tahoma&fs=10&bgc=
black&b=0&c=white&t=Mini"/><br/>
</body>
```

Listing 12.1: Der Aufruf der dynamisch erzeugten Bilder (PictureHandler.aspx)

Die Bedeutung der Parameter können Sie der folgenden Tabelle entnehmen:

Parameter	Bedeutung
w	Breite (Width)
h	Höhe (Height)
ff	Schriftartennamen (Font Face)
fs	Schrifthöhe in Punkt (Font Size)
c	Farbe (Color)
bgc	Hintergrundfarbe (Background Color)
bg	Hintergrundbild (Background Image)
b	Randbreite in Pixel (Border)
bc	Randfarbe (Border Color)
a	Ausrichtung des Textes (Align)
text	Der Text selber
type	Das Bildformat (GIF; JPG etc.)

Tabelle 12.1: Die Parameter des Programms zum dynamischen Erzeugen von Bildern

Freilich steckt das Know-how woanders. Das vollständige Listing finden Sie nachfolgend:

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.ComponentModel;
using System.IO;
using System.Text.RegularExpressions;
using System.Collections;

namespace Addison.CSharp.Advanced
{
  public class HandlePics : IHttpHandler
  {
```

```
private Int16 _width;
private Int16 _height;
private String _fontface;
private Double _fontsize;
private Int16 _rotate;
private String _color;
private String _bgcolor;
private String _background;
private Int16 _border;
private String _bordercolor;
private String _align;
private String _text;
private String _type;

private SolidBrush oSB1, oSB2;
private Pen oSP1;

public void ProcessRequest (HttpContext context)
{
 HttpRequest Request = context.Request;
 HttpResponse Response = context.Response;

 // extract params from querystring
 if (Request.QueryString["w"] != null)
   _width = Convert.ToInt16(Request.QueryString["w"]);
 else
   _width = 480;
 if (Request.QueryString["h"] != null)
   _height = Convert.ToInt16(Request.QueryString["h"]);
 else
   _height = 80;
 if (Request.QueryString["ff"] != null)
   _fontface = Request.QueryString["ff"];
 else
   _fontface = "Verdana";
 if (Request.QueryString["fs"] != null)
   _fontsize = Convert.ToSingle(Request.QueryString["fs"]);
 else
   _fontsize = 12.0;
 if (Request.QueryString["c"] != null)
   _color = Request.QueryString["c"];
 else
   _color = "black";
 if (Request.QueryString["bgc"] != null)
   _bgcolor = Request.QueryString["bgc"];
 else
   _bgcolor = "white";
 if (Request.QueryString["bg"] != null)
   _background = Request.QueryString["bg"];
```

```
else
  _background = String.Empty;
if (Request.QueryString["b"] != null)
  _border = Convert.ToInt16(Request.QueryString["b"]);
else
  _border = 1;
if (Request.QueryString["bc"] != null)
  _bordercolor = Request.QueryString["bc"];
else
  _bordercolor = "blue";
if (Request.QueryString["a"] != null)
  _align = Request.QueryString["a"];
else
 _align = "left";
if (Request.QueryString["t"] != null)
 _text = Request.QueryString["t"];
else
 _text = "No Text";
if (Request.QueryString["type"] != null)
 _type = Request.QueryString["type"];
else
 _type = "gif";
if (Request.QueryString["r"] != null)
 _rotate = Convert.ToInt16(Request.QueryString["r"]);
else
 _rotate = 0;

// build picture dynamically
// create bitmap
StringFormat oStringF = new StringFormat();
if ((_rotate > 45 && _rotate < 135)
|| (_rotate > 225 && _rotate < 315))
{
   Int16 _dummy = _width;
   _width = _height;
   _height = _dummy;
   oStringF.FormatFlags ↵
     = StringFormatFlags.DirectionVertical;
}
Bitmap oBM = new Bitmap (_width, _height);
Graphics oG = Graphics.FromImage (oBM);
ColorConverter cc = new ColorConverter();
oSB1 = new SolidBrush ↵
   ((System.Drawing.Color) ↵
     cc.ConvertFromString(_bgcolor));
oSP1 = new Pen ↵
   ((System.Drawing.Color) ↵
     cc.ConvertFromString(_bordercolor));
oSB2 = new SolidBrush ↵
```

```
    ((System.Drawing.Color) ↵
      cc.ConvertFromString(_color));
Font oF = new Font (_fontface, (float) _fontsize);
oG.FillRectangle (oSB1, 0, 0, ↵
                  _width-_border, _height-_border);
SizeF oSF = oG.MeasureString (_text, oF);
PointF oPF;
switch (_align)
{
  case "center":
     oStringF.Alignment = StringAlignment.Center;
     oPF = new ↵
     PointF(_width/2 - (oSF.Width/2), ↵
            _height/2 - (oSF.Height/2));
     break;
  case "right":
     oStringF.Alignment = StringAlignment.Far;
     oPF = new
     PointF(_width - oSF.Width, ↵
            _height/2 - (oSF.Height/2));
     break;
  default:
     oStringF.Alignment = StringAlignment.Near;
     oPF = new PointF(0, _height/2 - (oSF.Height/2));
     break;
}
Matrix oM = new Matrix ();
oM.Rotate (-_rotate, MatrixOrder.Prepend);
oG.Transform = oM;
oG.SmoothingMode = SmoothingMode.AntiAlias;
RectangleF oRF = new RectangleF (oPF, oSF);
oG.DrawString (_text, oF, oSB2, oRF, oStringF);
if (_border > 0)
{
  oSP1.Width = Convert.ToSingle(_border);
  oG.DrawRectangle (oSP1, ↵
     new Rectangle (0, 0, _width, _height));
}
// send picture
Response.ClearContent();
switch (_type)
{
  case "jpg":
  case "jpeg":
    Response.ContentType = "image/jpeg";
    oBM.Save (Response.OutputStream, ImageFormat.Jpeg);
    break;
  case "png":
    Response.ContentType = "image/png";
```

```
            oBM.Save (Response.OutputStream, ImageFormat.Png);
            break;
          default:
            Response.ContentType = "image/gif";
            oBM.Save (Response.OutputStream, ImageFormat.Gif);
            break;
        }
        Response.End();
        oBM.Dispose();
        oG.Dispose();
      }

      public bool IsReusable
      {
         get { return true; }
      }
   }
}
```

Listing 12.2: Der komplette Handler zum dynamischen Erzeugen von Bildern (PictureHandler.cs)

Wie es funktioniert

Auf die Aspekte der Grafikprogrammierung soll hier nicht allzu tief eingegangen werden. Interessanter ist das Prinzip des Handlers. Dazu zuerst ein Blick auf die Struktur – befreit von den Bildfunktionen:

```
using System;
// Weitere using-Anweisungen

namespace Addison.CSharp.Advanced
{
   public class HandlePics : IHttpHandler
   {

      public void ProcessRequest (HttpContext context)
      {
         HttpRequest Request = context.Request;
         HttpResponse Response = context.Response;

      }

      public bool IsReusable
      {
         get { return true; }
      }
   }
}
```

IHttpHandler

Wichtig ist die Nutzung der Schnittstelle `IHttpHandler`. Zugriff auf den Datenstrom von und zum Browser erhalten Sie über den Parameter vom Typ `HttpContext`. Die Entnahme der Objekte `Request` und `Response` ist nicht zwingend erforderlich, wird aber für Ausgaben benötigt. Sollte Ihr Handler nur eine Datenbank bedienen, können Sie darauf verzichten. Bleibt noch die Eigenschaft `IsReusable` zu erklären. Gibt diese `true` zurück, wird nur eine Instanz der Klasse angelegt und immer wieder verwendet. Das ist schneller, muss aber bei der Programmierung beachtet werden, denn so wird der Konstruktor nur einmal aufgerufen. Wenn Ihr Handler verschiedene Sitzungen bedienen soll und die Abfrage der Sitzungsdaten im Konstruktor erfolgt, wird das nur funktionieren, wenn `IsReusable` den Wert `false` zurückgibt.

Das eigentliche Grafikprogramm wird in Abschnitt 11.2.3, »Eine kleine Grafikbibliothek« ab Seite 996 genauer diskutiert. In der Anwendung im Handler werden zuerst die übergebenen Parameter im *Querystring* analysiert. Eine Übersicht enthielt bereits Tabelle 11.1. Dann wird mit Hilfe der Grafikfunktionen aus GDI+ ein rechteckiger Banner erstellt. Die Ausgabe erfolgt dann direkt an den Ausgabedatenstrom:

```
Response.ContentType = "image/gif";

oBM.Save (Response.OutputStream, ImageFormat.Gif);
```

Das funktioniert, weil der Auslöser für den Handler-Aufruf das `<img>`-Tag in der *aspx*-Seite war. Der Browser erwartet deshalb ausschließlich Bilddaten. Genau die erzeugt der Handler.

Einen Handler in *web.config* registrieren

Wenn Sie das Programm jetzt übersetzen, wird es dennoch nicht funktionieren. Denn die Dateierweiterung *.pix* gelangt zwar zu ASP.NET, wird aber nicht weiter beachtet. Sie müssen noch die Konfiguration in *web.config* anpassen.

Wenn Sie Zugriff auf den Server haben, können Sie die Einstellung auch für alle Projekte in machine.config *vornehmen.*

Tragen Sie in *web.config* Folgendes im Zweig `<system.web>` ein:

```
<configuration>
   <system.web>
```

```
      <httpHandlers>
         <add verb="GET" path="*.pix"
              type="Addison.CSharp.Advanced.HandlePics,
csharpadvanced"/>
    </httpHandlers>
```

Der Handler soll nur auf HTTP-GET reagieren – dies wird durch das Attribut `verb` bestimmt. Dann wird in `path` die Dateierweiterung verknüpft. Sie können hier auch »bild.pix« schreiben, dann ist die Wahl des Dateinamens nicht mehr frei. Bleibt als Letztes noch das Attribut `type`. Hier ist der Name der Klasse mit vollständigem Namensraum und, durch Komma getrennt, der Name der Assembly anzugeben.

Den Assemblynamen können Sie den Eigenschaften des Projekts entnehmen:

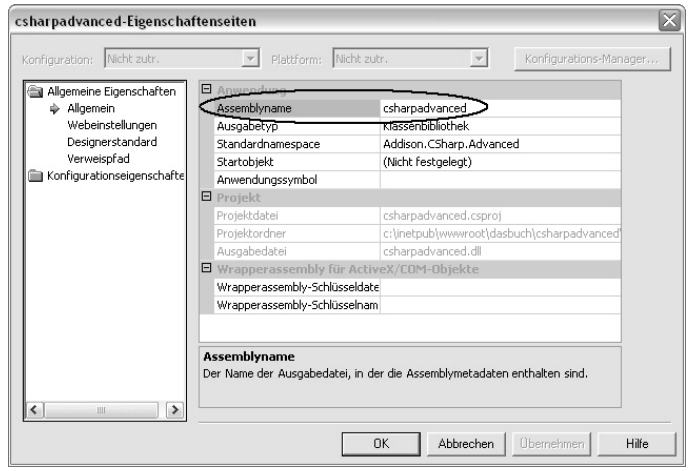

Abbildung 12.2: So ermitteln Sie den Namen einer Assembly

Danach steht einem Test nichts mehr im Wege. Das am Anfang gezeigte Beispielprogramm erzeugt folgende Ausgabe:

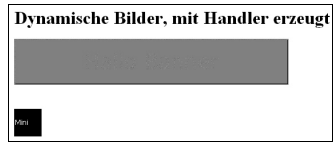

Abbildung 12.3: Über HTTP-Handler dynamisch erzeugte Bilder

12.2.4 HTTP-Handler ohne IIS-Konfiguration verwenden

Wenn Sie nun denken, dass Sie das Verfahren auch im Webspace, beim Provider, verwenden können, werden Sie schnell an eine Grenze stoßen. Denn der Provider wird vermutlich nicht zulassen, dass Sie die IIS-Konfiguration ändern. Überdies ist das auch ein zusätzlicher Schritt, den man sich jedoch ersparen kann. Wie zuvor bereits angedeutet, verträgt das Attribut `path` des Elements `<httpHandlers>` nicht nur eine Dateierweiterung, sondern auch einen vollständigen Dateinamen. Wenn Sie hier eine bereits konfigurierte Erweiterung verwenden, ist eine Änderung an der IIS-Konfiguration nicht erforderlich.

Wählen Sie in *web.config* beispielsweise folgendes Attribut:

```
path="pixx.aspx"
```

Ändern Sie nun den Aufruf in der *aspx*-Datei folgendermaßen:

```
<img src="pixx.aspx?w=480&h=80&ff=Verdana&fs=28&a=center&b=2&bgc=
yellow&c=blue&t=Hallo%20Banner"/>
...
<img src="pixx.aspx?w=48&h=
48&ff=Tahoma&fs=10&bgc=black&b=0&c=white&t=Mini"/><br/>
```

Schon funktioniert der HTTP-Handler wieder und der IIS ist aus dem Rennen.

12.3 HTTP-Module

HTTP-Module sind nicht an bestimmte Aufrufe oder die Verknüpfung mit einer Dateierweiterung gebunden. Sie agieren ähnlich wie die früheren ISAPI-Filter und beeinflussen den gesamten Ein- und Ausgabedatenstrom. Das ist dann interessant, wenn Sie globale Verhaltensweisen wünschen, aber nicht in bestehende Applikationen eingreifen möchten.

12.3.1 Grundsätzliche Funktionsweise

Ereignisorientiert

HTTP-Module funktionieren etwas anders als Handler. Auch hier wird zwar eine Klasse erstellt und in *web.config* verknüpft, aber es erfolgt kein direkter Zugriff auf `Request` oder `Response`. Stattdessen können Sie für alle Applikationsereignisse Ereignisbehandlungs-

methoden schreiben und damit den Zugriffspunkt exakt definieren. Eine Übersicht über diese Ereignisse und deren Bedeutung finden Sie in Abschnitt 8.6.1, »Einführung in das Applikationsereignismodell« ab Seite 805.

Realisiert wird ein Modul, indem Sie eine Klasse schreiben, die von der Schnittstelle `IHttpModule` ableitet. Sie müssen hier zwei Methoden implementieren:

- `void Init (HttpApplication)`

  Diese Methode initialisiert die Ereignisbehandlungsmethoden. Um den Zugriff auf das aktuelle `Application`-Objekt zu ermöglichen, wird es als Parameter übergeben.

- `void Dispose ()`

  Falls Aufräumvorgänge notwendig sind, platzieren Sie diese hier. Normalerweise bleibt diese Methode jedoch leer. Parameter werden nicht übergeben.

Die folgende Tabelle zeigt die Reihenfolge, in der die Ereignisse abgearbeitet werden:

Ereignis	Aktion
`BeginRequest`	Start der Anforderung
`AuthenticateRequest`	Authentifizierung (Benutzererkennung)
`AuthorizeRequest`	Autorisierung (Rechtvergabe)
`ResolveRequestCache`	Cache prüfen
	HTTP-Handler instanziieren (die für die Seite (Page), aber auch eigene)
`AcquireRequestState`	Status prüfen
`PreRequestHandlerExecute`	Vor der Anforderungsverarbeitung
	HTTP-Handler ausführen
`PostRequestHandlerExecute`	Nach der Ausführung, Seite fertig
`ReleaseRequestState`	Freigabe der Ressourcen
	Aufruf von Ausgabefiltern (siehe `Response.Filter`)
`UpdateRequestCache`	Cache aktualisieren
`EndRequest`	Anforderung fertig, Daten senden

Tabelle 12.2: Ereignisverarbeitung auf der Anforderungsseite

HTTP-Module und global.asax

HTTP-Module sind also auch im Hinblick auf die Prozessverarbeitung sehr leistungsfähig. Sie können nämlich ihre Leistungen zu jedem Zeitpunkt der Prozessverarbeitungskette anbieten. Vergleichen Sie aber in diesem Zusammenhang auch die Möglichkeiten, die Eingriffe in die Datei *global.asax* bieten. Mehr dazu finden Sie im Abschnitt 8.6.2, »Die Datei global.asax« ab Seite 812.

Registrierung des Moduls in der Datei *web.config*

Ähnlich wie bei den HTTP-Handlern muss auch hier eine Registrierung in *web.config* erfolgen. Kennen sollten Sie den Namensraum, den Namen der Klasse und den der Assembly, um das Element `<httpModules>` entsprechend zu konfigurieren:

```
<configuration>
  <system.web>
    <httpModules>
      <add type="Addison.CSharp.Advanced.CopyrightModule, ↵
                 csharpadvanced" ↵
           name="CopyrightModule"/>
    </httpModules>
```

Das Attribut `type` nimmt den Namen der Klasse auf und – durch Komma getrennt – den Namen der Assembly. Abbildung 12.2 zeigte bereits, wie Sie den Assemblynamen ermitteln können. In `name` wird außerdem noch ein Name des Moduls vergeben. Empfehlenswert ist es, den Klassennamen zu verwenden.

12.3.2 Anwendungsbeispiel

Als Beispiel soll hier gezeigt werden, wie Sie verdeckte Copyright-Informationen in alle gesendeten Seiten einfügen können:

```
using System;
using System.Web;

namespace Addison.CSharp.Advanced
{
   public class CopyrightModule : IHttpModule
   {
      private string copyrightBefore ↵
         = "<!-- (c) 2002 by Joerg Krause, ↵
                          Uwe Buenning -->\n\n";
      private string copyrightAfter ↵
         = "\n\n<!--  " ↵
```

```
            + System.DateTime.Now.ToLongDateString() + "-->";

        private void CopyrightBefore (object sender, ↵
                                      EventArgs e)
        {
            HttpApplication ha = (HttpApplication) sender;
            ha.Context.Response.Write (copyrightBefore);
        }
        private void CopyrightAfter (object sender, ↵
                                     EventArgs e)
        {
            HttpApplication ha = (HttpApplication) sender;
            ha.Context.Response.Write (copyrightAfter);
        }

        public void Init (HttpApplication app)
        {
            app.BeginRequest ↵
             += new EventHandler (CopyrightBefore);
            app.EndRequest ↵
             += new EventHandler (CopyrightAfter);
        }

        public void Dispose ()  { }

    }
}
```

Listing 12.3: Ein HTTP-Modul (CopyrightModule.cs)

Wie es funktioniert

Aufgabe dieses Programm ist es, oberhalb und unterhalb jeder gesendeten Seite einen HTML-Kommentar einzufügen. Dazu wird in der Methode `Init` zuerst eine Ereignisbehandlung für den Anfang der Anforderung definiert:

```
app.BeginRequest += new EventHandler (CopyrightBefore);
```

Dann noch eine für das Ende:

```
app.EndRequest += new EventHandler (CopyrightAfter);
```

Die Methode *CopyrightBefore* greift nun auf das `Application`-Objekt zu:

```
HttpApplication ha = (HttpApplication) sender;
```

Sie gibt dann mit `Response.Write` den gewünschten Text aus:

```
ha.Context.Response.Write (copyrightBefore);
```

Am Ende der Anforderung funktioniert das vergleichbar. Ein Blick in eine beliebige Seite des Projekts verrät, dass es funktioniert hat:

Abbildung 12.4: Die Wirkung des Moduls im Quelltext einer Seite

HTTP-Module sind ein sicheres Mittel, allgemeine Vorgänge, die unabhängig vom Benutzer sind, zu genau definierten Zeitpunkten auszuführen. Dazu gehören beispielsweise auch eigene Implementierungen für eine Authentifizierung oder die Steuerung der Daten des Ein- oder Ausgabestromes.