PROGRAMMER'S CHOICE

Christian Wenz
Tobias Hauser
Karsten Samaschke
Jürgen Kotz

ASP.NET 3.5 mit Visual C# 2008

Leistungsfähige Webapplikationen programmieren

DVD

> Visual Studio 2008 Express Editions
> Alle Beispiele aus dem Buch

ADDISON-WESLEY

4 Formulare mit HTML Controls

Die wichtigsten Aufgaben bei jeder serverseitigen Webskriptsprache sind Abfrage, Auswertung und Verarbeitung von Formulardaten. Anwendungsgebiete hierfür gibt es viele:

- Feedback-Formulare, mit denen der Nutzer Rückmeldungen über die Website tätigen kann. Der Benutzer muss dazu nicht extra sein E-Mail-Programm starten und unter Umständen nicht einmal seine E-Mail-Adresse preisgeben. Damit wird eine wichtige Hemmschwelle überschritten, und die Chancen, dass Sie wertvolle Rückmeldungen Ihrer Besucher erhalten, steigen.
- Support-Formulare, mit denen der Benutzer technische Anfragen stellen kann. Der Zwang, die Daten in verschiedene, exakt spezifizierte Formularelemente einzugeben, liefert Ihnen bei der Auswertung Vorteile; so kommen Sie schneller an die gewünschten Daten, als wenn Sie eine Freitext-E-Mail interpretieren und dort die interessanten Inhalte extrahieren müssen.
- Web-Front-Ends (also Masken) für andere Anwendungen, beispielsweise Gästebücher.

ASP.NET bietet mehrere Möglichkeiten, auf Formulardaten zuzugreifen. In diesem und den folgenden Kapiteln werden wir alle vorstellen. Die erste Möglichkeit ist noch von ASP bekannt und wird hauptsächlich aus Gründen der Abwärtskompatibilität beibehalten. Die neuen Möglichkeiten von ASP.NET, insbesondere die mögliche strikte Trennung von Code und Content, bieten dem Programmierer weitere Ansätze der Formulargestaltung und -verarbeitung. Zum einen ist es möglich, bekannte HTML-Formularelemente serverseitig neu zu beleben, und zum anderen bietet ASP.NET neue, eigene Elemente, die in Formularcode umgesetzt werden.

Die Hauptanwendungen beim Formularzugriff sind folgende:

- Zugriff auf Formulardaten
- Vollständigkeitsüberprüfung
- Vorausfüllung, falls das Formular zuvor nicht komplett ausgefüllt wurde

Alle diese Punkte werden wir im Folgenden mit den verschiedenen Möglichkeiten von ASP.NET für Formulare behandeln. Zunächst stellen wir den »alten« Weg vor, auf Formulardaten mit ASP.NET zuzugreifen. Das funktioniert wunderbar, ist für einige Anwendungen immer noch sehr praktisch, nutzt aber einige der eingebauten Vorteile von ASP.NET nicht aus. In Abschnitt 4.2 erfahren Sie dann, wie ASP.NET eine Brücke zwischen der herkömmlichen Formularbehandlung und HTML zu schlagen versucht.

4.1 Formulardaten von Hand

Wer schon einmal mit ASP programmiert hat, weiß bereits, dass der Zugriff auf Formulardaten sehr einfach über ein spezielles Objekt von ASP erfolgt: das `Request`-Objekt. Generell kann über `Request("xyz")` (Visual Basic) bzw. `Request["xyz"]` (C#) auf den Wert in dem Formularfeld zugegriffen werden, das als `name`-Attribut `"xyz"` hat.

Betrachten wir ein einfaches HTML-Textfeld:

```
<input type="text" name="Login" />
```

Der Wert in diesem Formularfeld steht nach dem Versand in `Request("xyz")` (VB) bzw. `Request["xyz"]` (C#).

4.1.1 Versandmethode

Für den Versand von HTML-Formularen über das World Wide Web gibt es zwei gängige Methoden:

- GET und
- POST

Standardmäßig wird GET verwendet. Das bedeutet, dass die Formulardaten in der URL übergeben, also dort angehängt werden. Sie können das mit einer einfachen, statischen HTML-Seite ausprobieren. Nachfolgend finden Sie ein HTML-Formular mit ein paar Feldern:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Formular</title>
</head>
<body>
  <form>
    Textfeld:
    <input type="text" name="Textfeld" />
    <br />
    Passwortfeld:
```

```
    <input type="password" name="Passwortfeld" />
    <br />
    Mehrzeiliges Textfeld
    <textarea name="Mehrzeilig"></textarea>
    <br />
    Checkbox
    <input type="checkbox" name="Checkbox" value="an" />
    <br />
    Radiobutton
    <input type="radio" name="Radio" value="r1" />1
    <input type="radio" name="Radio" value="r2" />2
    <br />
    Auswahlliste
    <select name="Auswahlliste" size="3" multiple="multiple">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Listing 4.1: Ein einfaches HTML-Formular ohne Skriptcode (*formular.html*)

In Abbildung 4.1 sehen Sie das Formular sowie einige Beispielwerte, die wir eingetragen haben. Wenn Sie das Formular verschicken, wird es neu geladen, aber die zuvor eingegebenen Formularwerte sind verschwunden.

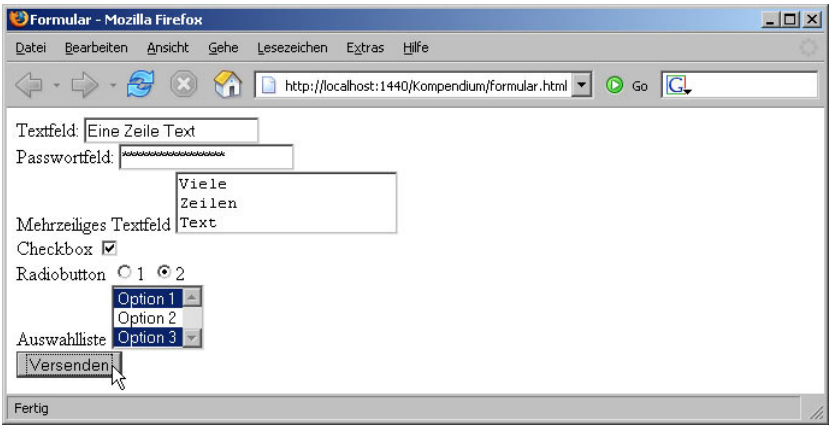

Abbildung 4.1: Das HTML-Formular mit ein paar Beispielwerten

Was ist passiert? Nun, die Formulardaten wurden per GET verschickt. Die Bezeichnungen der Formularfelder (das sind die `name`-Attribute) und die dort eingegebenen Werte wurden im Format `Name=Wert` an die URL angehängt. Die einzelnen dieser sogenannten *Name-Wert-Paare* werden durch das kaufmännische Und (`&`) voneinander getrennt.

Hier die komplette URL, die beim Versand des obigen Formulars aufgerufen wurde:

```
http://localhost:1440/Kompendium/formular.html?Textfeld=Eine+Zeile+Text&Passwortfeld=
v%F6llig+unsichtbar&Mehrzeilig=Viele%0D%0AZeilen%0D%0AText&Checkbox=
an&Radio=r2&Auswahlliste=o1&Auswahlliste=o3
```

Da dies ein wenig unübersichtlich ist, haben wir die URL in mehrere einzelne Zeilen aufgebrochen. Dazu schreiben wir jeweils daneben, welchem Formularwert welcher Ausschnitt der URL entspricht:

- `http://localhost:1440/Kompendium/formular.html` – die URL des Formulars
- `?Textfeld=Eine+Zeile+Text` – das einzeilige Textfeld
- `&Passwortfeld=v%F6llig+unsichtbar` – das Passwortfeld
- `&Mehrzeilig=Viele%0D%0AZeilen%0D%0AText` – das mehrzeilige Textfeld
- `&Checkbox=an` – die Checkbox
- `&Radio=r2` – der Radiobutton
- `&Auswahlliste=o1&Auswahlliste=o3` – die Auswahlliste

Wir können also festhalten:

- Name und Werte werden durch Gleichheitszeichen voneinander getrennt.
- Die einzelnen Name-Wert-Paare werden durch das kaufmännische Und (`&`) voneinander getrennt.
- Die ganzen Name-Wert-Paare werden mit einem vorangestellten Fragezeichen (?) an den Namen der Datei (hier: *formular.html*) angehängt.
- Sonderzeichen werden besonders maskiert. Aus Leerzeichen werden Pluszeichen, andere Sonderzeichen werden durch `%` und ihren hexadezimalen Zeichencode ersetzt. Beispielsweise hat ein Zeilensprung den Zeichencode 13, hexadezimal 0D. Also entspricht `%0D` einem Zeilensprung.

Dies ist der Versand per GET. Diese Methode ist zum Testen sehr bequem, sind doch aus der URL alle relevanten Daten ersichtlich. In der Praxis wird jedoch meistens auf GET verzichtet, von Suchmaschinen einmal abgesehen. GET hat nämlich eine Reihe von Nachteilen:

- Die Länge einer URL ist bei Browsern, Proxy-Servern und Webservern begrenzt. Einige Systeme erlauben nur maximal 500 Zeichen URL, aber die meisten Softwareprodukte machen spätestens bei 2000 Zeichen dicht.
- Aus der URL sind die kompletten Formulardaten ersichtlich. Diese URL wird in der History-Liste (Netscape) bzw. in der Verlaufsliste (Internet Explorer) des Browsers gespeichert und ist in Firmennetzwerken zumeist auch aus dem Proxy-Log ermittelbar. Sensible Daten wie Passwörter oder Kreditkartennummern sind somit unter Umständen von Dritten einsehbar.

POST hat diese Nachteile nicht. Hier werden die Formulardaten zunächst auch in Name-Wert-Paare umgewandelt. Allerdings werden diese Daten dann nicht an die URL angehängt, sondern in der HTTP-Anforderung an den Webserver hinter dem HTTP-Header untergebracht. Im obigen Beispiel könnte dann die HTTP-Anforderung beispielsweise folgendermaßen aussehen:

```
POST /Kompendium/formular.html HTTP/1.1
Host: localhost:1440
User-agent: Mozilla/47.11
Content-length: 286
Content-type: application/x-www-form-urlencoded

Textfeld=Eine+Zeile+Text&Passwortfeld=v%F6llig+unsichtbar&Mehrzeilig=Viele%0D%0AZeilen%
0D%0AText&Checkbox=an&Radio=r2&Auswahlliste=o1&Auswahlliste=o3
```

Sie sehen also: Zunächst erscheinen die HTTP-Header-Informationen, dann eine Leerzeile und dann die Formulardaten.

Damit dies auch tatsächlich so funktioniert, müssen Sie im HTML-`<form>`-Tag den `method`-Parameter auf `"post"` setzen:

```
<form method="post">
```

*** * * TIPP**

Das `<form>`-Element kennt den Parameter `action`, in dem das Skript angegeben werden kann, an das die Formulardaten verschickt werden müssen. Wenn Sie jedoch den Parameter nicht angeben (wie in den vorherigen Beispielen geschehen), werden die Formulardaten an die aktuelle Datei verschickt. Diese »Abkürzung« werden wir in den nächsten Kapiteln noch häufiger verwenden. Wenn Sie ganz korrekt sein möchten, verwenden Sie `action=""`, was denselben Effekt erzielt.

Aber zurück zu ASP.NET und dem `Request`-Objekt. Im `Request`-Objekt selbst finden Sie sowohl POST- als auch GET-Daten und sogar Cookies (sowie weitere Werte, auf die wir an dieser Stelle nicht eingehen möchten). Im Sinne einer sauberen Entwicklung macht es jedoch sehr viel Sinn, explizit auf POST- oder auf GET-Werte zuzugreifen. Aus diesem Grund gibt es innerhalb des `Request`-Objekts Unterkollektionen, eine speziell für GET und eine speziell für POST:

- Über `Request.QueryString` greifen Sie auf GET-Daten zurück (die hinter dem Fragezeichen an eine URL angehängten Daten werden im Englischen als *Query-String* bezeichnet).
- Über `Request.Form` greifen Sie auf POST-Daten zurück. Das rührt daher, dass POST-Daten nur per Formular zustande kommen, während GET-Daten auch von Hand erzeugt worden sein könnten (indem einfach eine URL mit angehängtem Query-String eingegeben wird).

*** * * TIPP**

Die beiden Eselsbrücken, GET/Query-String und POST/Formular, sollten Ihnen helfen, Verwechslungen zu vermeiden.

Doch nun genug der langen Vorrede – werfen wir einen Blick auf die verschiedenen Formularfelder und wie Sie mit ASP.NET darauf zugreifen können.

4.1.2 Formularfelder

Im Allgemeinen können Sie über `Request.Form["xxx"]` auf den Wert im Formularelement mit `name`-Attribut `"xxx"` zugreifen. Je nach Formularfeldtyp gibt es jedoch ein paar Besonderheiten; daher werden wir die einzelnen Feldtypen jeweils explizit aufführen und untersuchen.

i i i INFO

Wir verwenden im Folgenden jeweils den Formularversand per POST. Wenn Sie stattdessen auf GET setzen, müssen Sie alle Vorkommen von `Request.Form` *durch* `Request.QueryString` *ersetzen.*

Textfeld

Ein Textfeld wird durch folgendes HTML-Element dargestellt:

```
<input type="text" name="Feldname" />
```

Über `Request.Form["Feldname"]` können Sie dann auf den Text im Formularfeld zugreifen.

Im nachfolgenden Beispiel enthält das Listing ein Textfeld; nach dem Formularversand wird der eingegebene Text ausgegeben. Der zugehörige Code wird ausnahmsweise nicht in einem serverseitigen `<script>`-Block am Anfang der Seite ausgegeben, sondern direkt mitten auf der Seite, mit `<% ... %>`. Diese Art des »Spaghetti-Codes« ist mittlerweile verpönt und wird nur noch recht selten gebraucht (manchmal in Verbindung mit der Anzeige von Datenbankdaten), aber an dieser Stelle erläutert er recht schön das Konzept. Wie gesagt, die von Microsoft empfohlene Ansteuerung von Formulardaten folgt in Abschnitt 4.2.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <%  Response.Write(HttpUtility.HtmlEncode(
    Request.Form["Feldname"])); %>
  <form method="post">
    <input type="text" name="Feldname" /><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Listing 4.2: Der Inhalt des Textfeldes wird ausgegeben (*textfeld.aspx*).

! ! ! ACHTUNG

Mit `HttpUtility.HtmlEncode()` *wandeln Sie gefährliche Sonderzeichen in der Eingabe in entsprechendes HTML um; so wird zum Beispiel aus der öffnenden spitzen Klammer < die zugehörige HTML-Entität* `&lt;`*. Mehr Informationen zum Thema Sicherheit erhalten Sie in Kapitel 28.*

*** * * TIPP**

Für einen schnelleren Zugriff auf die Methode `HtmlEncode` *können Sie den Namensraum* `System.Web.HttpUtility` *wie folgt importieren:*

```
<%@ Import Namespace="System.Web.HttpUtility" %>
```

Sie können dann direkt über `HtmlEncode` *auf die Methode zugreifen.*

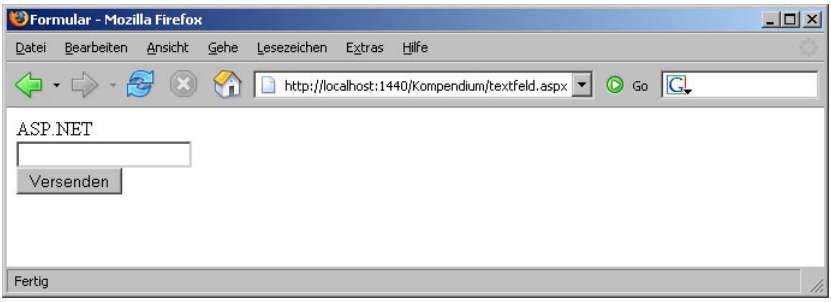

Abbildung 4.2: Der eingegebene Wert wird über dem Textfeld angezeigt.

Bei den folgenden Beispielen können Sie analog testen; wir verzichten dort auf ausführlichere Erklärungen und setzen bei der Skriptsprache wieder verstärkt auf Visual Basic.

Passwortfeld

Die HTML-Darstellung eines Passwortfeldes ist folgende:

```
<input type="password" name="Feldname" />
```

Auch hier können Sie über `Request.Form("Feldname")` (VB) bzw. über `Request.Form["Feldname"]` (C#) auf den Feldinhalt zugreifen. Hier ein komplettes Listing:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <%  Response.Write(HttpUtility.HtmlEncode(
    Request.Form["Feldname"])); %>
  <form method="post">
    <input type="password" name="Feldname" /><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Listing 4.3: Der Inhalt des Passwortfeldes wird ausgegeben (*passwortfeld.aspx*).

Mehrzeiliges Textfeld

Ein mehrzeiliges Textfeld wird in HTML durch `<textarea>` und `</textarea>` eingeschlossen:

```
<textarea name="Feldname"></textarea>
```

Der Text im mehrzeiligen Feld kann wie gehabt über `Request.Form("Feldname")` bzw. `Request.Form["Feldname"]` ermittelt werden, je nachdem, ob Sie VB oder C# einsetzen. Hier ein Beispiellisting in C#:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <% Response.Write(HttpUtility.HtmlEncode(
     Request.Form["Feldname"])); %>
  <form method="post">
    <textarea name="Feldname"></textarea><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Listing 4.4: Der Inhalt des mehrzeiligen Feldes wird ausgegeben (*mehrzeilig.aspx*).

Checkbox

Eine Checkbox hat einen (eindeutigen) Namen und einen zugehörigen Wert. In HTML wird das durch die Parameter `name` und `value` ausgedrückt:

```
<input type="checkbox" name="Feldname" value="an" />
```

Code

Nach dem Versand des Formulars enthält `Request.Form["Feldname"]` *(C#) bzw.* `Request.Form("Feldname")` *(VB) den Wert* `"an"`*, wenn die Checkbox aktiviert worden ist*[a]*, andernfalls enthält er eine leere Zeichenkette.*

a. Hätte die Checkbox kein `value`-Attribut besessen, wäre beim Ankreuzen als Wert »on« übertragen worden.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Formular</title>
</head>
<body>
 <% Response.Write(HttpUtility.HtmlEncode(
 Request.Form["Feldname"])); %>
 <form method="post">
 <input type="checkbox" name="Feldname" value="an" /><br />
 <input type="submit" value="Versenden" />
 </form>
</body>
</html>
```

Listing 4.5: Der Name der Checkbox wird ausgegeben, falls aktiviert (*checkbox.aspx*).

Radiobutton

Genau wie eine Checkbox ist auch ein Radiobutton entweder aktiviert (»angekreuzt«) oder nicht. Der große Unterschied ist, dass Radiobuttons in Gruppen unterteilt werden. Von allen Radiobuttons einer Gruppe kann immer nur maximal einer aktiviert sein. Es ist natürlich auch möglich, dass keiner der Radiobuttons aktiviert werden kann.

*** * * TIPP**

Wenn auf jeden Fall ein Radiobutton aktiviert werden soll, sollten Sie einen der Radiobuttons einer Gruppe vorauswählen. Die meisten Browser ermöglichen es dem Benutzer nicht, einen Radiobutton zu deselektieren. Die einzige Möglichkeit besteht darin, einen anderen Radiobutton aus derselben Gruppe zu aktivieren.

Hier der HTML-Code für einen Radiobutton:

```
<input type="radio" name="Feldname" value="Button" />
```

Ist dieser Radiobutton aktiviert, so enthält `Request.Form("Feldname")` den Wert `"Button"`. Wenn Sie C# verwenden, müssen Sie dementsprechend auf `Request.Form["Feldname"]` zurückgreifen.

Der Name der Gruppe von Radiobuttons wird im `name`-Attribut angegeben. Alle Radiobuttons mit dem gleichen `name`-Attribut gehören zu einer Gruppe, und nur einer dieser Buttons kann aktiviert werden. Die einzelnen Radiobuttons unterscheiden sich also anhand des `value`-Attributs.

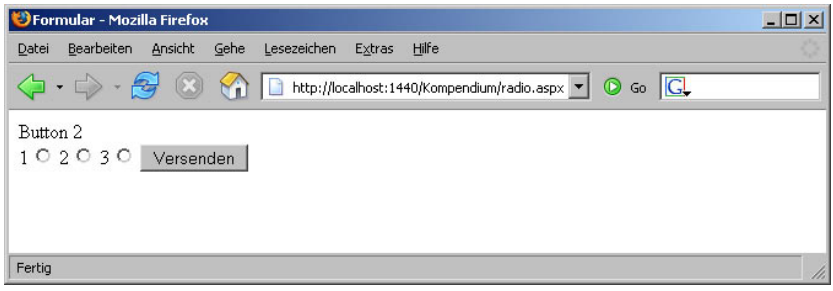

Abbildung 4.3: Der mittlere Radiobutton war beim Versand aktiviert.

Hier ein Listing, in dem der Name (in diesem Fall das `value`-Attribut) des ausgewählten Radiobuttons ausgegeben wird:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <%Response.Write(HttpUtility.HtmlEncode(
    Request.Form("Feldname")));%>
  <form method="post">
    1<input type="radio" name="Feldname" value="Button 1" />
```

```
    2<input type="radio" name="Feldname" value="Button 2" />
    3<input type="radio" name="Feldname" value="Button 3" />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Listing 4.6: Der Name des gewählten Radiobuttons wird ausgegeben (*radio.aspx*).

Auswahlliste

In einer Auswahlliste, auch Drop-down-Menü genannt, kann der Benutzer einen oder mehrere Einträge auswählen, je nachdem, welche Einstellung im HTML-Code verwendet wird. Es gibt die folgenden Möglichkeiten:

- Der folgende Code erzeugt eine Drop-down-Liste, aus der ein Element ausgewählt werden kann:

```
<select name="Feldname">
  <option value="Element 1">1. Element</option>
  <option value="Element 2">2. Element</option>
  <option value="Element 3">3. Element</option>
</select>
```

- Der folgende Code erzeugt eine Auswahlliste mit drei Elementen (davon alle zu Anfang sichtbar); trotzdem kann nur eines oder keines der Elemente ausgewählt werden:

```
<select name="Feldname" size="3">
  <option value="Element 1">1. Element</option>
  <option value="Element 2">2. Element</option>
  <option value="Element 3">3. Element</option>
</select>
```

- Folgender Code erzeugt schließlich eine Auswahlliste der Höhe 3; mithilfe der Tasten ⇧ und Strg (sowie der Maus) können beliebig viele Elemente ausgewählt werden:

```
<select name="Feldname" size="3" multiple="multiple">
  <option value="Element 1">1. Element</option>
  <option value="Element 2">2. Element</option>
  <option value="Element 3">3. Element</option>
</select>
```

Der Zugriff auf das oder die ausgewählten Elemente geschieht wie bei den vorherigen Formularfeldern auch über `Request.Form("Feldname")` bzw. `Request.Form["Feldname"]`. Es ist nun interessant zu untersuchen, welcher Unterschied bei Einfach- und Mehrfachlisten besteht; letztere Listen sind jene mit dem Attribut `multiple` im `<select>`-Tag. Das nachfolgende Listing bietet daher zwei Auswahllisten an.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
```

```
<%
  Response.Write("Liste 1: " +
    HttpUtility.HtmlEncode(Request.Form["Liste 1"]));
  Response.Write("<br />");
  Response.Write("Liste 2: " +
    HttpUtility.HtmlEncode(Request.Form["Liste 2"]));
%>
<form method="post">
  <select name="Liste 1">
    <option value="Element 1">1. Element</option>
    <option value="Element 2">2. Element</option>
    <option value="Element 3">3. Element</option>
  </select>
  <br />
  <select name="Liste 2" size="3" multiple="multiple">
    <option value="Element 1">1. Element</option>
    <option value="Element 2">2. Element</option>
    <option value="Element 3">3. Element</option>
  </select>
  <br />
  <input type="submit" value="Versenden" />
</form>
</body>
</html>
```

Listing 4.7: Die Namen der gewählten Listenelemente werden ausgegeben (*liste.aspx*).

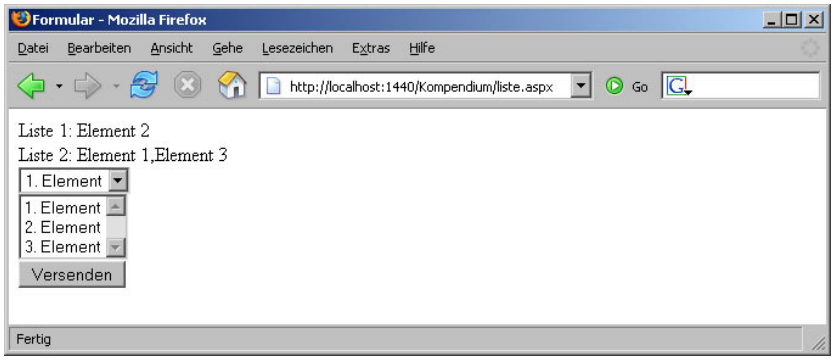

Abbildung 4.4: Die ausgewählten Elemente werden angezeigt.

Wie Sie in Ihrem Webbrowser oder in Abbildung 4.4 sehen können, wird jeweils der oder die `value`-Parameter der entsprechenden Listenelemente ausgegeben. Bei mehreren Elementen (also bei <`select multiple="multiple"`>) werden diese Werte durch Kommata voneinander getrennt.

! ! ! ACHTUNG

Wenn Sie bei Ihren Listenelementen (<`option`>*-Element) das* `value`*-Attribut weglassen, übermitteln die meisten Browser an seiner Stelle die Beschriftung des Elements, also den Text zwischen* <`option`> *und* </`option`>*. Verlassen Sie sich aber nicht darauf – und setzen Sie immer den* `value`*-Parameter.*

Datei-Upload

Ein oft vergessenes Formularfeld ist das zum Upload von Dateien:

```
<input type="file" name="Feldname" />
```

Mit den herkömmlichen Mitteln von ASP war es nicht möglich, auf diese Formularwerte zuzugreifen. Es kursierten zwar einige mögliche Lösungen im Web, die aber zumindest bei größeren Dateianhängen allesamt versagten. Für diesen Zweck musste eine Third-Party-Komponente angeschafft werden. ASP.NET bietet hierfür einen Ausweg. Dazu benötigen Sie aber Techniken, die erst in Abschnitt 4.2 vorgestellt werden; über das `Request`-Objekt direkt ist das nicht bzw. nur mit großem Aufwand möglich.

Damit haben Sie einen Überblick über alle relevanten Formularfeldtypen erhalten.

i i i INFO

Die folgenden Feldtypen haben unter anderem gefehlt:

- `<input type="hidden" />` *– verstecktes Formularfeld, funktioniert analog zu Textfeldern.*
- `<input type="submit" />` *– Versendeschaltfläche, wird in der Regel nicht serverseitig abgefragt, funktioniert aber analog zu Textfeldern.*
- `<input type="image" />` *– Versendegrafik, funktioniert wie eine Versendeschaltfläche, übergibt aber gleichzeitig in* `<Name>.x` *und* `<Name>.y` *die relativen Koordinaten des Mausklicks.*
- `<input type="button" />`, `<input type="reset" />` *– Diese Schaltflächen lösen keinen Formularversand aus.*

4.1.3 Ausgabe aller Formularangaben

Als letztes Beispiel wollen wir alle Daten im Formular ausgeben. Diese recht trivial klingende Aufgabe ist in ähnlicher Form Bestandteil vieler Skripte. Anstelle der Ausgabe der einzelnen Werte werden Sie im Praxisbetrieb die angegebenen Daten beispielsweise in einer Datenbank abspeichern.

Die naheliegendste Möglichkeit besteht darin, für jedes Formularfeld von Hand `Request.Form["Feldname"]` bzw. `Request.Form("Feldname")` auszugeben. Wir wollen an dieser Stelle einen bequemeren, aber nicht ganz so flexiblen Weg gehen. Per `For-Each`-Schleife bzw. `for-in`-Schleife werden alle Formularwerte ausgegeben.

Hier das Codestück, wie es mit Visual Basic realisiert werden könnte:

```
foreach (string element in Request.Form) {
  Response.Write("<b>" +
    HttpUtility.HtmlEncode(element) +
    ":</b> ");
  Response.Write(
    HttpUtility.HtmlEncode(Request.Form[element]));
  Response.Write("<br />");
}
```

Der komplette VB-Code sieht dann folgendermaßen aus – wir verwenden ein spartanisches Formular mit allen wichtigen Feldtypen:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
```

```
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <%
    foreach (string element in Request.Form) {
      Response.Write("<b>" +
        HttpUtility.HtmlEncode(element) +
        ":</b> ");
      Response.Write(
        HttpUtility.HtmlEncode(Request.Form[element]));
      Response.Write("<br />");
    }
  %>
  <form method="post">
    Textfeld:
    <input type="text" name="Textfeld" />
    <br />
    Passwortfeld:
    <input type="password" name="Passwortfeld" />
    <br />
    Mehrzeiliges Textfeld
    <textarea name="Mehrzeilig"></textarea>
    <br />
    Checkbox
    <input type="checkbox" name="Checkbox" value="an" />
    <br />
    Radiobutton
    <input type="radio" name="Radio" value="r1" />1
    <input type="radio" name="Radio" value="r2" />2
    <br />
    Auswahlliste
    <select name="Auswahlliste" size="3" multiple="multiple">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Listing 4.8: Alle Formulardaten werden ausgegeben (*ausgabe.aspx*).

Nun ist es aber in der Regel so, dass Sie nur die Formulardaten ausgeben möchten, das Formular jedoch nicht. Ein Ansatz besteht darin, als Ziel des Formulars (Parameter `action`) eine eigene *.aspx*-Seite zu verwenden. Dies hat jedoch den Nachteil, dass Sie beim Fehlen von Pflichtfeldern das Formular nicht erneut anzeigen können – Sie befinden sich ja mittlerweile auf einer anderen ASP.NET-Seite.

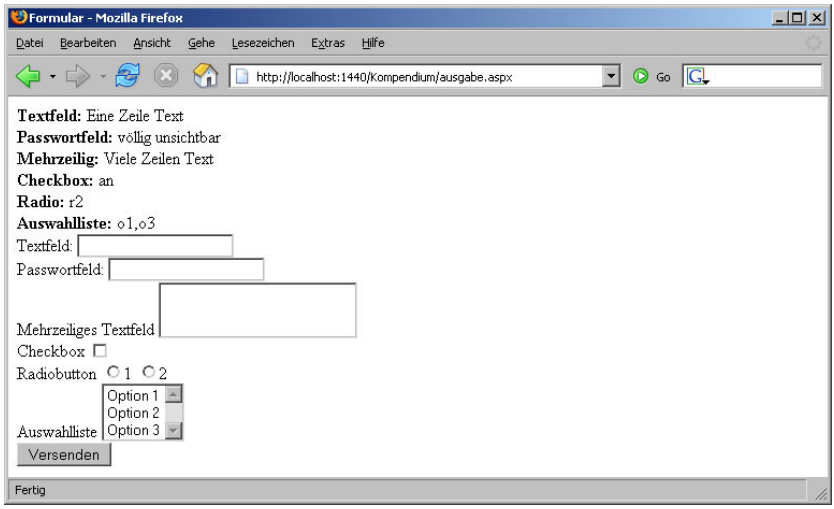

Abbildung 4.5: Die Formulardaten werden per Schleife ausgegeben.

Aus diesem Grund wird zumeist ein anderes Vorgehen gewählt. Zunächst erhält die Schaltfläche zum Verschicken des Formulars ein `name`-Attribut, was normalerweise nicht erforderlich ist:

```
<input type="submit" name="Submit" value="Versenden" />
```

Der Vorteil: Wenn das Formular verschickt wird, enthält `Request.Form["Submit"]` (bzw. bei VB `Request.Form("Submit")`) den Wert `"Versenden"`. So kann also einfach überprüft werden, ob das Formular gerade verschickt wurde (dann: Formulardaten ausgeben) oder nicht (dann: das nackte Formular anzeigen):

```
if (Request.Form["Submit"] == "Versenden") {
  //' Formulardaten per For-Each-Schleife ausgeben
} else {
  // Formular ausgeben
}
```

Exkurs

Nachfolgend das entsprechende VB-Listing:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
<%
    if (Request.Form["Submit"] == "Versenden") {
      foreach (string element in Request.Form) {
        Response.Write("<b>" +
```

```
            HttpUtility.HtmlEncode(element) +
            ":</b> ");
          Response.Write(
            HttpUtility.HtmlEncode(Request.Form[element]));
          Response.Write("<br />");
        }
      } else {
    %>
    <form method="post">
      Textfeld:
      <input type="text" name="Textfeld" />
      <br />
      Passwortfeld:
      <input type="password" name="Passwortfeld" />
      <br />
      Mehrzeiliges Textfeld
      <textarea name="Mehrzeilig"></textarea>
      <br />
      Checkbox
      <input type="checkbox" name="Checkbox" value="an" />
      <br />
      Radiobutton
      <input type="radio" name="Radio" value="r1" />1
      <input type="radio" name="Radio" value="r2" />2
      <br />
      Auswahlliste
      <select name="Auswahlliste" size="3" multiple="multiple">
        <option value="o1">Option 1</option>
        <option value="o2">Option 2</option>
        <option value="o3">Option 3</option>
      </select>
      <br />
      <input type="submit" name="Submit" value="Versenden" />
    </form>
    <%
      }
    %>
  </body>
  </html>
```

Listing 4.9: Entweder das Formular oder die Daten werden ausgegeben (*ausgabe-gezielt.aspx*).

i i i INFO

Wo sind die Umbrüche aus dem mehrzeiligen Feld hin? Die sind zwar immer noch da, doch ein Umbruch in HTML wird im Browser als Leerzeichen angezeigt. Sie müssen also alle Zeilensprünge (in .NET durch `System.Environment.NewLine` *dargestellt) durch das entsprechende HTML-Markup (*`<br />`*) ersetzen.*

Die nächste nahe liegende Anwendung besteht darin, die Formulardaten zu prüfen: Sind alle Felder ausgefüllt? Sind die eingegebenen Werte sinnvoll? Doch hier selbst Hand anzulegen und das zu programmieren wäre unsinnig, denn das ASP.NET Framework hat solche Standardszenarien abgebildet. Mehr zum Thema Validierung erfahren Sie in Kapitel 6, die dazu notwendigen Grundlagen über die Funktionsweise von Steuerelementen in ASP.NET erhalten Sie im folgenden Abschnitt und in Kapitel 5.

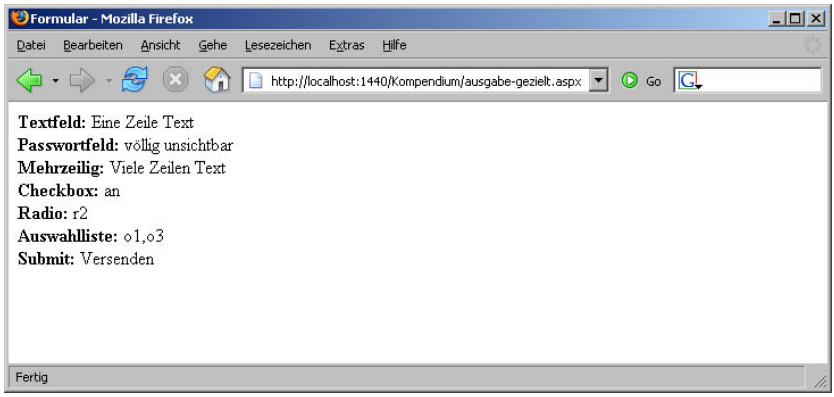

Abbildung 4.6: Diesmal werden nach dem Versand nur die Formulardaten angezeigt.

Wenn wir ein kleines Zwischenfazit ziehen möchten: Die Abfrage von Formulardaten mit ASP.NET geht relativ simpel, genau wie in den meisten anderen Webtechnologien von Haus aus auch, sei es PHP oder Cold Fusion oder was auch immer. Der Vorteil des alles umspannenden Frameworks ist hier aber zunächst vergeben. Nur auf HTTP-Daten zuzugreifen ist lediglich ein kleiner Teilaspekt einer modernen Webanwendung. Die Formularfelder selbst sollten (wie oben angesprochen) validiert werden, wenn das fehlschlägt, ist eine Vorausfüllung fällig. Viele weitere Anforderungen ergeben sich in alltäglichen Praxisprojekten. Der nächste Abschnitt stellt deswegen den Ansatz von ASP.NET vor, das Arbeiten mit HTML-Formularen etwas zu vereinfachen und auf eine solide und objektorientierte Basis zu stellen.

4.2 Grundlegendes zu HTML Controls

In diesem Buch werden Sie immer wieder sehen, dass Sie im Kopf der *.aspx*-Seite innerhalb der Funktion `Page_Load()` auf HTML-Elemente weiter unten zugreifen können, etwa nach folgendem Muster:

```
void Page_Load() {
  ausgabe.InnerText = "ASP.NET macht Spaß";
}
```

Dabei ist `ausgabe` gleichzeitig der Wert des `id`-Parameters eines (fast beliebigen) HTML-Elements weiter unten in der Seite. Durch `runat="server"` wird der ASP.NET-Interpreter angewiesen, dieses HTML-Element serverseitig zu verarbeiten:

```
<p id="ausgabe" runat="server"></p>
```

Nach Ausführung des obigen Codes wird an den Browser ein `<p>`-Element geschickt, das den angegebenen Text (`"ASP.NET macht Spaß"`) enthält. Diese Form der speziellen HTML-Elemente wird **HTML Controls** genannt und funktioniert im Wesentlichen für jedes HTML-Element. Einzige Voraussetzung: Sie benötigen eine ID und ein `runat="server"`.

Es liegt natürlich nahe, auch für Formularelemente HTML Controls zu verwenden; alleine die Aufgabe der Vorausfüllung der Felder scheint damit einfacher zu sein als noch im vorherigen Abschnitt.

Die HTML Controls für Formularelemente (und auch andere Elemente, wenngleich diese nur wenig Funktionalität bieten) sind alle im Namespace `System.Web.UI.HtmlControls` untergebracht. Wenn Sie die Dokumentation aus dem .NET Framework SDK verwenden, können Sie eine schnelle Übersicht über die einzelnen Klassen in diesem Namespace erhalten (siehe Abbildung 4.7).

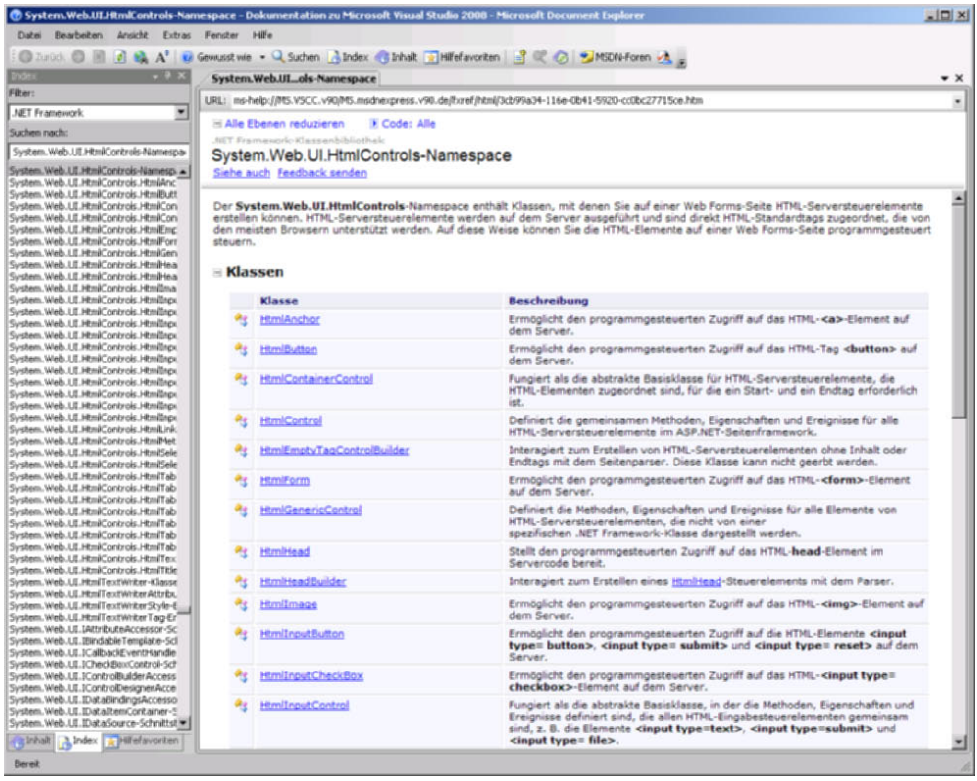

Abbildung 4.7: Die Klassen in `System.Web.UI.HtmlControls`

Die Basisklasse für alle HTML-Elemente (die ein `runat="server"` aufweisen) ist `HtmlGenericControl`. Ein Blick in den Klassenbrowser zeigt hier die Ereignisse und Eigenschaften auf, die jedes HTML-Element unter ASP.NET unterstützt. Die wichtigsten sind hierbei `InnerText` und `InnerHtml`. Damit können Sie den Inhalt eines HTML Controls sowohl auslesen als auch setzen.

Wie der Name schon andeutet, wird durch `InnerHtml` der HTML-Code innerhalb des Elements repräsentiert; `InnerText` steht für den Text (ohne HTML-Formatierungen), der in dem Element steht.

Betrachten wir folgendes Beispiel:

```
<p id="absatz" runat="server">
<b>&lt;%@ Page Language="C#" %&gt;</b>
</p>
```

Die Eigenschaft `absatz.InnerHtml` ist offensichtlich der HTML-Code innerhalb des <p>-Elements, also Folgendes:

```
<b>&lt;%@ Page Language="C#" %&gt;</b>
```

Welchen Wert hat aber nun `InnerText`? Nun, HTML-Tags, die nicht als »Klartext« im Browser angezeigt werden, werden natürlich ignoriert. Die spitzen Klammern, im HTML-Code noch HTML-codiert (`<` und `>`), werden durch das jeweilige Ausgabezeichen ersetzt, hier also < und >. Damit hat `InnerText` folgenden Wert:

```
<%@ Page Language="C#" %>
```

Besonders interessant wird der Unterschied jedoch erst beim Setzen der Eigenschaften. Wenn Sie eine lange Zeichenkette haben und sie nicht gesondert mit `HtmlEncode` vorbehandeln möchten, setzen Sie `InnerText`, und der ASP.NET-Prozess erledigt automatisch die Konvertierung für Sie. Wollen Sie stattdessen HTML-Formatierungen verwenden, wie beispielsweise in obigem Codeausschnitt Fettdruck (...), dann müssen Sie `InnerHtml` setzen und die entsprechenden HTML-Auszeichnungen verwenden.

Da es in diesem Kapitel aber um Formulare geht, werfen wir nun einen genaueren Blick auf die Formular-HTML-Controls. Eines der allgemeineren Controls ist `HtmlInputControl`, das die Oberklasse für alle mit dem <input>-Element erzeugten Formularfelder ist (also einzeilige Textfelder, Passwortfelder, Radiobuttons, Checkboxen, versteckte Formularfelder, File-Uploads, Versende-Grafiken und Versende-Schaltflächen). Die Dokumentation offenbart hier alle zur Verfügung stehenden Eigenschaften, wobei insbesondere die Eigenschaft `Value` interessant ist; sie enthält den Wert im entsprechenden Formularfeld.

i i i INFO

Später in diesem Kapitel werden wir ausführlich auf die einzelnen relevanten Formularfeldtypen und ihre Umsetzung in ASP.NET-HTML-Controls eingehen.

Die zur Verfügung stehenden Klassen innerhalb von `System.Web.UI.HtmlControls` können Sie Tabelle 4.1 entnehmen[1].

Klasse	Entsprechendes HTML-Element	Beschreibung
`HtmlAnchor`	<a>	Link
`HtmlButton`	<button>	Schaltfläche
`HtmlContainerControl`		HTML-Container (kann weitere HTML-Elemente enthalten)
`HtmlControl`	alle hier vorgestellten Elemente	Allgemeine Oberklasse
`HtmlForm`	<form>	Formular
`HtmlGenericControl`	diverse	Klasse für HTML-Elemente ohne eigene Klasse (z. B.)
`HtmlHead`	<head>	Kopfbereich der Seite

Tabelle 4.1: Die Klassen für HTML Controls

1 Zwei Klassen, die nur programmativ, aber nicht deklarativ, also mit ASP.NET-Markup verwendet werden können, wurden herausgelassen.

Klasse	Entsprechendes HTML-Element	Beschreibung
HtmlImage	<img>	Grafik
HtmlInputButton	<input type="button"> <input type="submit">	Formular-Schaltfläche
HtmlInputCheckbox	<input type="checkbox">	Checkbox
HtmlInputControl	<input>	Oberklasse für <input>
HtmlInputFile	<input type="file">	File-Upload
HtmlInputHidden	<input type="hidden">	Verstecktes Formularfeld
HtmlInputImage	<input type="image">	Versende-Grafik
HtmlInputRadioButton	<input type="radio">	Radiobutton
HtmlInputText	<input type="text"> <input type="password">	Einzeiliges Eingabefeld (auch Passwortfeld)
HtmlLink	<a>	Link
HtmlMeta	<meta>	Meta-Tag
HtmlSelect	<select>	Auswahlliste
HtmlTable	<table>	Tabelle
HtmlTableCell	<td>	Tabellenzelle
HtmlTableCellCollection	<td> (mehrfach)	Mehrere Tabellenzellen
HtmlTableRow	<tr>	Tabellenzeile
HtmlTableRowCollection	<tr> (mehrfach)	Mehrere Tabellenzeilen
HtmlTextArea	<textarea>	Mehrzeiliges Textfeld
HtmlTitle	<title>	Seitentitel

Tabelle 4.1: Die Klassen für HTML Controls (Fortsetzung)

Die Namen der einzelnen Controls sind für Sie insbesondere dann interessant, wenn Sie eine Eigenschaft nachschlagen möchten, beispielsweise in der Online-Referenz oder – natürlich noch besser und unserer Meinung nach übersichtlicher – im Referenzteil dieses Buchs. Bei der Programmierung selbst werden Ihnen diese Klassennamen nicht begegnen, Sie greifen auf Eigenschaften und Ereignisse zu, vermutlich immer auf dieselben.

4.3 Formularversand mit HTML Controls

Bevor wir nun direkt einsteigen noch ein wichtiger Hinweis: Alle Formular-HTML-Controls müssen innerhalb eines serverseitigen Formulars stehen, also innerhalb von <form runat="server">...</form>. Zusätzliche Parameter, beispielsweise den Namen des Formulars oder die Versandmethode, müssen Sie nicht angeben; das macht ASP.NET im Alleingang.

4.3.1 Formular serverseitig

Betrachten Sie folgende minimalistische ASP.NET-Seite:

```
<form runat="server" />
```

Wenn Sie eine Datei mit dieser einen Zeile erstellen, ihr die Endung `.aspx` geben und sie im Webbrowser aufrufen, erhalten Sie deutlich längeren Code zurück (optisch etwas aufgehübschte Wiedergabe):

```
<form name="ctl00" method="post" action="skript.aspx" id="ctl00">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUJMjk3MDAwMjUwZGS2glfCpW07aVv/z8uDaVipJ6HehA==" />
  </div>
</form>
```

Zunächst hat das Formular also einen Namen (`name`-Parameter) bekommen, `"_ctl00"`. Dann wurde die Versandmethode auf POST gesetzt (`method="post"`). Schließlich wurde noch das Ziel des Formulars gesetzt, und zwar auf das aktuelle Skript (`action="skript.aspx"`). Daran sehen Sie, dass wir das aus einer Zeile bestehende Testskript *skript.aspx* genannt haben.

Sie werden ebenfalls überrascht feststellen, dass das Formular ein verstecktes Feld namens `__VIEWSTATE` enthält (brav, von ASP.NET XHTML-konform in einem `<div>`-Element platziert); als Wert ist eine kryptische, 48 Zeichen lange Zeichenkette angegeben. ASP.NET benötigt diese Zeichenkette, um auf dieser Basis auf eingegebene Formulardaten ohne Cookies zugreifen und Formularfelder vorausfüllen zu können. Anhand der Zeichenkette weiß der ASP.NET-Interpreter, wo die Formulareingaben auf dem Server temporär abgelegt worden sind. Die Auswirkung dieses `__VIEWSTATE`-Feldes sehen Sie noch an späterer Stelle in diesem Kapitel.

Was aber passiert nun, wenn Sie Eigenschaften wie Formularnamen, Versandmethode und Versandziel selbst setzen möchten? Auf ein Neues, diesmal wird folgender Einzeiler (aus drucktechnischen Gründen auf zwei Zeilen aufgeteilt) getestet:

```
<form name="Formular" method="get"
      action="skript2.aspx" runat="server" />
```

Das Ergebnis sehen Sie hier:

```
<form name="form1" method="get" action="skript.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUJMjk3MDAwMjUwZGS2glfCpW07aVv/z8uDaVipJ6HehA==" />
  </div>
</form>
```

Sie sehen also – ASP.NET besitzt einen eigenen Willen. Lediglich die Versandmethode GET wurde beibehalten, der Rest wurde eliminiert. Jetzt ist Ihnen vermutlich schon klar, wieso wir im vorherigen Kapitel den herkömmlichen Zugriff auf Formulare so ausführlich erklärt haben. Zwar nimmt Ihnen ASP.NET eine Menge Arbeit ab, Sie verlieren damit aber auch einen Teil Ihrer Flexibilität als Programmierer.

! ! ! ACHTUNG

*Es gibt noch eine weitere Einschränkung, die gerne vergessen oder verdrängt wird: Sie können pro ASP.NET nur **ein** serverseitiges Formular verwenden. Mit einem »serverseitigen Formular« meinen wir ein Formular mit* `runat="server"`.

4.3.2 Versand ermitteln

Wie zuvor gesehen, ist das Ziel des Formularversands immer die aktuelle Seite. Es spielt sich also alles, Formularausgabe und -verarbeitung, auf einer *.aspx*-Seite ab. So ähnlich haben wir das übrigens auch im vorherigen Kapitel und vermutlich auch Sie in Ihren bisherigen ASP-Projekten gehandhabt.

Es ist also notwendig festzustellen, ob die ASP.NET-Seite »frisch« aufgerufen oder gerade Formulardaten verschickt werden. Wir wollen an dieser Stelle drei Ansätze untersuchen, um festzustellen, ob ein Formular verschickt worden ist. Falls ja, wird eine entsprechende Meldung ausgegeben, andernfalls wird das Formular angezeigt. Nicht jeder der Ansätze führt übrigens zum Erfolg.

Das Formular selbst besteht nur aus einer Versende-Schaltfläche, um das Ganze einfach und übersichtlich zu halten:

```
<form runat="server">
  <input type="submit" value="Versenden"
         runat="server" />
</form>
```

! ! ! ACHTUNG

Eine Warnung, die wir gar nicht oft genug anbringen können: Vergessen Sie auf keinen Fall das `runat="server"` *bei allen Formularelementen und natürlich auch bei dem Formular selbst. HTML Input Controls für Formulare werden nur innerhalb von serverseitigen Formularen unterstützt. Wenn Sie ein solches Element außerhalb eines serverseitigen Formulars einsetzen oder umgekehrt (also ein »normales« Formularelement innerhalb eines serverseitigen Formulars), erhalten Sie keine Fehlermeldung. Bei unerklärlichen Fehlern sollten Sie an dieser Stelle zuerst suchen!*

Schlag ins Wasser: Überprüfung der Versende-Schaltfläche

Der naheliegendste, aber nicht wirklich funktionierende Weg (dazu später mehr) besteht wie im vorherigen Kapitel auch darin, der Schaltfläche einen Namen zu geben und diesen dann abzufragen. Der erste Unterschied zu *ASP Classic* (ASP 1.0-3.0) ist zunächst der, dass Sie hierfür nicht mehr das `name`-Attribut verwenden dürfen, sondern das `id`-Attribut nehmen. Damit folgt Microsoft einer Empfehlung des W3C, den Standardisierungshütern des Internets. Die Schaltfläche sieht also folgendermaßen aus:

```
<input id="Submit" type="submit" value="Versenden"
       runat="server" />
```

In der Funktion `Page_Load`, die beim Laden der ASP.NET-Seite immer ausgeführt wird, können Sie dann über die ID (`"Submit"`) den Wert (`Value`) abfragen:

```
if (Submit.Value == "Versenden") {
  // Formular wurde verschickt
} else {
  // Formular wurde nicht verschickt (also ausgeben)
}
```

Hier ein Listing, das dies in die Tat umsetzen will. Wir definieren zusätzlich noch ein `<p>`-Element für die Ausgabe des Ergebnisses nach dem Versand:

```
<p id="ausgabe" runat="server" />
```

! ! ! ACHTUNG

Sie müssen alle serverseitigen Tags (also Tags mit `runat="server"`*) wieder schließen, sonst beschwert sich die ASP.NET-Engine. Wenn wir beispielsweise den Schrägstrich am Ende des obigen* `<p>`*-Elements nicht setzen, würde die Fehlermeldung aus angezeigt werden.*

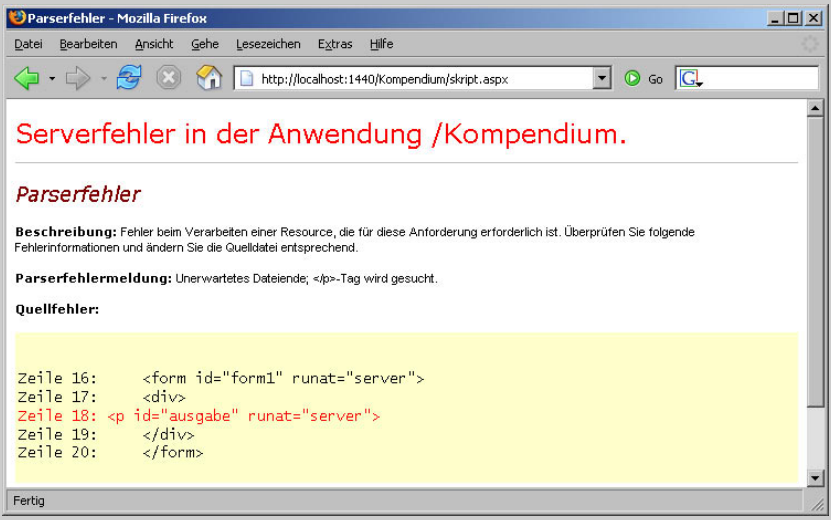

Abbildung 4.8: Fehlermeldung bei fehlendem Tag-Ende

Hier das komplette Listing:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Page_Load() {
    if (Submit.Value == "Versenden") {
      ausgabe.InnerText = "Vielen Dank für Ihre Angaben";
    }
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" runat="server" />
  </form>
</body>
</html>
```

Listing 4.10: Der erste Ansatz – aber er funktioniert nicht (*htmlausgabe1.aspx*).

Im Browser sehen Sie jedoch schon beim ersten Aufrufen der Seite eine Dankesmeldung (siehe Abbildung 4.9).

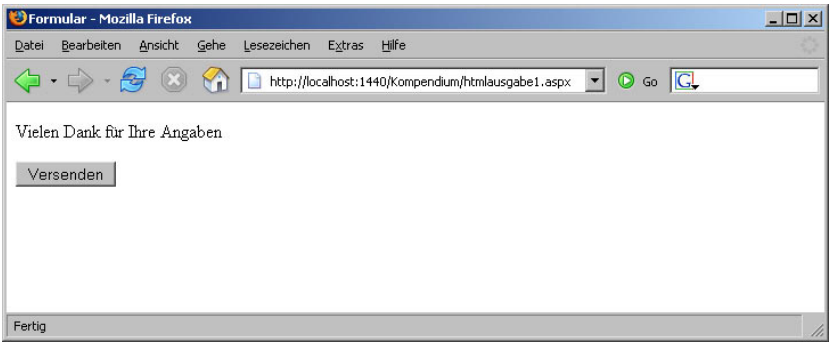

Abbildung 4.9: Die Ausgabe beim ersten Aufruf

Eine Erklärung ist schnell gefunden. Sie greifen auf `Submit.Value` zu, also den Wert im Formularelement mit `id="Submit"`. Die Funktion `Page_Load` wird allerdings erst beim Laden der Seite ausgeführt, bevor sie an den Browser geschickt wird. An dieser Stelle des Ablaufs sind alle Formularelemente schon erstellt worden, es gibt bereits die Versende-Schaltfläche mit `id="Submit"`. Dementsprechend hat `Submit.Value` schon beim ersten Laden den Wert `"Versenden"`. Diese Lösung kann also nicht funktionieren, und wir müssen nach Alternativen suchen.

Schon besser: Die Eigenschaft IsPostBack

Die `Page`-Klasse beschreibt die aktuelle ASP.NET-Seite. Wenn Sie beispielsweise im Kopf einer Seite die Angabe der verwendeten Skriptsprache näher betrachten, finden Sie diese Klasse dort wieder:

```
<%@ Page Language="C#" %>
```

i i i INFO

Die Klasse `Page` *befindet sich in der Assembly* `System.Web.UI`.

Die `Page`-Klasse hat die Eigenschaft `IsPostBack`. Diese ist genau dann `True` (bzw. `true`, bei C#), wenn gerade ein serverseitiges Formular auf die aktuelle Seite verschickt wurde. Diesen Vorgang nennt man *PostBack*. Damit ist die Überprüfung, ob ein Formular gerade verschickt worden ist, sehr einfach:

```
if (Page.IsPostBack) {
  // Formular wurde verschickt
} else {
  // Formular wurde nicht verschickt (also ausgeben)
}
```

Hier ein komplettes Beispiel:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Page_Load() {
    if (Page.IsPostBack) {
      ausgabe.InnerText = "Vielen Dank für Ihre Angaben";
    }
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" runat="server" />
  </form>
</body>
</html>
```

Listing 4.11: Überprüfung mit `Page.IsPostBack` (*htmlausgabe2.aspx*)

Alternativ: Serverseitige Funktion

Wieder einmal werfen wir einen Blick in einen Klassenbrowser (oder in den Anhang B), und zwar insbesondere auf die Klasse `System.Web.UI.HtmlControls.HtmlInputButton`. Diese ist wie zuvor erläutert für Schaltflächen zuständig, insbesondere also auch für Versende-Schaltflächen. Im Class Browser ist das Event (Ereignis) `ServerClick` zu sehen. Es wird aktiviert, wenn die Schaltfläche angeklickt wird.

Events funktionieren ähnlich wie ihre JavaScript-Pendants. Vor den Namen des Events wird ein `"On"` gesetzt. Damit erhalten Sie den Namen des Parameters, über den Sie den sogenannten *Event-Handler*, also die Behandlungsfunktion beim Eintreten des Ereignisses, angeben können.

In JavaScript kann das beispielsweise folgendermaßen aussehen:

```
<html>
<body onload="alert('Das ist clientseitig ...');">
</body>
</html>
```

Das Ereignis heißt hier `load` und tritt beim Laden der HTML-Seite im Browser ein. Sie können also durch `onload` angeben, was passieren soll, wenn das `load`-Ereignis eintritt. In diesem Fall wird ein modales Dialogfenster ausgegeben (siehe Abbildung 4.10).

Serverseitig funktioniert das ähnlich; folgendermaßen können Sie beim Eintreten des Ereignisses `ServerClick` für die Schaltfläche eine Behandlungsroutine angeben:

Abbildung 4.10: Ein (clientseitig) erzeugtes Info-Fenster

```
<input type="submit" value="Versenden"
       OnServerClick="Versand" runat="server" />
```

Wenn der Benutzer auf die Schaltfläche klickt, wird die serverseitige Funktion `Versand()` aufgerufen. Beachten Sie, dass Sie dies als Programmierer einstellen, sich aber nicht mit der tatsächlichen HTML-Ausgabe beschäftigen müssen. Im Browser selbst wird eine normale Schaltfläche dargestellt und das Formular per Mausklick an den Webserver geschickt. Der Endnutzer bekommt die Funktion `Versand()` gar nicht erst zu Gesicht, er weiß nicht einmal etwas von ihrer Existenz!

Bei der Erstellung der Funktion `Versand()`müssen Sie beachten, dass diese Funktion zwei Parameter enthält. Als erster Parameter wird eine Referenz auf das Objekt übergeben, das den Funktionsaufruf angestoßen hat; in diesem Fall also die Schaltfläche. Der zweite Parameter sind zusätzliche Argumente, die an das Ereignis übermittelt wurden (beispielsweise bei anderen Elementen die Koordinaten des auslösenden Mausklicks).

i i i INFO

In der Regel benötigen Sie diese zwei Parameter nicht. Sie müssen sie zwar im Funktionskopf angeben, aber Sie werden sie innerhalb der Funktion nicht verwenden. Wir werden später noch eine Einsatzmöglichkeit aufzeigen.

*** * * TIPP**

Im Übrigen heißt der Parameter genau deswegen `onserverclick` *und nicht* `OnClick`*, weil* `onclick` *ein HTML-Event-Handler und für die Ausführung von clientseitigem Skriptcode zuständig ist. Durch* `onserverclick` *wird eine Namenskollision vermieden.*

Der Funktionskopf sieht unter C# folgendermaßen aus:

```
void Versand(Object o, EventArgs e) {
  // ...
}
```

Nachfolgend nun das komplette Beispiel:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerText = "Vielen Dank für Ihre Angaben";
```

```
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.12: Überprüfung mit einer serverseitigen Funktion (*htmlausgabe3.aspx*)

Fazit

Von den drei vorgestellten Methoden funktionieren nur zwei. In der Regel bevorzugen wir die letzte Methode. Der Hauptgrund liegt darin, dass der Code zur Formularverarbeitung in eine gesonderte Funktion ausgelagert werden kann, was den Code übersichtlicher macht. Der zweite Grund hat nichts mit Performance oder Effektivität zu tun, sondern hängt mit den technischen Gegebenheiten beim Buchdruck zusammen. Wenn wir `Page_Load` verwenden, benötigen wir eine `If`-Abfrage (nämlich die von `Page.IsPostBack`), die den folgenden Code um zwei zusätzliche Leerzeichen einrückt. Damit rückt jedoch auch gleichzeitig der rechte Seitenrand näher, wir versuchen allerdings, Umbrüche möglichst zu vermeiden. Aber für Sie wird vermutlich nur der Hauptgrund zutreffen. Ansonsten sind beide Möglichkeiten als gleichwertig anzusehen.

4.3.3 Das Formular ausblenden

Wenn die Formulardaten verschickt (und verarbeitet) worden sind, wollen Sie womöglich das Formular nicht mehr anzeigen. Auch hier gibt es wieder mehrere Möglichkeiten.

Zunächst einmal könnten Sie nach dem Formularversand den Benutzer auf eine andere Seite umleiten, die eine Dankesmeldung oder Ähnliches enthält:

```
void Versand(Object o, EventArgs e) {
  Response.Redirect("danke.aspx");
}
```

Durch den Aufruf von `Response.Redirect ()` wird der Browser des Benutzers auf die als Parameter übergebene URL weitergeleitet, in diesem Fall `danke.aspx`.

Es gibt jedoch eine bequemere Möglichkeit, bei der Sie die aktuelle Seite nicht verlassen müssen. Zuerst müssen Sie dem Formular selbst auch eine ID vergeben (falls nicht eh schon automatisch von Visual Web Developer/Visual Studio gemacht), um es ansprechen zu können:

```
<form id="form1" runat="server">
...
</form>
```

Die zugehörige Klasse in `System.Web.UI.HtmlControls` ist `HtmlForm`. Dort gibt es die boolesche Eigenschaft `Visible`, die angibt, ob das Formular sichtbar ist oder nicht. Wenn Sie `Visible` auf `False` (bzw. bei C#: `false`, wobei auch VB die Kleinschreibung akzeptieren würde) setzen, wird das Formular nicht angezeigt. Hier ein komplettes Listing:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerText = "Vielen Dank für Ihre Angaben";
    form1.Visible = false;
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.13: Nach dem Versand wird das Formular unsichtbar gemacht (*htmlausgabe4.aspx*).

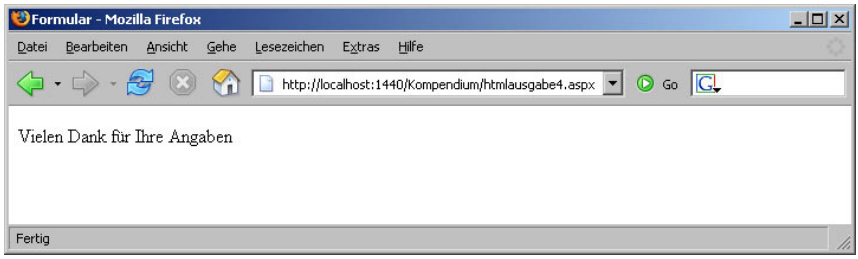

Abbildung 4.11: Das Formular wird nicht (mehr) angezeigt.

Wenn Sie den HTML-Code nach dem Versand betrachten, sieht er ungefähr folgendermaßen aus:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
  Formular
</title></head>
<body>
```

```
    <p id="ausgabe">Vielen Dank f&#252;r Ihre Angaben</p>
  </body>
  </html>
```

Das Formular ist also nicht nur unsichtbar, es ist verschwunden. Der ASP.NET-Interpreter schickt per `Visible = False` unsichtbar gemachte Elemente erst gar nicht an den Browser.

*** * * TIPP**

Sie können das Formular auch unsichtbar machen, indem Sie es in einen `<div>`*-Container legen:*

```
<div id="Container" runat="server">...</div>
```

Die `visibility`*-Stileigenschaft des Formulars setzen Sie dann serverseitig auf* `"hidden"`*:*

```
Container.Style["visibility"] = "hidden";
```

Das Formular wird nun zwar an den Browser geschickt; wenn dieser aber Stylesheets unterstützt, sieht es der Benutzer nicht.

4.4 HTML Controls im Einsatz

Wie schon im vorhergehenden Kapitel, werden wir auch an dieser Stelle die Formularfeldtypen einzeln untersuchen und dabei jeweils aufzeigen, wie Sie auf die dort angegebenen Daten zugreifen können. Die Vorgehensweise ist immer ähnlich, der Teufel steckt aber sprichwörtlich im Detail, was nach einer genauen Darstellung verlangt.

4.4.1 Textfeld

Ein einzeiliges Textfeld können Sie mit dem `<input>`-Element darstellen; aber vergessen Sie hier (und bei den anderen Formularfeldtypen) nicht, `runat="server"` anzugeben!

```
<input type="text" id="Feldname" runat="server" />
```

Der Text in dem Textfeld steht in der Eigenschaft `Value`. Sie können sich das recht einfach merken, wenn Sie daran denken, dass Sie mit dem HTML-Parameter `value` ein Textfeld mit einem Wert vorbelegen können.

Folgendes Beispiellisting stellt ein Textfeld dar und gibt nach dem Versand den eingegebenen Text aus:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerText = "Ihre Eingabe: " + Feldname.Value;
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
```

```
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input type="text" id="Feldname" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.14: Der Wert aus dem Textfeld wird ausgegeben (*htmltextfeld.aspx*).

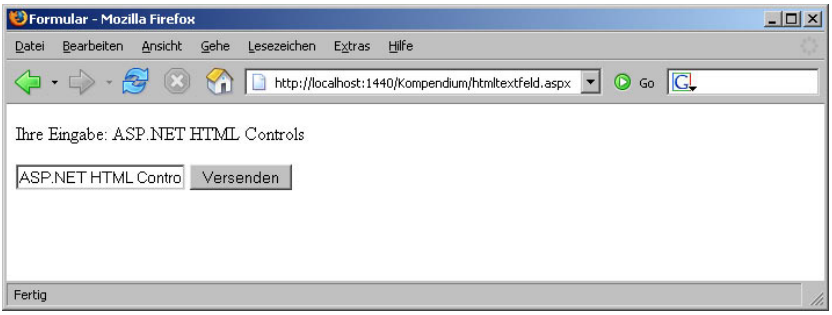

Abbildung 4.12: Der eingegebene Wert wird wieder ausgegeben.

Abbildung 4.12 können Sie entnehmen, dass der ASP.NET-Interpreter die Eingabe im Textfeld beibehält. Das Geheimnis liegt hier in dem versteckten Formularfeld, das Sie auch in der Ausgabe des Skripts *htmltextfeld.aspx* wieder finden:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/
wEPDwULLTExODM1OTUxNzUPZBYCAgQPFgIeCWlubmVyaHRtbAUjSWhyZSBFaW5nYWJlOiBBU1AuTkVUIEhUTUwg
Q29udHJvbHNkZH/mnFISQV9N8x4yttI/4PgSldbi" />
```

Wie bereits erläutert, werden über dieses Feld Formulareingaben beibehalten. Es war also kein zusätzlicher Code mehr nötig, um das Formular vorauszufüllen.

i i i INFO

Sie können natürlich das Feld bei Bedarf mit einem anderen Wert füllen, indem Sie die Eigenschaft `Value` *setzen. In C# kann das dann so aussehen:*

```
Feldname.Value = "Vorausfüllung";
```

Diese Vorausfüllung lässt sich nicht deaktivieren, da die Daten bei jedem POST-Versand erneut an das Skript übergeben werden und der ASP.NET-Interpreter sie jedes Mal wieder in die entsprechenden Formularfelder einfügt. Wenn Sie das Formularfeld leeren möchten, müssen Sie es auf eine leere Zeichenkette setzen:

```
Feldname.Value = "";
```

Zwar schlagen einige Dokumentationen noch vor, den Parameter `EnableViewState` des entsprechenden Formularelements auf `"False"` zu setzen, das aber genügt nicht.

4.4.2 Passwortfeld

Passwortfelder werden – wie die meisten Formularfeldtypen – mit dem HTML-Element `<input>` dargestellt:

```
<input type="password" id="Feldname" runat="server" />
```

Auch hier steht in der Eigenschaft `Value` des Elements der angegebene Wert. Hier ein illustratives Beispiel:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerText = "Ihre Eingabe: " + Feldname.Value;
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input type="password" id="Feldname" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.15: Der Wert aus dem Passwortfeld wird ausgegeben (*html/passwortfeld.aspx*).

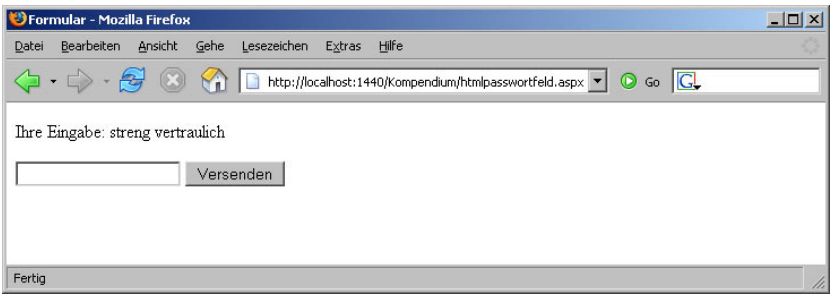

Abbildung 4.13: Der Wert aus dem Passwortfeld wird ausgegeben.

Sie sehen in Abbildung 4.13, dass das Passwortfeld nicht vorausgefüllt ist. Offensichtlich können Passwortfelder aus Sicherheitsgründen nicht vorbelegt werden. Wir haben das im vorherigen Abschnitt schon einmal kurz analysiert – das Passwort wäre dann im Klartext aus dem Cache

des Webbrowsers einsehbar, was ein potenzielles Sicherheitsrisiko darstellen würde (obwohl normalerweise nur das eigene Windows-Konto Zugriff auf den Browsercache hat). Schlimmer noch: Das Passwort würde dann eventuell erneut ungesichert im Internet übertragen, und das ist ein tatsächliches Problem. Also: Keine Vorausfüllung bei Passwortfeldern.

4.4.3 Mehrzeiliges Textfeld

Das mehrzeilige Textfeld wird über `<textarea>` realisiert, also nicht mit dem `<input>`-Element.

```
<textarea id="Feldname" runat="server"></textarea>
```

Auch wenn das `<textarea>`-Element keinen HTML-Parameter `value` kennt (der Inhalt des Formularfelds steht hier zwischen `<textarea>` und `</textarea>`), können Sie vonseiten ASP.NET über die Eigenschaft `Value` auf den Wert im mehrzeiligen Textfeld zugreifen. Folgendes Listing illustriert dieses Vorgehen:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerText = "Ihre Eingabe: " + Feldname.Value;
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <textarea id="Feldname" runat="server"></textarea>
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.16: Der Feldwert wird ausgegeben (*htmlmehrzeilig.aspx*).

4.4.4 Checkbox

Wie die meisten anderen Formularfelder auch, wird eine Checkbox mit dem `<input>`-Element dargestellt – der `type`-Parameter macht den Unterschied:

```
<input type="checkbox" id="Feldname" value="an"
       runat="server" />
```

Zwar gibt es auch bei Checkboxen von ASP.NET-Seite her eine Eigenschaft `Value`, sie nimmt aber immer den Wert des `value`-Parameters der Checkbox an, egal ob sie aktiviert ist oder nicht. Was Sie jedoch interessiert, ist der Zustand der Checkbox. Hier ist die Eigenschaft `Checked` geeignet. Bei `true` ist sie aktiviert, bei `false` dagegen nicht.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerText = "Angekreuzt: " + Feldname.Checked;
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input type="checkbox" id="Feldname" value="an"
           runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.17: Es wird ausgegeben, ob die Checkbox angekreuzt (aktiviert) wurde (*htmlcheckbox.aspx*).

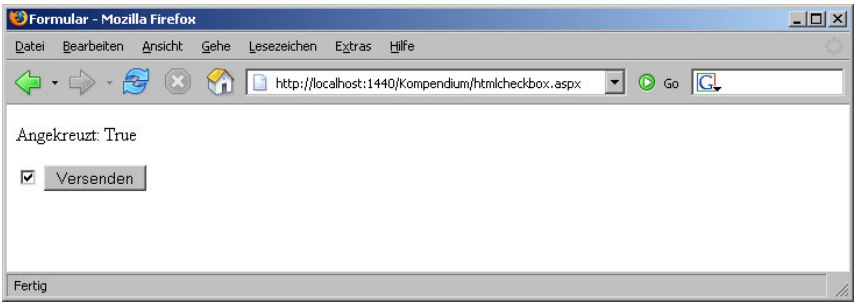

Abbildung 4.14: Der Zustand der Checkbox wird ausgegeben.

4.4.5 Radiobutton

Radiobuttons treten ja immer in Rudeln auf, wobei alle zusammengehörigen Radiobuttons denselben Namen (`name`-Parameter) tragen (aber unterschiedliche Werte, d. h. `value`-Parameter haben). Bei HTML Controls benötigen Sie jedoch den `id`-Parameter. Das Vorgehen ist nun folgendes:

- Setzen Sie den `name`-Parameter wie gewohnt, d.h., alle Radiobuttons aus einer Gruppe haben denselben Wert.
- Anstelle des `value`-Parameters verwenden Sie jedoch den `id`-Parameter.

Das sieht dann beispielsweise folgendermaßen aus:

```
1<input type="radio" name="Feldname" id="Button1"
        runat="server" />
2<input type="radio" name="Feldname" id="Button2"
        runat="server" />
3<input type="radio" name="Feldname" id="Button3"
        runat="server" />
```

! ! ! ACHTUNG

Achten Sie darauf, im `id`*-Parameter keine Sonderzeichen und auch keine Leerzeichen zu verwenden. Unter HTML und ASP war das kein Problem, bei ASP.NET ist es jedoch eines.*

Folgender Code (vorausgesetzt natürlich, er befindet sich innerhalb von `<form runat="server">` ...`</form>`) wird vom ASP.NET-Interpreter in folgenden Code umgesetzt:

```
1<input value="Button1" name="Feldname" id="Button1" type="radio" />
2<input value="Button2" name="Feldname" id="Button2" type="radio" />
3<input value="Button3" name="Feldname" id="Button3" type="radio" />
```

Sie sehen also – der `value`-Parameter wird automatisch eingesetzt, Sie brauchen sich darum nicht zu kümmern.

Um nun den Zustand der einzelnen Radiobuttons abzufragen, können Sie auf die Eigenschaft `Checked` zugreifen. Wie auch bei Checkboxen können Sie damit erkennen, ob ein Radiobutton aktiviert ist oder nicht.

*** * * TIPP**

Wenn Sie jedoch schnell feststellen möchten, welcher Radiobutton aus einer Gruppe aktiviert wurde, können Sie auf `Request.Form["name-Attribut"]` *zugreifen.*

Nachfolgendes Listing verwendet sowohl `Request.Form` als auch die Eigenschaft `Checked` der HTML Controls:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerHtml = "<ul><li>Button1: " +
                        Button1.Checked + "</li>" +
                        "<li>Button2: " +
                        Button2.Checked + "</li>" +
                        "<li>Button3: " +
                        Button3.Checked + "</li>" +
                        "</ul>Aktiviert: " +
                        Request.Form["Feldname"];
  }
</script>
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    1<input type="radio" name="Feldname" id="Button1" runat="server" />
    2<input type="radio" name="Feldname" id="Button2" runat="server" />
    3<input type="radio" name="Feldname" id="Button3" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.18: Die einzelnen Radiobuttons werden untersucht (*htmlradio.aspx*).

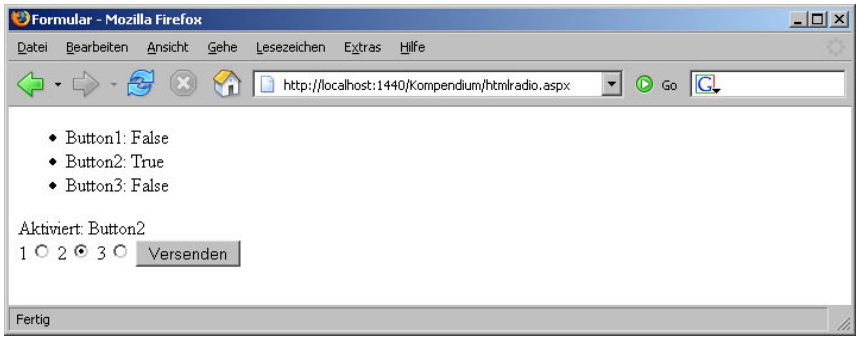

Abbildung 4.15: Die Zustände der einzelnen Radiobuttons

4.4.6 Auswahlliste

Das letzte Formularelement ist wieder das komplizierteste: die Auswahllisten. Das Problem liegt hier wie zuvor nicht bei den einfachen Auswahllisten, sondern bei den mehrfachen Auswahllisten (also denen mit `multiple`-Attribut im `<select>`-Tag). Aber der Reihe nach.

Beginnen wir mit den einfachen Auswahllisten:

```
<select name="Auswahlliste" size="3" runat="server">
  <option value="o1">Option 1</option>
  <option value="o2">Option 2</option>
  <option value="o3">Option 3</option>
</select>
```

Ein Blick in die Referenz beschert für Auswahllisten (Klasse `HtmlSelect`) eine ganze Reihe von Eigenschaften. Für den Zugriff auf das gewählte Element gibt es zwei Möglichkeiten:

- Sie greifen auf die Eigenschaft `Value` zu, die den Wert des `value`-Parameters des gewählten Listenelements enthält:

```
Feldname.Value
```

- Oder Sie verwenden die Eigenschaft `SelectedIndex`, die den numerischen Index des gewählten Listenelements ausgibt (Achtung: Zählung beginnt bei 0). Diesen Index verwenden Sie, um über die Kollektion `Items` auf das entsprechende Element zuzugreifen. Bei diesem Element erhalten Sie dann über `Value` seinen Wert:

```
Feldname.Items[Feldname.SelectedIndex].Value
```

Die letztere Methode ist natürlich viel umständlicher. Außerdem gibt es Probleme, wenn gar kein Element ausgewählt wurde. Dann hat `SelectedIndex` den Wert –1, und der Zugriff auf `Feldname.Items[-1]`schlägt natürlich fehl. Sie müssen also eine zusätzliche Abfrage einführen.

Ein anderer Knackpunkt ist die Mehrfach-Auswahlliste:

```
<select id="Feldname" multiple="multiple" size="3" runat="server">
  <option value="o1">Option 1</option>
  <option value="o2">Option 2</option>
  <option value="o3">Option 3</option>
</select>
```

Zunächst ein großes Ärgernis an ASP.NET an sich. Wenn Sie die Liste wie oben gezeigt im Browser aufrufen möchten, erhalten Sie die in Abbildung 4.16 gezeigte Fehlermeldung.

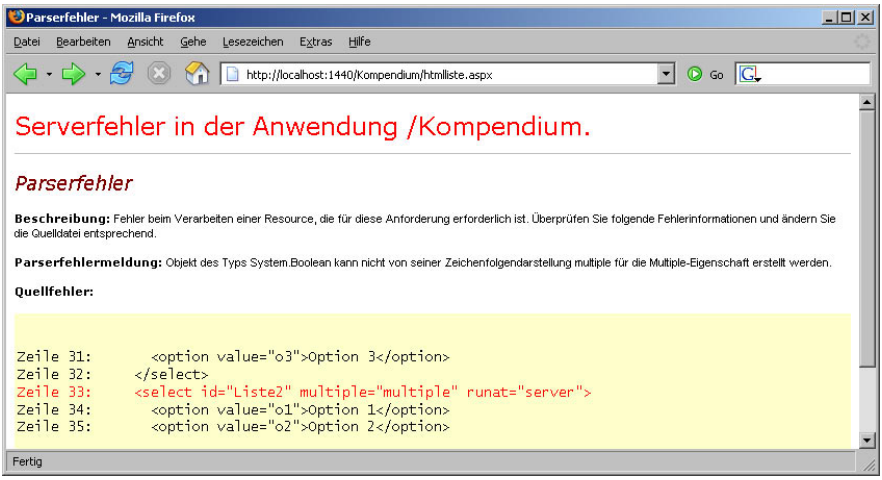

Abbildung 4.16: ASP.NET hat ein großes Problem …

Der Grund: ASP.NET setzt die Auswahlliste intern in ein Objekt um, das unter anderem die Eigenschaft `Multiple` besitzt. Diese Eigenschaft ist ein boolescher Wert, also erwartet ASP.NET eine entsprechende Angabe. Das Folgende funktioniert – auch wenn es Visual Web Developer als fehlerhaft unterkringelt:

```
<select id="Feldname" multiple="true" size="3" runat="server">
  <option value="o1">Option 1</option>
  <option value="o2">Option 2</option>
  <option value="o3">Option 3</option>
</select>
```

Doch zurück zur Programmierung an sich: Bei Mehrfachlisten enthält die Eigenschaft `SelectedIndex` lediglich die Position des ersten gewählten Listenelements; auf alle weiteren können Sie so nicht zugreifen. Hier können Sie sich behelfen, indem Sie per Schleife alle Elemente durchlaufen und dann überprüfen, ob das jeweilige Element ausgewählt wurde oder nicht.

Dazu brauchen Sie noch die folgenden Informationen:

- Die Anzahl der Listenelemente erhalten Sie über die Eigenschaft `Count` der Elemente der Auswahlliste (in unserem Beispiel: `Feldname.Items.Count`).
- Ob ein Element ausgewählt wurde oder nicht, sehen Sie anhand der booleschen Eigenschaft `Selected`:

```
Feldname.Items[0].Selected
```

Die folgende Schleife durchläuft also die gesamte Liste und gibt aus, welche Elemente ausgewählt wurden:

```
for (int i = 0; i < Feldname.Items.Count; i++) {
  if (Feldname.Items[i].Selected) {
    Response.Write(Feldname.Items[i].Value + "<br />");
  }
}
```

i i i INFO

Alternativ können Sie natürlich auch auf eine `For-Each`*-Schleife setzen.*

Hier ein Listing, das sowohl Einfach- als auch Mehrfach-Auswahllisten enthält:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerHtml = "Liste1: " + Liste1.Value;
    ausgabe.InnerHtml += " (Element " +
                         Liste1.SelectedIndex + ")<br />";
    ausgabe.InnerHtml += "Liste2: ";
    for (int i = 0; i < Liste2.Items.Count; i++) {
      if (Liste2.Items[i].Selected) {
        ausgabe.InnerHtml +=
          HttpUtility.HtmlEncode(Liste2.Items[i].Value) + " ";
      }
    }
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
```

```
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <select id="Liste1" size="3" runat="server">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <select id="Liste2" multiple="true" runat="server">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.19: Zwei verschiedene Auswahllisten (*htmlliste.aspx*)

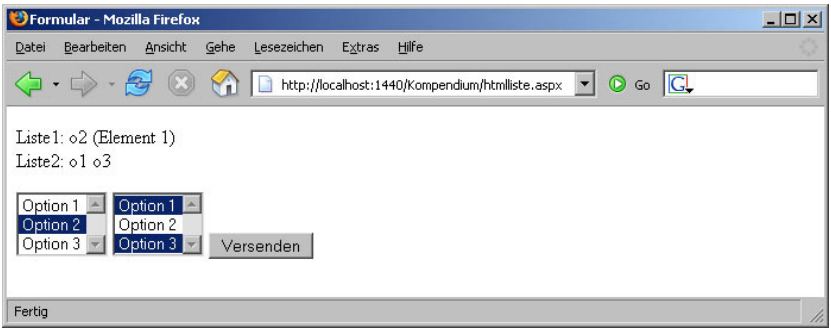

Abbildung 4.17: Die beiden Auswahllisten werden untersucht.

4.4.7 Komplettes Beispiel

Am Ende dieses Abschnitts geben wir Ihnen noch ein komplettes Beispiel, in dem Sie noch einmal alle Formularfeldtypen wiederfinden und bei dem alle Werte ausgegeben werden. Außerdem wird – wie zuvor schon einmal gezeigt – nach der Eingabe der Daten das Formular ausgeblendet, indem seine `Visible`-Eigenschaft auf `False` gesetzt wird. Da es vom theoretischen Aspekt her keine Neuerungen gibt, geht es gleich mit dem Code los.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    ausgabe.InnerHtml = "Textfeld: " + Textfeld.Value;
    ausgabe.InnerHtml += "<br />Passwortfeld: " +
                         Passwortfeld.Value + "<br />";
```

```
      ausgabe.InnerHtml += "Mehrzeiliges Textfeld: " +
                           Mehrzeilig.Value + "<br />";
      ausgabe.InnerHtml += "Checkbox: " + Checkbox.Checked;
      ausgabe.InnerHtml += "<br />Radiobutton: " +
                           Request.Form["Radio"];
      ausgabe.InnerHtml += "<br />Auswahlliste: ";
      for (int i= 0; i < Auswahlliste.Items.Count; i++) {
        if (Auswahlliste.Items[i].Selected) {
          ausgabe.InnerHtml += Auswahlliste.Items[i].Value + " ";
        }
      }
      form1.Visible = false;
    }
  </script>

  <html xmlns="http://www.w3.org/1999/xhtml">
  <head runat="server">
    <title>Formular</title>
  </head>
  <body>
    <p id="ausgabe" runat="server" />
    <form id="form1" runat="server">
      Textfeld:
      <input type="text" id="Textfeld" runat="server" />
      <br />
      Passwortfeld:
      <input type="password" id="Passwortfeld" runat="server" />
      <br />
      Mehrzeiliges Textfeld
      <textarea id="Mehrzeilig" wrap="virtual" runat="server" />
      <br />
      Checkbox
      <input type="checkbox" id="Checkbox" value="an" runat="server" />
      <br />
      Radiobutton
      <input type="radio" name="Radio" id="r1" runat="server" />1
      <input type="radio" name="Radio" id="r2" runat="server" />2
      <br />
      Auswahlliste
      <select id="Auswahlliste" size="3" multiple runat="server">
        <option value="o1">Option 1</option>
        <option value="o2">Option 2</option>
        <option value="o3">Option 3</option>
      </select>
      <br />
      <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
  runat="server" />
    </form>
  </body>
  </html>
```

Listing 4.20: Das komplette Beispiel (*html/ausgabe.aspx*)

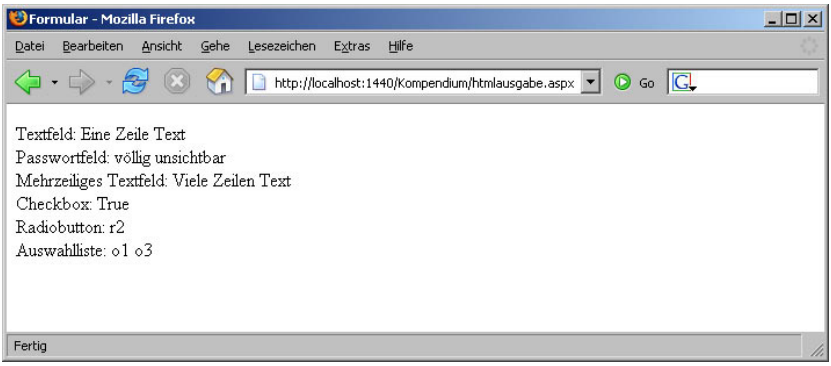

Abbildung 4.18: Die Formulardaten werden ausgegeben und das Formular verschwindet.

4.5 Spezialfall File-Upload

Ein Formularfeld haben wir bis jetzt immer außen vor gelassen – die Rede ist von `<input type="file">`. Damit können Dateien an den Webserver übertragen werden. Bei ASP war der Zugriff auf diese Daten entweder nicht zuverlässig oder musste über Third-Party-Komponenten realisiert werden. Diese Zeiten sind seit dem Erscheinen von ASP.NET 1.0 passé, denn dort ist die Unterstützung für Datei-Uploads integriert.

Zunächst müssen Sie das Formular auf den File-Upload vorbereiten. Dazu müssen Sie das Attribut `enctype` (gibt den Kodierungstyp für die Daten an) auf `"multipart/form-data"` setzen:

```
<form enctype="multipart/form-data" runat="server">
```

Damit werden die Formulardaten nicht wie gewöhnlich als Name-Wert-Paare verschickt (siehe vorheriges Kapitel), sondern als einzelne, MIME-codierte Teile.

Erstellen Sie anschließend ein entsprechendes Formularelement, und vergessen Sie nicht die ID und – ganz wichtig – das `runat="server"`:

```
<input type="file" id="datei" runat="server" />
```

Nach dem Formularversand erhalten Sie über `datei.PostedFile` eine Referenz auf die übertragene Datei. Um genau zu sein, erhalten Sie ein Objekt des Typs `HttpPostedFile`, das unterhalb von `System.Web` angesiedelt ist. Dieses Objekt hat die Methode `SaveAs`, mit der Sie die Datei an einer zu spezifizierenden Stelle ablegen können – ASP.NET benötigt natürlich die entsprechenden Schreibrechte dafür!

```
datei.PostedFile.SaveAs("c:\temp\datei.xxx");
```

Die Klasse `HttpPostedFile` hat zudem noch einige Eigenschaften:

Eigenschaft	Beschreibung
`ContentLength`	Dateigröße
`ContentType`	MIME-Typ der Datei
`FileName`	Ursprünglicher Dateiname
`InputStream`	`Stream`-Objekt für die übertragene Datei

Tabelle 4.2: Die Eigenschaften der Klasse `HttpPostedFile`

Exkurs

Im folgenden Beispiel übertragen wir eines der Listings aus diesem Kapitel an den Webserver und geben die entsprechenden Eigenschaften der Klasse `HttpPostedFile` aus:

```
<%@ Page Language="C#" %>
<script runat="server">
  void Versand(Object o, EventArgs e) {
    HttpPostedFile d = datei.PostedFile;
    ausgabe.InnerHtml = "Gr&ouml;&szlig;e: " +
                        d.ContentLength +
                        "<br />MIME-Typ: " +
                        d.ContentType +
                        "<br />Dateiname: " +
                        d.FileName;
  }
</script>
<html>
<head>
<title>File-Upload</title>
</head>
<body>
<p id="ausgabe" runat="server" />
<form enctype="multipart/form-data" runat="server">
  <input type="file" id="datei" runat="server" />
  <input type="submit" value="Versenden"
         OnServerClick="Versand" runat="server" />
</form>
</body>
</html>
```

Listing 4.21: Die Eigenschaften der übertragenen Datei werden ausgegeben (*htmlupload1.aspx*).

Je nach verwendetem Browser erfolgt eine unterschiedliche Ausgabe. Das ist insbesondere anhand der MIME-Typen ersichtlich. Der Internet Explorer beispielsweise verwendet bei *.aspx*-Dateien den MIME-Typ *text/html*, der für HTML-Dateien vorgesehen ist; Mozilla-Browser beispielsweise übermitteln *application/octet-stream*. Besonders problematisch ist es jedoch bei der Eigenschaft `FileName`. Gemäß offizieller Dokumentation enthält diese Eigenschaft den originalen Dateinamen der übertragenen Datei auf dem Client-System, inklusive Pfad. Das trifft allerdings nur beim Microsoft Internet Explorer zu, denn es ist natürlich Sache des Browsers, diese Daten zu übermitteln. Während also der Internet Explorer sehr geschwätzig ist und den kompletten Dateinamen samt Pfad übermittelt, beschränken sich andere Browser auf den bloßen Dateinamen (was auch sinnvoll ist). In Abbildung 4.19 sehen Sie die Ausgabe im Internet Explorer, Abbildung 4.20 zeigt das Skript im Firefox, der auf Mozilla basiert.

* * * TIPP

Sie sehen also: Überprüfen Sie Ihre Skripte unbedingt in unterschiedlichen Browsern!

Zum Abschluss dieses Abschnitts ein noch etwas komplexeres Beispiel. Es soll – beispielsweise als Bestandteil eines *Content Management Systems* (CMS) – eine Grafik auf den Server übertragen und dann direkt angezeigt werden. Dabei gehen wir folgendermaßen vor:

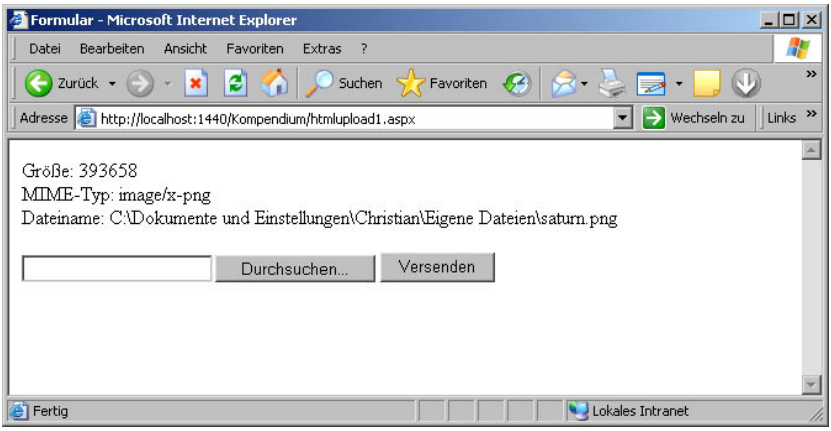

Abbildung 4.19: Der File-Upload im Internet Explorer

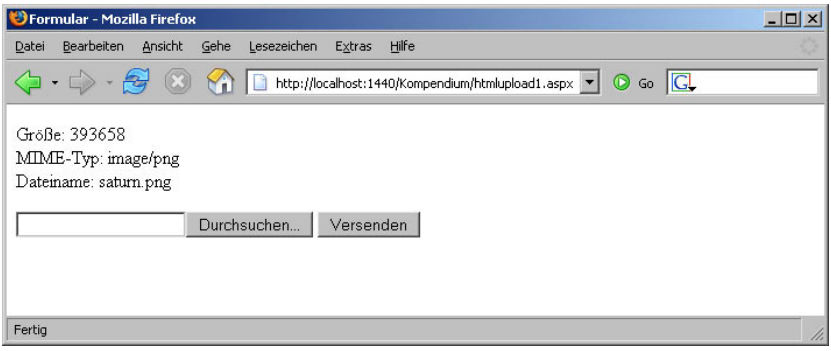

Abbildung 4.20: Der File-Upload im Firefox (lediglich der Dateiname, kein Pfad)

- Der originale Dateiname der übertragenen Datei wird ermittelt. Dazu wird der letzte Backslash im Dateinamen gesucht (falls vorhanden), und danach werden alle Zeichen im String verwendet.
- Die übertragene Datei wird in ein Verzeichnis *temp* unterhalb des aktuellen Verzeichnisses kopiert. Als Dateiname wird der zuvor ermittelte Name verwendet.
- Schließlich wird die Datei als Grafik in die HTML-Seite eingebunden. Der Dateiname sollte mittlerweile hinlänglich bekannt sein.

Beginnen wir mit der Ermittlung des Dateinamens. Wie erläutert, wird nach dem letzten Backslash im Dateinamen gesucht. Alle folgenden Zeichen gehören dann zum Dateinamen. Wenn der Dateiname von vornherein keinen Backslash enthält (beispielsweise bei Verwendung eines Netscape-Browsers), wird der gesamte Name verwendet:

```
string dateiname = d.FileName;
if (dateiname != "") { // überhaupt Upload?
  int start = dateiname.LastIndexOf("\");
  dateiname = dateiname.Substring(
```

```
                start + 1, dateiname.Length - start - 1);
  // ... Weiterverarbeitung der Datei
}
```

Der zweite Schritt besteht aus dem Kopieren der Datei:

```
string pfad = Server.MapPath("./temp/");
pfad += HttpUtility.UrlEncode(dateiname);
d.SaveAs(pfad);
```

Im letzten Schritt wird die Grafik in die Seite eingebunden – dazu wird der Name vorher noch von Sonderzeichen befreit bzw. diese in das korrekte (URL-)Format gebracht:

```
ausgabe.InnerHtml = "<img src=\"temp/" +
  HttpUtility.UrlEncode(dateiname) + "\" />";
```

Exkurs

Hier nun der vollständige Code:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Versand(Object o, EventArgs e) {
    HttpPostedFile d = datei.PostedFile;
    string dateiname = d.FileName;
    if (dateiname != "") { // überhaupt Upload?
      int start = dateiname.LastIndexOf("\");
      dateiname = dateiname.Substring(
                  start + 1, dateiname.Length - start - 1);
      string pfad = Server.MapPath("./temp/");
      pfad += HttpUtility.UrlEncode(dateiname);
      d.SaveAs(pfad);
      ausgabe.InnerHtml = "<img src=\"temp/" +
        HttpUtility.UrlEncode(dateiname) + "\" />";
    }
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" enctype="multipart/form-data" runat="server">
    <input type="file" id="datei" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Listing 4.22: Die Datei wird übertragen und ausgegeben (*html\upload2.aspx*).

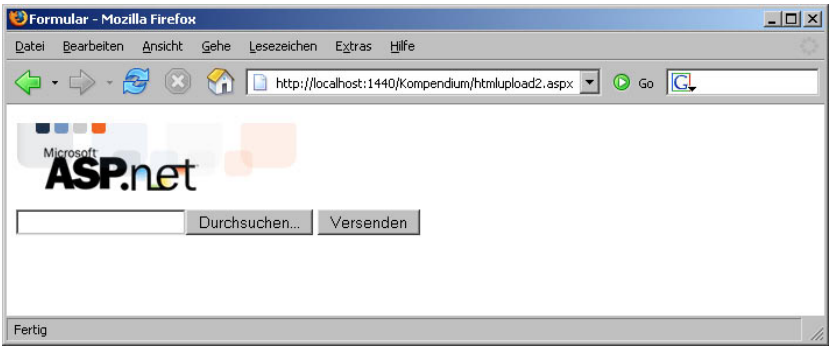

Abbildung 4.21: Die Grafik wurde hochgeladen und direkt eingebunden.

Bevor Sie diese Applikation auf einer öffentlich zugänglichen Website einsetzen, sollten Sie noch einige Sicherheitsüberprüfungen einbauen, die wir hier aus Platzgründen auslassen mussten. Dazu gehören:

- Überprüfung, ob überhaupt eine Grafik hochgeladen wurde (mindestens Überprüfung der Endung)
- Überprüfung, ob der Dateiname schon existiert (sonst würde ja wahllos überschrieben werden können)
- Überprüfung der Dateigröße (zu große Dateien sollten Sie nicht abspeichern, ansonsten stößt Ihr Webserver vermutlich mittelfristig an seine Grenzen)

* * * TIPP

Die übertragene Datei wird übrigens nur dann auf dem Webserver abgelegt, wenn Sie sie explizit mit `SaveAs()` *abspeichern; andernfalls löscht der ASP.NET-Prozess die temporäre Datei nach Beendigung des Skripts.*

4.6 Daten im Kopfabschnitt der Seite

Wie in Tabelle 4.1 bereits angedeutet, gibt es im .NET Framework auch HTML Controls, die sich um den Kopfabschnitt (`<head>`) einer Seite kümmern:

- `HTMLHead` steht für den <head>-Abschnitt der Seite an sich.
- `HtmlMeta` steht für einen `<meta>`-Tag im `<head>`-Abschnitt der Seite.

Der Zugriff erfolgt wieder wie gehabt: Sobald ein `runat="server"` am Element steht und es eine ID besitzt, ist OOP-Zugriff möglich. Das folgende Skript füllt ein (noch leeres) `<meta>`-Tag mit Inhalt:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<script runat="server">
  void Page_Load() {
    meta1.Name = "date";
    meta1.Content = DateTime.Now.ToString("yyyy-MM-dd hh:mm:ss");
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Meta</title>
  <meta id="meta1" runat="server" />
</head>
<body>
</body>
</html>
```

Listing 4.23: Ein Meta-Element wird dynamisch erzeugt (*meta.aspx*).

Das Ergebnis dieses ASP.NET-Skripts enthält unter anderem ein `<meta>`-Element nach folgendem Muster:

```
<meta id="meta1" name="date" content="2008-03-31 04:13:18" />
```

i i i INFO

Einige Varianten des `<meta>`*-Tags erfordern als Attribut* `http-equiv`*; entsprechend besitzt auch das ASP.NET-Objekt* `HTMLMeta` *die korrespondierende Eigenschaft* `HttpEquiv`.

Der Zugriff auf den `<head>`-Abschnitt einer Seite erfolgt entweder über dessen ID oder über die spezielle Eigenschaft `Page.Header`. Die Hauptanwendung besteht dann darin, auf die Eigenschaft Title zuzugreifen, denn die enthält den Seitentitel (und damit das, was zwischen `<title>` und `</title>` im HTML-Markup steht). Hier ein Beispiel dafür:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  void Page_Load() {
    Page.Header.Title = "ASP.NET 3.5";
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Head</title>
</head>
<body>
</body>
</html>
```

Listing 4.24: Der Seitentitel wird dynamisch geändert (*head.aspx*).

Hier die Ausgabe von Listing 4.24 (optisch etwas verschönert):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
  ASP.NET 3.5
</title></head>
<body>
</body>
</html>
```

ASP.NET hat also automatisch den Inhalt des `<title>`-Elements angepasst. Gerade bei einem Content Management System, bei dem eine Seite unter anderem einen speziellen Titel haben soll, ist diese Zugriffsmöglichkeit sehr hilfreich.

4.7 Fazit

In diesem Kapitel haben Sie zunächst den herkömmlichen Zugriff auf HTTP-Daten, die per GET und POST beim Skript eintreffen, kennengelernt. Dann haben Sie mit HTML Controls eine Möglichkeit im Einsatz gesehen, einen bequemen Zugriff auf HTML-Formularelemente zu erhalten. Eine Migrierung bestehender HTML-ASP-Formulare auf HTML Controls ist recht schnell erledigt; insbesondere der oft aufwendige Code für die Vorausfüllung kann wegfallen. Deswegen können wir den Einsatz dieser Elemente empfehlen, weil sich allein schon die Entwicklungszeit im Vergleich zu früher verringert. Auch ist es relativ einfach, alte Projekte (etwa in ASP) auf ASP.NET 3.5 zu migrieren.

Wenn Sie jedoch ein Projekt ganz neu von vorne aufziehen, sollten Sie einen Blick in das nächste Kapitel werfen. Dort stellen wir Ihnen eine weitere, sehr mächtige Form serverseitiger ASP.NET-Controls vor.