```
<configuration>
  <system.web>
    ...
    <pages theme="BasicGreen">
    ...
    </pages>
  </system.web>
</configuration>
```

Ha különböző `Button`, `Calendar` és `TextBox` elemeket helyeznénk a `Default.aspx` fájlba, majd futtatnánk az alkalmazást, láthatnánk, hogy az összes elemre a BasicGreen felhasználói felületet alkalmaztuk. Ha CrazyOrange-ra módosítanánk a témaattribútumot, és újra futtatnánk az alkalmazást, akkor már az ezen téma által definiált felhasználói felületet látnánk.

Témák alkalmazása oldalanként

A témákat oldalak szintjén is beállíthatjuk. Ez a lehetőség bizonyos körülmények között rendkívül hasznosnak bizonyulhat. Akkor például, ha a `web.config` fájl a teljes webhelyre kiterjedő témát definiál (ahogy az előző részben leírtuk), mi azonban más témát szeretnénk alkalmazni egy adott oldalra. Ekkor egyszerűen módosítsuk a `<%@Page%>` direktívát. Ha ezt a Visual Studio 2008 használatával tesszük, az IntelliSense az App_Theme mappa összes definiált témáját megjeleníti.

```
<%@ Page Language="C#" AutoEventWireup="true"
         CodeFile="Default.aspx.cs" Inherits="_Default"
         Theme ="CrazyOrange" %>
```

Mivel a CrazyOrange témát rendeltük hozzá az oldalhoz, a `Web.config` fájl viszont a BasicGreen témát adja meg, *ennek az oldalnak a kivételével* az összes oldal a BasicGreen témának megfelelően jelenik meg.

A SkinID tulajdonság

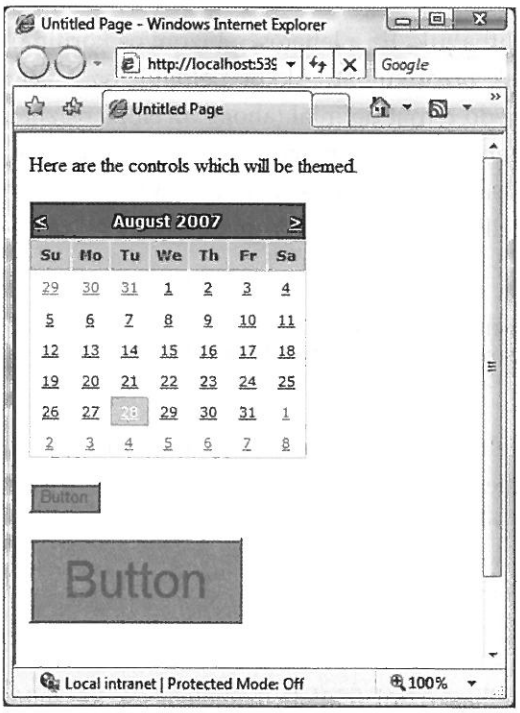
Előfordulhat, hogy a lehetséges felhasználói felületi megjelenések készletét egyetlen vezérlőelem számára szeretnénk meghatározni. Tételezzük fel például, hogy két lehetséges megjelenítést szeretnénk definiálni a `Button` típusnak a CrazyOrange témán belül. Ekkor a `SkinID` tulajdonsággal tehetünk különbséget az egyes megjelenési beállítások között:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:Button runat="server" SkinID = "BigFontButton"
     Font-Size="30pt" BackColor="#FF8000"/>
```

Azzal, hogy az oldalunk a CrazyOrange témát használja, az alapértelmezés szerint az összes Buttonhoz a névtelen `Button` arculatot rendeljük hozzá. Ha azt szeretnénk, hogy az `*.aspx` fájlban a különböző gombok a `BigFontButton` arculatot használják, állítsuk be a `SkinID` tulajdonságot a markupban:

```
<asp:Button ID="Button2" runat="server"
     SkinID="BigFontButton" Text="Button" /><br />
```

Erre mutat példát a 32.24. ábra, amelyen egy CrazyOrange témát használó oldal látható. A legfelső gombhoz a névtelen arculatot rendeltük hozzá, míg az oldal alján található gomb `SkinID` tulajdonsága a `BigFontButton` értéket vette fel.

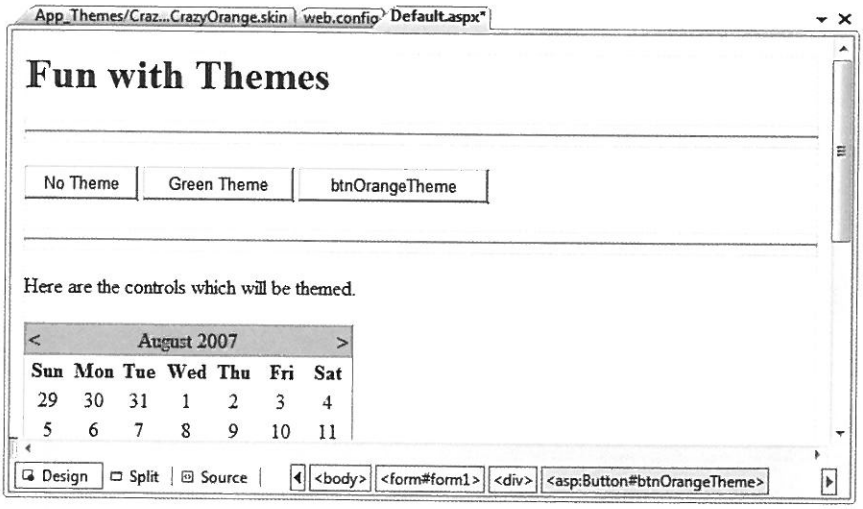


32.24. ábra: *Játék a SkinID tulajdonsággal*

Témák beállítása kódból

Végül, de nem utolsósorban, a témákat kódból is beállíthatjuk. Ez a megoldás akkor lehet hasznos, amikor biztosítani szeretnénk a végfelhasználók számára a témaválasztás lehetőségét az aktuális munkamenetben. Az állapottal rendelkező webalkalmazások létrehozását még nem vizsgáltuk, így ebben az esetben az aktuális témaválasztás eltűnik a visszaküldések között. Egy termékszintű webhelyen valószínűleg szeretnénk tárolni a felhasználó aktuális témaválasztását egy munkamenet-változóban vagy rögzíteni egy adatbázisban.

A témák programozott beállításának szemléltetésére egészítsük ki a `Default.aspx` oldalunkat három új gombbal, a 32.25. ábra alapján. Ezután kezeljük a gombok kattintási eseményét.



32.25. ábra: *A módosított felhasználói felület*

Csak az oldalak bizonyos életciklusfázisai alatt állíthatunk be témákat kódból. Ezt általában a `Page_PreInit` esemény kiváltásakor tehetjük. Ezek után a következőképpen módosul a kódfájlt:

```
partial class _Default : System.Web.UI.Page
{
  protected void btnNoTheme_Click(object sender, System.EventArgs e)
  {
    // Az üres sztringek azt eredményezik, hogy nem lesz téma
    // beállítva.
    Session["UserTheme"] = "";
```

```
    // Újra kiváltja a PreInit eseményt.
    Server.Transfer(Request.FilePath);
  }

  protected void btnGreenTheme_Click(object sender,
                                     System.EventArgs e)
  {
    Session["UserTheme"] = "BasicGreen";

    // Újra kiváltja a PreInit eseményt.
    Server.Transfer(Request.FilePath);
  }

  protected void btnOrangeTheme_Click(object sender,
                                      System.EventArgs e)
  {
    Session["UserTheme"] = "CrazyOrange";

    // Újra kiváltja a PreInit eseményt.
    Server.Transfer(Request.FilePath);
  }

  protected void Page_PreInit(object sender, System.EventArgs e)
  {
    try
    {
      Theme = Session["UserTheme"].ToString();
    }
    catch
    {
      Theme = "";
    }
  }
}
```

A UserTheme nevű kiválasztott témát, amelyet egy munkamenet-változóban tárolunk (további részletekért lásd a 33. fejezetet), formálisan a `Page_PreInit()` eseménykezelőben állítottuk be. Amikor a felhasználó egy adott gombra kattint, programozottan kényszerítjük a `PreInit` eseményt az indulásra a `Server.Transfer()` metódus hívásával és az aktuális oldal újbóli lekérésével. Ha futtatnánk az oldalt, azt látnánk, hogy különféle gombokra kattintva tudjuk bevezetni a témát.

Forráskód A FunWithThemes kódfájlokat a forráskódkönyvtár 32. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Vezérlőelemek elhelyezése HTML-táblákkal

A vezérlőelemek tervezői felületen való elhelyezése nem intuitív. A Windows Formsszal ellentétben például az alapértelmezés szerint egy felhasználóifelület-elemet nem húzhatunk ki az eszköztárból, és nem helyezhetjük *pontosan* oda, ahova szeretnénk (ez elég zavaró).

Az ASP.NET korábbi verzióiban kétféle üzemmódban lehetett pozicionálást végrehajtani (GridLayout és FlowLayout), amelyet a DOCUMENT `pageLayout` attribútumában adhattunk meg. Amikor a `GridLayout` volt beállítva, az abszolút pozicionálást lehetett elérni DHTML használatával. Ezzel azonban az ASP.NET 1.x weblapok a dinamikus HTML-t támogató böngészőkre korlátozódtak. A FlowLayout (az ASP.NET aktuális alapértelmezett üzemmódja) használatakor *nem volt elérhető* az abszolút pozicionálás, ez pedig programozáskor elég zavaró lehet. A FlowLayout azonban biztosítja, hogy minden böngésző megfelelően jeleníthesse meg a webtartalmat.

Az ASP.NET tulajdonképpen még most is lehetővé teszi a GridLayout használatát a vezérlőelemek (manuális) definiálásához. A tervezők azonban panaszkodnak, hiszen a szükséges infrastruktúra nem érvényes az XHTML-specifikációban. Nézzük meg például a következő `*.aspx` fájlt, amely egy gomb `style` attribútuma révén biztosítja a gomb abszolút pozicionálását:

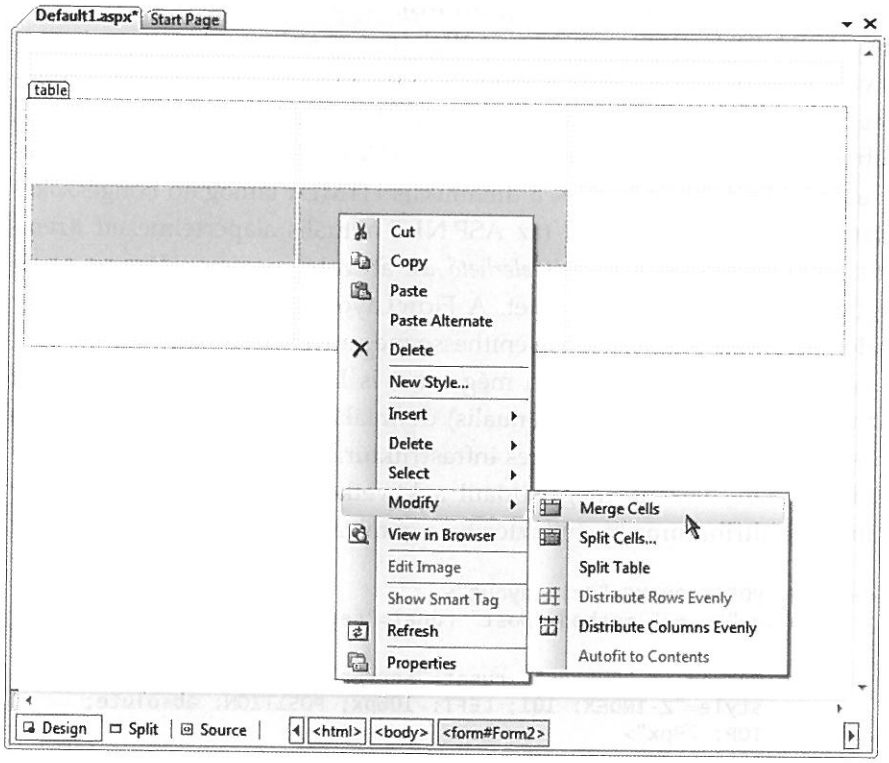
```
<body MS_POSITIONING="GridLayout">
  <form id="Form2" method="post" runat="server">

    <asp:Button id="Button1" runat="server" Text="Button"
         style="Z-INDEX: 101; LEFT: 106px; POSITION: absolute;
         TOP: 79px">
    </asp:Button>

    <asp:TextBox id="TextBox1" runat="server">
    </asp:TextBox>
  </form>
</body>
```

A nem XHTML-kompatibilis kódok használata (és annak kockázata, hogy nem működik minden böngészőben) helyett, sok webfejlesztő HTML-táblákban helyezi el a vezérlőket. A HTML-tábla a szó szoros értelmében nem látható a böngészőben, tervezés közben azonban a vezérlőelemek a cellákba helyezhetők, abszolút pozicionálást biztosítva ezzel.

A Visual Studio 2008 lehetővé teszi ezeknek a celláknak a vizuális szerkesztését és manipulálását úgy, mint egy Excel-táblázatban. A Tab billentyűvel például mozoghatunk a cellák között, és, ha több cellát is kijelölünk, egyesíthetjük vagy átméretezhetjük azokat az úszómenün (context menü) keresztül. Továbbá a cellákat testre szabhatjuk különféle stílusokkal a Properties ablak révén. A példa kedvéért nézzünk meg egy tetszőleges `*.aspx` oldalon található HTML-táblázatot a tervezőben, ezt láthatjuk a 32.26. ábrán.



32.26. ábra: *A Visual Studio 2008 kitűnő HTML-táblakonfigurációkat biztosít*

Miután konfiguráltuk a tábla celláit (amely általában további beágyazott táblákat is tartalmaz), az ASP.NET webes vezérlőelemeket a kívánt módon rendezhetjük. Az egésznek az az előnye, hogy amikor a felhasználó átméretezi a böngészőt, a vezérlőelemek megőrzik a relatív pozíciójukat.

Megjegyzés A könyv példái nem igénylik, hogy HTML-táblák segítségével tervezzünk oldalakat, ám nem haszontalan, ha tisztában vagyunk ezek hasznosságával a webes fejlesztésben.

Összefoglalás

Ebben a fejezetben a különféle ASP.NET webes vezérlőelemek használatával foglalkoztunk. A `Control` és a `WebControl` alaposztályok szerepének vizsgálatával kezdtük, és megnéztük, hogyan kell dinamikusan kommunikálni egy panel belsővezérlőelem-gyűjteményével. Eközben bemutattuk az új webhely-navigációs modellt (a `*.sitemap` fájlokat és a `SiteMapDataSource` összetevőt), az új adatkötési modult (az `SqlDataSource` összetevőn és az új `GridView` típuson keresztül), illetve különböző ellenőrző vezérlőelemeket.

A továbbiakban a mesteroldalak és a témák szerepét tekintettük át. A mesteroldalakat egy közös keret definiálására használtuk, a webhely egy lapcsoportja számára. A `*.master` fájl bármilyen számú „tartalomhelyőrzőt" definiálhat, amelyekhez a tartalomlapok csatolják az egyedi felhasználói felületi tartalmukat. Végezetül azt láthattuk, hogy az ASP.NET-téma lehetővé teszi, hogy programozottan vagy deklaratív módon biztosítsunk közös megjelenést a vezérlőelemeknek a webkiszolgálón.

HARMINCHARMADIK FEJEZET

ASP.NET állapotkezelési technikák

Az előző két fejezet az ASP.NET webes vezérlőelemek és az azokat tartalmazó lapok felépítéséről és viselkedéséről szólt. Ez a fejezet az előzőkre épül, és bemutatja a `Global.asax` fájl és az alapul szolgáló `HttpApplication` típus szerepét. Mint az látni fogjuk, a `HttpApplication` funkcionalitása több olyan esemény kezelését lehetővé teszi, amelyek segítségével a webalkalmazást különálló `*.aspx` fájlok helyett összetartó egységként kezelhetjük.

A `HttpApplication` típus vizsgálata mellett a fejezetben szót ejtünk az állapotkezeléssel kapcsolatos valamennyi fontos témakörről. Megismerjük a nézetállapotot (viewstate), a munkamenet- és az alkalmazásváltozók (az alkalmazás-gyorsítótárat is beleértve), a sütik és az ASP.NET-profil szerepét.

Az állapot

A 31. fejezet elején említettük, hogy a weben használt HTTP állapotmentes átviteli protokoll. Emiatt a webfejlesztés nagyban különbözik a végrehajtható szerelvény készítésének folyamatától. Ha például Windows Forms-alkalmazást készítünk, biztosak lehetünk abban, hogy a `Form`-leszármazott osztályban meghatározott bármely tagváltozó addig megtalálható a memóriában, amíg a felhasználó nem állítja le a végrehajtható fájlt:

```
public partial class MainWindow : System.Windows.Forms.Form
{
  // Állapotadat!
  private string userFavoriteCar = "Yugo";
}
```

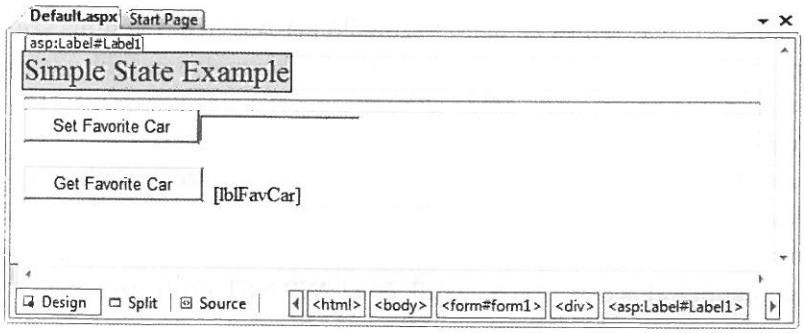
A web világában azonban ez túl nagyvonalú feltételezés. Ennek bizonyítására hozzunk létre egy új, `SimpleStateExample` nevű ASP.NET-webhelyet. Az `*.aspx` fájlhoz tartozó mögöttes kódfájlban definiáljuk a `userFavoriteCar` oldalszintű sztringváltozót:

```
public partial class _Default : System.Web.UI.Page
{
// Állapotadat?
  private string userFavoriteCar = "Yugo";

  protected void Page_Load(object sender, EventArgs e)
  {

  }
}
```

Alakítsuk ki a webes felhasználói felületet a 33.1. ábrának megfelelően.



33.1. ábra: *Az egyszerű állapotlap felhasználói felülete*

A Set gomb kiszolgálóoldali (`btnSetCar`) kattintási eseménykezelőjének segítségével a felhasználó a `txtFavCar` `TextBox` értékéhez a sztringtagváltozót rendelheti:

```
protected void btnSetCar_Click(object sender, EventArgs e)
{
  // Kedvenc autó eltárolása a tagváltozóban.
  userFavoriteCar = txtFavCar.Text;
}
```

míg a Get gomb kattintási eseménykezelője (`btnGetCar`) a tagváltozó jelenlegi értékét a lapon található címkében (`lblFavCar`) jeleníti meg:

```
protected void btnGetCar_Click(object sender, EventArgs e)
{
  // A tagváltozó értékének megjelenítése.
  lblFavCar.Text = userFavoriteCar;
}
```

Ha Windows Forms-alkalmazást készítenénk, joggal feltételezhetnénk, hogy ha egyszer a felhasználó beállítja a változó kezdőértékét, azt a rendszer megőrzi az asztali alkalmazás élettartama alatt. Sajnos, ha elindítjuk ezt a webalkalmazást, azt tapasztaljuk, hogy minden alkalommal, amikor adatot küldünk vissza a webkiszolgálónak (ha a gombokra kattintunk), a `userFavoriteCar` sztringváltozó visszaáll a „Yugo" kezdőértékre, ezért a címke mindig ugyanazt a szöveget jeleníti meg.

Mivel a HTTP protokoll automatikusan nem képes azután is megőrizni az adatokat, miután elküldte a HTTP-választ, joggal feltételezhetjük, hogy ilyenkor a `Page` objektum gyakorlatilag azonnal megsemmisül. Így, amikor az ügyfél adatot küld vissza az `*.aspx` fájlnak, új `Page` objektum jön létre, amely bármelyik lapszintű tagváltozót alaphelyzetbe állítja. Ez egyértelműen probléma. Képzeljük el, milyen lenne egy olyan webáruház, ahol minden egyes visszaküldés során a korábban megadott adatok (például a megvásárolni kívánt termékek listája) elvesznének. Ha fontos, hogy a webhelyre bejelentkezett felhasználók adatait eltároljuk, különböző állapotkezelési módszereket kell alkalmaznunk.

Megjegyzés Ez a probléma semmi esetre sem korlátozódik az ASP.NET-technológiára. A Java-szervletek, a CGI-alkalmazások, a klasszikus ASP-oldalak és a PHP-alkalmazások is megküzdenek az állapotkezelés problémájával.

Ha szeretnénk megőrizni a `userFavoriteCar` sztringtípus értékét a visszaküldések között, értékét *munkamenet-változóban* kell eltárolnunk. A munkamenet-állapot részleteit a következő oldalakon mutatjuk be. A teljesség kedvéért tekintsük meg az aktuális példához szükséges módosításokat (vegyük észre, hogy ebben már nem használjuk a privát sztringtagváltozót, így akár törölhetjük is a definíciót):

```
public partial class _Default : System.Web.UI.Page
{
  // Állapotadat?
  // private string userFavoriteCar = "Yugo";

  protected void Page_Load(object sender, EventArgs e)
  {

  }
```

```
    protected void btnSetCar_Click(object sender, EventArgs e)
    {
      // A megjegyezni kívánt értéket a munkamenet-változóban
      // tároljuk el.
      Session["UserFavCar"] = txtFavCar.Text;
    }

    protected void btnGetCar_Click(object sender, EventArgs e)
    {
      // Beolvassuk a munkamenet-változó értékét.
      lblFavCar.Text = (string)Session["UserFavCar"];
    }
}
```

Ha így lefuttatjuk programot, a `HttpSessionState` objektumnak köszönhetően a kedvenc autónk márkája megőrződik a visszaküldések között is, amelyet közvetett módon az örökölt `Session` tulajdonságon keresztül kezelünk.

Forráskód A SimpleStateExample kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Állapotkezelési módszerek az ASP.NET-ben

Az ASP.NET számos olyan megoldást kínál, amellyel a webalkalmazás állapotinformációit kezelhetjük. Az alábbi lehetőségek állnak rendelkezésünkre:

- ASP.NET-nézetállapot használata,
- ASP.NET-vezérlőelem állapotának használata,
- alkalmazásszintű változók definiálása,
- a gyorsítótár-objektum használata,
- munkamenetszintű változók definiálása,
- süti használata.

Közös pontjuk, hogy mindegyik megközelítés megköveteli, hogy a meghatározott felhasználó munkamenete és a webalkalmazás a memóriában helyezkedjen el. Amint a felhasználó kijelentkezik a webhelyről (vagy lejár az időkorlát-

ja), esetleg a webhely leáll, az alkalmazás ismét állapotmentessé válik. Ha maradandóan szeretnénk a felhasználói adatokat rögzíteni, az ASP.NET által biztosított Profile API-t kell használnunk. A következőkben valamennyi megoldást megvizsgáljuk, először az ASP.NET nézetállapotával ismerkedünk meg.

Az ASP.NET-nézetállapot szerepe

Ebben és az előző két fejezetben említettük már a *nézetállapotot*, de nem adtuk meg a formális definícióját. Itt az ideje tisztázni ezt a kifejezést. A klasszikus (COM-alapú) ASP alatt a kimenő HTTP-válasz összeállítása során a webfejlesztő feladata volt, hogy manuálisan újra adatokkal töltse fel a bejövő űrlap vezérlőinek az értékeit. Ha például a bejövő HTTP-kérésben öt szövegdoboz szerepelt meghatározott értékekkel, az `*.asp` fájl szkriptkódjának feladata volt, hogy kiolvassa az aktuális értékeket (a `Request` objektum `Form` vagy `QueryString` gyűjteményéből), és manuálisan visszahelyezze a HTTP-válaszfolyamba (szükségtelen mondani, ez mennyire unalmas feladat). Ha a fejlesztő ezt elmulasztotta megtenni, a felhasználó öt üres szövegdobozt kapott.

ASP.NET-ben nem szükséges manuálisan „kibányászni" és újra beállítani a HTML-vezérlőkben tárolt értékeket, mivel az ASP.NET-futtatókörnyezet automatikusan beágyaz egy rejtett űrlapmezőt (`__VIEWSTATE` néven), amelyet a rendszer a böngésző és a megadott oldal között továbbít. A mezőben egy Base64-kódolt sztring található, amely az adott oldalon a felhasználói felületelemek név/érték párjait foglalja magában.

A `System.Web.UI.Page` alaposztály `Init` eseménykezelője felel a `__VIEWSTATE` mezőben található bemeneti értékek beolvasásáért és a leszármazott osztályban a megfelelő tagváltozók feltöltéséért (ezért kockázatos a webes vezérlők állapotának hozzáférése az oldal `Init` eseménykezelőjében).

Mielőtt a kimenő választ elküldenénk a böngészőnek, a rendszer a `__VIEWSTATE` adatok segítségével feltölti az űrlap vezérlőit, és biztosítja, hogy a HTML-vezérlők aktuális értékei az előző visszaküldés előtti állapotnak megfelelően jelennek meg.

Az ASP.NET ezen szolgáltatásában az a legjobb, hogy anélkül működik, hogy ezért nekünk bármit tennünk kellene. Természetesen, igény szerint alkalmazhatjuk, módosíthatjuk vagy akár letilthatjuk az alapértelmezett funkcionalitást. Hogy megértsük a működését, nézzünk meg egy konkrét példát.

A nézetállapot bemutatása

Hozzuk létre a ViewStateApp új ASP.NET-webalkalmazást. A kezdő `*.aspx` oldalhoz adjunk hozzá egyetlen ASP.NET ListBox vezérlőelemet (myListBox néven) és egy sima Buttont (`btnPostBack` néven). Kezeljük le a gomb kattintási eseményét, ezzel biztosítsunk lehetőséget a felhasználónak, hogy adatokat küldjön vissza a webkiszolgálónak:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
  protected void btnPostback_Click(object sender, EventArgs e)
  {
    // Nincs művelet. Csak azért van itt a metódus,
    // hogy biztosítsuk a visszaküldést.
  }
}
```

A Visual Studio 2008 Properties ablakában válasszuk ki az `Items` tulajdonságot, és vegyünk fel négy `ListItem` vezérlőelemet a ListBoxhoz a kapcsolódó párbeszédablak segítségével. Eredményül az alábbi markuphoz hasonlót kapunk:

```
<asp:ListBox ID="myListBox" runat="server">
  <asp:ListItem>Item One</asp:ListItem>
  <asp:ListItem>Item Two</asp:ListItem>
  <asp:ListItem>Item Three</asp:ListItem>
  <asp:ListItem>Item Four</asp:ListItem>
</asp:ListBox>
```

Vegyük észre, hogy a ListBox elemeket közvetlenül az `*.aspx` fájlba drótozzuk. Mint azt már tudjuk, a HTML-űrlapon található összes `<asp:>` definíció automatikusan kirendereli a HTML megjelenítését a végső HTTP-válasz előtt (feltéve, hogy rendelkeznek a `runat="server"` attribútummal).

A `<%@Page%>` direktíva rendelkezik az opcionális `EnableViewState` attribútummal, amelynek értéke alapértelmezés szerint `true`. A viselkedés letiltásához a következőképpen módosítsuk a `<%@Page%>` direktívát:

```
<%@ Page Language="C#" AutoEventWireup="true"
         CodeFile="Default.aspx.cs" Inherits="_Default"
         EnableViewState ="false" %>
```

Pontosan mit jelent a nézetállapot letiltása? A válasz az, hogy attól függ. Ha a kifejezés előző definícióját nézzük, úgy gondolhatnánk, hogy ha letiltjuk az `*.aspx` fájl nézetállapotát, akkor a webkiszolgálóhoz irányított visszaküldések között elvesznének a ListBox értékei. Ezzel szemben, ha így futtatjuk az alkalmazást, meglepve tapasztaljuk, hogy a ListBox értéke attól függetlenül megmarad, hogy hányszor küldünk vissza adatot.

Sőt, ha megnézzük a böngészőnek küldött forrás-HTML-t (a böngészőben az oldalon kattintsunk a jobb gombbal, majd válasszuk a View Source menüpontot), azt látjuk, hogy a rejtett `__VIEWSTATE` mező *még mindig jelen van*:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
    value="/wEPDwUKLTM4MTM2MDM4NGRkqGC6gjEV25JnddkJiRmoIc10SIA=" />
```

A nézetállapotsztring azért látható még mindig, mert az `*.aspx` fájl a ListBox elemeit a HTML `<form>` címkék hatókörében explicit módon definiálja. Így a rendszer a ListBox-elemeket automatikusan újragenerálja, ha a webkiszolgáló a kliensnek választ küld.

Tételezzük fel, hogy a ListBox-elemet dinamikusan töltjük fel a mögöttes kódfájlban, nem pedig a HTML `<form>` definíciójában. Először is, töröljük az `<asp:ListItem>` deklarációkat az aktuális `*.aspx` fájlból:

```
<asp:ListBox ID="myListBox" runat="server">
</asp:ListBox>
```

Majd a mögöttes kódfájlban a `Load` eseménykezelőjéből töltsük fel a listaelemeket:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    // A ListBox dinamikus feltöltése!
    myListBox.Items.Add("Item One");
    myListBox.Items.Add("Item Two");
    myListBox.Items.Add("Item Three");
    myListBox.Items.Add("Item Four");
  }
}
```

Ha először megnézzük a módosított oldalt, azt tapasztaljuk, hogy az első alkalommal, amikor a böngésző az oldalt kéri, a ListBox értékei jelen vannak és használhatók. Viszont, ha adatot küldünk vissza az oldalnak, a ListBox hirtelen üres lesz. Az ASP.NET-nézetállapot első szabálya, hogy a hatását csak

akkor fejti ki, ha rendelkezünk olyan vezérlőkkel, amelyek értékeit dinamikusan, kódból generáljuk. Ha az *.aspx fájl <form> címkéibe kódolunk értékeket, az elemek állapota a visszaküldések között mindig megmarad (még akkor is, ha az adott oldalon az EnableViewState értékét false-ra állítjuk).

A nézetállapotnak akkor vesszük a legnagyobb hasznát, ha olyan dinamikusan feltöltött webes vezérlőkkel rendelkezünk, amelyet minden egyes visszaküldés alkalmával újra fel kell tölteni (például egy ASP.NET GridView-val, amely mindig adatbázisból jelenít meg találatokat). Ha nem tiltjuk le az ilyen vezérlőket tartalmazó lapokon a nézetállapotot, a tábla egész állapotát a rejtett __VIEWSTATE mező képviseli. Mivel az összetett oldalak számos ASP.NET webes vezérlőelemet tartalmazhatnak, nem nehéz elképzelni, mekkorára nőhet a sztring mérete. A HTTP kérés/válasz ciklusok költsége jelentősen megnőhet, és ez problémát okozhat azok számára, akik betárcsázós internetelérással rendelkeznek. Ilyen esetekben növelhetjük az adatátvitel sebességét, ha letiltjuk a lapon a nézetállapotot.

A nézetállapot teljes *.aspx fájlra történő letiltása meglehetősen drasztikus lépésnek tűnhet. Jó, ha tisztában vagyunk azzal, hogy a System.Web.UI.Control ősosztály valamennyi leszármazottja örökli az EnableViewState tulajdonságot, amely megkönnyíti a nézetállapotot letiltását vezérlőelemenként:

```
<asp:GridView id="myHugeDynamicallyFilledGridOfData" runat="server"
     EnableViewState="false">
</asp:GridView>
```

Megjegyzés Az ASP.NET-oldalak a __VIEWSTATE kis részét belső használatra tartják fenn. Ez azt jelenti, hogy a __VIEWSTATE mező akkor is megtalálható az ügyféloldali forráskódban, ha az egész lapon (és annak minden vezérlőelemén) letiltottuk a nézetállapotot.

Egyedi nézetállapot-adat hozzáadása

Az EnableViewState tulajdonság mellett a System.Web.UI.Control ősosztály a ViewState örökölt tulajdonságot biztosítja. Valójában ez a tulajdonság a System.Web.UI.StateBag típushoz biztosít hozzáférést, amelyben a __VIEWSTATE mezőben tárolt összes adatot képviseli. A StateBag típus indexelőjének használatával egyedi információkat ágyazhatunk a rejtett __VIEWSTATE mezőbe név/érték párok segítségével. Íme, egy egyszerű példa:

```
protected void btnAddToVS_Click(object sender, EventArgs e)
{
  ViewState["CustomViewStateItem"] = "Some user data";
  lblVSValue.Text = (string)ViewState["CustomViewStateItem"];
}
```

Mivel a `System.Web.UI.StateBag` típus célja, hogy bármely, tetszőleges típusból származó `System.Object` objektummal működjön, ha egy adott kulcshoz tartozó értékhez szeretnénk hozzáférni, azt explicit módon a megfelelő alapadattípusra (jelen esetben `System.String`) kell kasztolnunk. Legyünk óvatosak, mert a `__VIEWSTATE` mezőben nem helyezhetünk el bármilyen objektumot. Valójában csak a `String`, az `Integer`, a `Boolean`, az `ArrayList` és a `Hashtable` típusok vagy az ezekből álló tömbök használhatók.

Mivel az `*.aspx` lapok egyedi információrészleteket helyezhetnek el a `__VIEWSTATE` sztringben, a következő logikus kérdés, hogy mikor célszerű ezt használni. Az esetek többségében az egyedi nézetállapot-adatok a felhasználói beállítások tárolására a legalkalmasabbak. A nézetállapotban például eltárolhatuk olyan adatokat, amelyek meghatározzák, hogy a felhasználó hogyan kívánja megtekinteni a GridView elem felhasználó felületét (például rendezéssorrendet). A nézetállapot-adatok nem alkalmasak nagy mennyiségű felhasználói adat – mint például egy bevásárlókosár tartalma vagy gyorsítótárazott `DataSet` – kezelésére. Az ilyen jellegű bonyolult adatok tárolásához munkamenet- vagy alkalmazásadatokkal kell dolgoznunk. De még mielőtt ezeket megnéznénk, meg kell értenünk a `Global.asax` fájl szerepét.

Forráskód A ViewStateApp kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

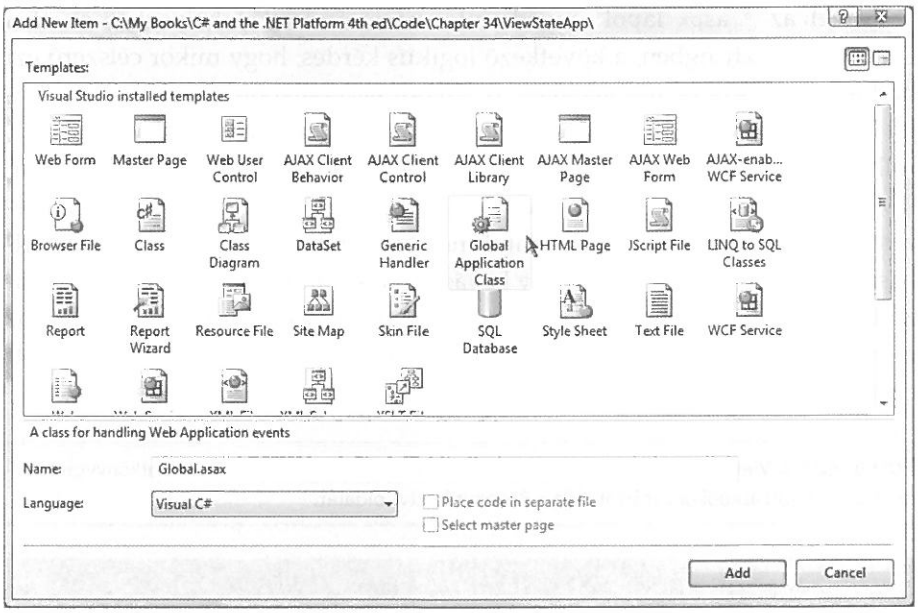
NÉHÁNY SZÓ A VEZÉRLŐELEM-ÁLLAPOTRÓL

A .NET 2.0 megjelenése óta a vezérlőelemek állapotát a nézetállapot helyett *vezérlőelem-állapot* segítségével célszerű eltárolni. Ez a módszer akkor a leghasznosabb, ha olyan egyedi ASP.NET webes vezérlőelemet fejlesztettünk, amelynek az üzenetküldések között meg kell őriznie az adatokat. Noha a `ViewState` tulajdonságot használhatjuk erre a célra, ha a nézetállapotot lapszinten tiltjuk le, az egyéni vezérlőelem gyakorlatilag használhatatlan. A webes vezérlőelemek pontosan ezért támogatják a `ControlState` tulajdonságot.

A vezérlőelem-állapot a nézetállapothoz hasonlóan működik. De ezt az állapotot a rendszer akkor sem tiltja le, ha a nézetállapotot lapszinten letiltottuk. Mint azt már említettük, ez a szolgáltatás az egyedi webes vezérlőelemek fejlesztői számára a leghasznosabb. Erre a témakörre nem térünk ki részletesen a könyvben, további információkért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

A Global.asax fájl szerepe

Jelenleg az ASP.NET-alkalmazás kicsit többnek tűnik, mint `*.aspx` fájlok és a hozzájuk tartozó webes vezérlőelemek halmaza. Úgy is felépíthetünk egy webalkalmazást, hogy csak összekapcsolunk néhány összetartozó weboldalt, de valószínűleg szükségünk lesz egy módszerre, amellyel az egész webalkalmazással kommunikálhatunk. Ezért az ASP.NET-webalkalmazásokhoz hozzáadhatunk egy opcionális, `Global.asax` fájlt a Web Site > Add New Item menüpont segítségével, ahogyan azt a 33.2. ábrán láthatjuk (vegyük észre, hogy a Global Application Class ikont kell választanunk).



33.2. ábra: *A Global.asax fájl*

Tulajdonképpen a `Global.asax` fájl közel áll a hagyományos, két kattintással indítható `*.exe` fájlokhoz, amelyekkel az ASP.NET világában találkozhatunk, és ez azt jelenti, hogy a típus a webhely futási idejű működését képviseli. Ha hozzáadjuk a `Global.asax` fájlt a webprojekthez, észrevehetjük, hogy alig több mint egy eseménykezelő halmazt magában foglaló `<script>` blokk:

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

<script runat="server">
  void Application_Start(object sender, EventArgs e)
```

```
    {
      // Az alkalmazás indulásakor lefuttatott kód.
    }

    void Application_End(object sender, EventArgs e)
    {
      // Az alkalmazás leállásakor lefuttatott kód.
    }

    void Application_Error(object sender, EventArgs e)
    {
      // Kezeletlen hibák bekövetkezésekor lefuttatott kód.
    }

    void Session_Start(object sender, EventArgs e)
    {
      // Új munkamenet létrehozásakor lefuttatott kód.
    }

    void Session_End(object sender, EventArgs e)
    {
      // A munkamenet végén lefuttatott kód. Megjegyzés:
      // A Session_End esemény csak akkor következik be, ha a
      // Web.config fájlban a munkamenet-állapotmód értéke InProc.
      // Ha a munkamenet módja StateServer vagy SQLServer, az esemény
      // nem következik be.
    }
</script>
```

A látszat azonban csal. A futásidőben a <script> blokkban található kód egy, a `System.Web.HttpApplication` típusból származó osztálytípusba kerül (ha rendelkezünk ASP.NET 1.x háttérismeretekkel, emlékezhetünk rá, hogy a `Global.asax` mögöttes kódfájlja valóban definiált egy `HttpApplication` típusból származó osztályt).

Mint említettük, a `Global.asax` fájlban definiált tagok eseménykezelők, amelyek segítségével alkalmazás- és munkamenetszintű eseményekkel működhetünk együtt. A 33.1. táblázat foglalja össze a tagok szerepét.

Eseménykezelő	Jelentés
`Application_Start()`	Ezt az eseménykezelőt a webalkalmazás legelső indulásakor hívja a rendszer. Ez azt jelenti, hogy a webalkalmazás élettartama során az esemény pontosan egyszer lép működésbe. Itt célszerű definiálni azokat az alkalmazásszintű adatokat, amelyeket a teljes webalkalmazásban használni kívánunk.

Eseménykezelő	Jelentés
`Application_End()`	Ez az eseménykezelő az alkalmazás leállítása során hívódik. Akkor következik be, amikor a legutolsó felhasználó időkorlátja lejár, vagy az alkalmazást az IIS-en keresztül leállítjuk.
`Session_Start()`	Az esemény akkor lép működésbe, amikor egy új felhasználó bejelentkezik az alkalmazásba. Itt állíthatunk be felhasználóspecifikus adatokat.
`Session_End()`	Ez az eseménykezelő a felhasználói munkamenet leállítása során (általában előre definiált időtúllépés miatt) hívódik.
`Application_Error()`	Ez a globális hibakezelő akkor hívódik, ha a webalkalmazás kezeletlen kivételt jelez.

33.1. táblázat: *A System.Web névtér alapvető típusai*

A véső, globális kivételkezelő

Először is, nézzük meg az `Application_Error()` eseménykezelő szerepét. Emlékezzünk rá, hogy az oldalak kezelhetik az `Error` eseményt, és így gondoskodhatnak az oldal hatókörében keletkezett kezeletlen kivételekről. Hasonlóképpen, az `Application_Error()` eseménykezelő az utolsó lehetőség, ahol az adott lap által fel nem dolgozott kivételek kezeléséről gondoskodhatunk. Az oldalszintű `Error` esemény kezeléséhez hasonlóan, az örökölt Server tulajdonság segítségével hozzáférhetünk a konkrét `System.Exceptionhöz`:

```
void Application_Error(object sender, EventArgs e)
{
  // A kezeletlen hiba beolvasása.
  Exception ex = Server.GetLastError();

  // A hiba feldolgozása...

  // A befejezést követően a hiba törlése.
  Server.ClearError();
}
```

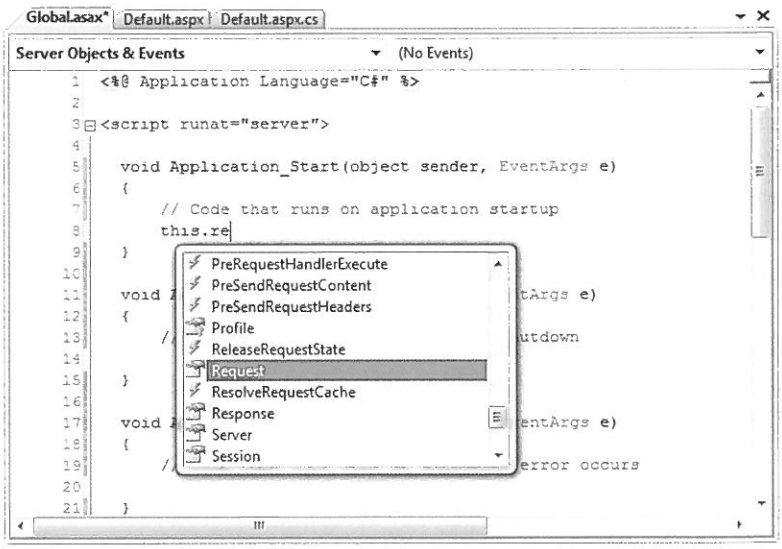
Mivel az `Application_Error()` eseménykezelő a webalkalmazásunk végső kivételkezelője, ezt a metódust gyakran úgy szokták implementálni, hogy átirányítsa a felhasználót a kiszolgálón egy előre meghatározott hibaoldalra. Az egyéb hibakezeléssel kapcsolatos műveletek közé tartozik például e-mail értesítés küldése a webes rendszergazdának vagy külső hibanapló írása.

A HttpApplication ősosztály

Amint azt már említettük, a `Global.asax` szkript dinamikusan a `System.Web.HttpApplication` ősosztályból származó osztályba generálódik, amely nagyjából ugyanazt a funkcionalitást biztosítja, mint a `System.Web.UI.Page` típus (a látható felhasználói felület nélkül). A 33.2. táblázat bemutatja a kulcsfontosságú tagokat.

Tulajdonság	Jelentés
`Application`	Ez a tulajdonság lehetővé teszi, hogy alkalmazásszintű változókkal dolgozzunk a rendelkezésünkre álló `HttpApplicationState` típus segítségével.
`Request`	Ez a tulajdonság az alapul szolgáló `HttpRequest` objektum segítségével lehetővé teszi a bejövő HTTP-kérés kezelését.
`Response`	Ez a tulajdonság az alapul szolgáló `HttpResponse` objektum segítségével lehetővé teszi a bejövő HTTP-válasz kezelését.
`Server`	Ez a tulajdonság az alapul szolgáló `HttpServerUtility` objektum segítségével lekérdezi az aktuális kérés belső kiszolgáló-objektumát.
`Session`	Ez a tulajdonság az alapul szolgáló `HttpSessionState` objektum segítségével lehetővé teszi munkamenetszintű változók kezelését.

33.2. táblázat: *A System.Web.HttpApplication típus kulcsfontosságú tagjai*



33.3. ábra: *Ne feledjük, hogy a Global.asax fájlban rejtőzködő tagok szülője a HttpApplication osztály*

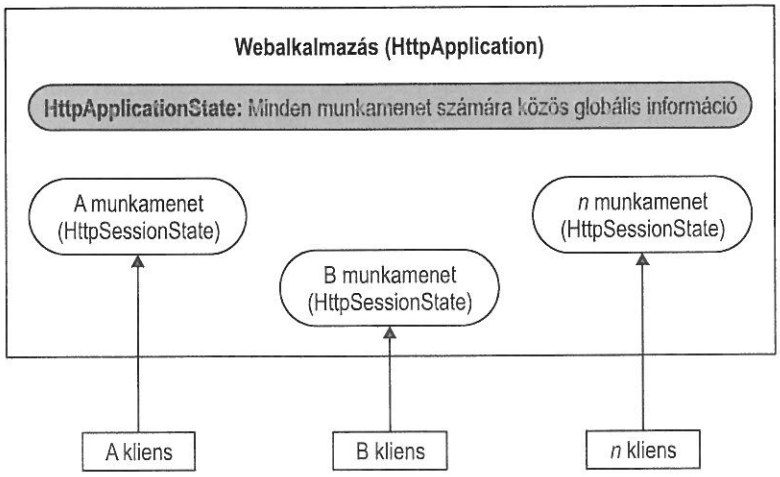
Mivel a `Global.asax` fájl explicit módon nem dokumentálja, hogy alapjául a `HttpApplication` ősosztály szolgál, fontos megjegyeznünk, hogy az „az egy" kapcsolat szabályai valóban érvényesek. Ha például a `Global.asax` fájl bármely tagjában a `base` kulcsszón a pont operátort alkalmazzuk, tapasztalhatjuk, hogy a származási lánc összes tagjához közvetlen hozzáféréssel rendelkezünk, mint az a 33.3. ábrán látható.

Az alkalmazás és a munkamenet közötti különbség

ASP.NET-ben az alkalmazás állapotát a `HttpApplicationState` típus példánya tartja karban. Az osztály segítségével globális információkat oszthatunk meg az ASP.NET-alkalmazásba bejelentkezett összes felhasználó (és minden lap) között. Nemcsak alkalmazásadatokat oszthatunk meg a webhely felhasználói között, de ha egy alkalmazásszintű adat értéke megváltozik, az új értékről a következő visszaküldés során minden felhasználó értesül.

Ezzel szemben a munkamenet-állapot segítségével meghatározott felhasználó információit tárolhatjuk (mint például egy bevásárlókosár tartalmát). Fizikailag a felhasználó munkamenetének állapotát a `HttpSessionState` osztály képviseli. Amikor egy új felhasználó bejelentkezik az ASP.NET-webalkalmazásba, a futtatórendszer automatikusan új munkamenet-azonosítót rendel a felhasználóhoz, amely alapértelmezés szerint 20 percnyi tétlenség után lejár. Így, ha a webhelyre 20000 felhasználó jelentkezik be, 20000 különböző `HttpSessionState` objektummal rendelkezünk, amelyek mindegyike automatikusan egyedi munkamenet-azonosítót kap. A webalkalmazás és a webes munkamenetek kapcsolatát a 33.4. ábra szemlélteti.

Korábbi tapasztalataink alapján emlékezhetünk rá, a hagyományos ASP-ben az alkalmazás- és munkamenetállapo- adatokat különálló COM-objektumok (például `Application` és `Session`) képviselik. ASP.NET-ben a `Page` osztályból származó típusok, valamint a `HttpApplication` megegyező nevű tulajdonságokat (pl. `Application` és `Session`) használhatnak, amelyeken keresztül az alapul szolgáló `HttpApplicationState` és `HttpSessionState` típusok érhetőek el.



33.4. ábra: *Az alkalmazás- és a munkamenet-állapot megkülönböztetése*

Alkalmazásszintű állapotadatok karbantartása

A `HttpApplicationState` típus segítségével a fejlesztők az ASP.NET-alkalmazásban több munkamenet között oszthatnak meg globális információkat. Így például karbantarthatunk alkalmazásszintű kapcsolatsztringeket, amelyeket minden oldal használhat, közös `DataSet` típust, amelyet több oldal alkalmazhat, vagy egyéb adatrészletet, amelyhez alkalmazásszinten kell hozzáférnünk. A 33.3. táblázat bemutatja a típus néhány alapvető tagját.

Tagok	Jelentés
`Add()`	A metódus segítségével új név/érték adatpárokat vehetünk fel a `HttpApplicationState` típusban. Jegyezzük meg, hogy a metódust rendszerint nem részesítjük előnyben a `HttpApplicationState` osztály indexelőjével szemben.
`AllKeys`	Visszaadja egy `System.String` típusú tömböt, amely a `HttpApplicationState` típusban található összes nevet képviseli.
`Clear()`	A metódus törli a `HttpApplicationState` típus valamennyi elemét. Funkcionális szempontból megegyezik a `RemoveAll()` metódussal.
`Count`	Ez a tulajdonság lekérdezi a `HttpApplicationState` típusban található objektumok számát.

Tagok	Jelentés
`Lock()`, `Unlock()`	Ezt a két metódust akkor használjuk, ha az alkalmazásváltozók készletét szálbiztos módon szeretnénk módosítani.
`RemoveAll()`, `Remove()`, `RemoveAt()`	Ezek a metódusok törölnek egy megadott elemet (a neve alapján) a `HttpApplicationState` típusból. A `RemoveAt()` az elemet numerikus index alapján távolítja el.

33.3. táblázat: *A HttpApplicationState típus tagjai*

Az alkalmazásállapot működésének bemutatásához hozzunk létre új ASP.NET-webalkalmazást `AppState` néven, és adjunk hozzá új `Global.asax` fájlt. Az aktív munkamenetek között megosztható adattagok létrehozása során meg kell adnunk egy név/érték párt. Általában ezt a `HttpApplication` osztályból származó típus `Application_Start()` eseménykezelőjében a legegyszerűbb megtenni, például a következőképpen:

```
void Application_Start(Object sender, EventArgs e)
{
  // Alkalmazásváltozók beállítása.
  Application["SalesPersonOfTheMonth"] = "Chucky";
  Application["CurrentCarOnSale"] = "Colt";
  Application["MostPopularColorOnLot"] = "Black";
}
```

A webalkalmazás élettartama során (amely addig tart, míg a webalkalmazást manuálisan le nem állítjuk, vagy a legutolsó felhasználó időkorlátja le nem jár), bármely felhasználó (bármely lapról) szükség szerint elérheti ezeket az értékeket. Tételezzük fel, hogy az oldalunk az éppen leértékelt autót jeleníti meg egy címkén a gomb kattintási eseménykezelőjének segítségével:

```
protected void btnShowCarOnSale_Click(object sender, EventArgs arg)
{
  lblCurrCarOnSale.Text = string.Format("Sale on {0}'s today!",
                          (string)Application["CurrentCarOnSale"]);
}
```

Vegyük észre, hogy a `ViewState` tulajdonsághoz hasonlóan a `HttpApplicationState` típus által visszaadott értéket kasztolni kell a megfelelő adattípusra, mivel az `Application` tulajdonság is általános `System.Object` típusokkal dolgozik.

Mivel a HttpApplicationState típus tetszőleges típusú információt tárolhat, beláthatjuk, hogy egyedi típusokat (vagy bármilyen .NET-objektumot) elhelyezhetünk a webhely alkalmazásállapotában. Tételezzük fel, hogy szívesebben tárolnánk a három aktuális alkalmazásváltozót az erősen típusos CarLotInfo nevű osztályban:

```
public class CarLotInfo
{
  public CarLotInfo(string s, string c, string m)
  {
    salesPersonOfTheMonth = s;
    currentCarOnSale = c;
    mostPopularColorOnLot = m;
  }
  // A könnyebb elérhetőség miatt publikus, de használhatnánk
  // az automatikustulajdonság-szintaxist is.
  public string salesPersonOfTheMonth;
  public string currentCarOnSale;
  public string mostPopularColorOnLot;
}
```

Ha elkészítettük a segédosztályt, a következőképpen módosítsuk az Application_Start() eseménykezelőt:

```
void Application_Start(Object sender, EventArgs e)
{
  // Az egyedi objektum elhelyezése az alkalmazás-adatszektorban.
  Application["CarSiteInfo"] =
            new CarLotInfo("Chucky", "Colt", "Black");
}
```

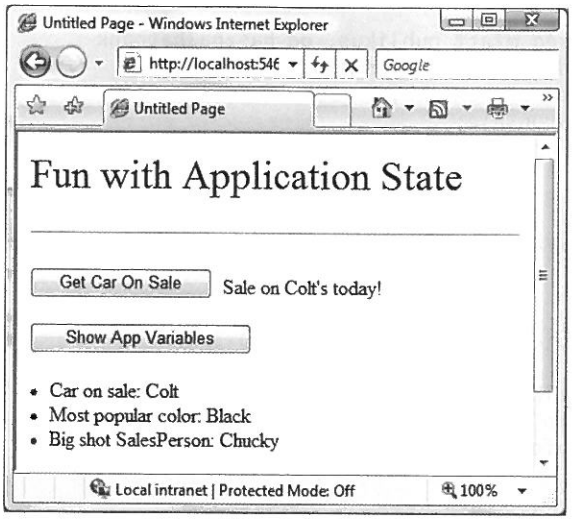
majd a btnShowAppVariables nevű gomb kiszolgálóoldali kattintási eseménykezelőjéből a nyilvános adatmezők segítségével férjünk hozzá az adatokhoz:

```
protected void btnShowAppVariables_Click(object sender, EventArgs e)
{
  CarLotInfo appVars =
            ((CarLotInfo)Application["CarSiteInfo"]);
  string appState =
         string.Format("<li>Car on sale: {0}</li>",
         appVars.currentCarOnSale);
  appState +=
     string.Format("<li>Most popular color: {0}</li>",
     appVars.mostPopularColorOnLot);
  appState +=
     string.Format("<li>Big shot SalesPerson: {0}</li>",
     appVars.salesPersonOfTheMonth);
  lblAppVariables.Text = appState;
}
```

Mivel az aktuális leértékelt autót egyedi osztálytípus segítségével tesszük elérhetővé, a `btnShowCarOnSale` kattintási eseménykezelőjét a következőképpen kell módosítanunk:

```
protected void btnShowCarOnSale_Click1(object sender, EventArgs e)
{
  lblCurrCarOnSale.Text = String.Format("Sale on {0}'s today!",
        ((CarLotInfo)Application["CarSiteInfo"]).currentCarOnSale);
}
```

Ha futtatjuk a programot, láthatjuk, hogy a 33.5. ábrának megfelelően az alkalmazásváltozók listája megjelenik a lap címkéiben.



33.5. ábra: *Alkalmazásadatok megjelenítése*

Alkalmazásadatok módosítása

A `HttpApplicationState` típus tagjaival programozottan frissíthetjük vagy törölhetjük bármely vagy akár az összes tagot a webalkalmazás futása során. Meghatározott elem törléséhez például egyszerűen a `Remove()` metódust kell meghívni. Ha az összes alkalmazásszintű adatot szeretnénk megsemmisíteni, a `RemoveAll()` metódust kell használnunk:

```
private void CleanAppData()
{
  // Egyetlen elem törlése sztringnév segítségével.
  Application.Remove["SomeItemIDontNeed"];

  // Minden alkalmazásadat törlése!
  Application.RemoveAll();
}
```

Ha meglévő alkalmazásszintű változó értékét szeretnénk megváltoztatni, egyszerűen hozzá kell rendelnünk az új értéket a kérdéses adatelemhez. Tételezzük fel, hogy az oldal egy új `Button` típust támogat, amely lehetővé teszi a felhasználó számára, hogy a `NewSP` nevű szövegdoboz segítségével beolvasott értékre módosítsa a hónap kereskedőjét. A kattintási eseménykezelője az elvárásainknak megfelelő:

```
protected void btnSetNewSP_Click(object sender, EventArgs e)
{
  // Új kereskedő beállítása.
  ((CarLotInfo)Application["CarSiteInfo"]).salesPersonOfTheMonth =
      txtNewSP.Text;
}
```

Ha futtatjuk a webalkalmazást, azt tapasztaljuk, hogy az alkalmazásszintű változó értéke módosult. Továbbá, mivel az alkalmazásváltozók minden felhasználói munkamenetből elérhetők, ha a webböngésző három vagy négy példányát indítjuk el, azt tapasztaljuk, hogy ha az egyik példány módosítja a hónap kereskedőjének nevét, visszaküldéskor a változás a többi böngészőben is megjelenik.

Előfordulhat olyan helyzet, amelyben több alkalmazásszintű változót egységként kell kezelnünk és módosítanunk. Ilyenkor fennállhat az adatsérülés veszélye (mivel megtörténhet, hogy alkalmazásszintű adatot éppen akkor módosítunk, amikor másvalaki megpróbál azokhoz hozzáférni). Bár választhatnánk a nehezebb utat, és zárolhatnánk manuálisan a logikát (a `System.Threading` névtér primitívjeinek segítségével), a `HttpApplicationState` típus azonban biztosítja a `Lock()` és az `Unlock()` metódust, amelyek automatikusan gondoskodnak a szálbiztonságról:

```
// Kapcsolódó alkalmazásadat biztonságos hozzáférése.
Application.Lock();
Application["SalesPersonOfTheMonth"] = "Maxine";
Application["CurrentBonusedEmployee"] =
          Application("SalesPersonOfTheMonth");
Application.Unlock();
```

A webalkalmazás leállításának kezelése

A `HttpApplicationState` típus addig tartja karban a tárolt elemek értékeit, amíg a következő két helyzet valamelyike be nem következik: a webhely utolsó felhasználója túllépi az időkorlátot (vagy kijelentkezik), vagy valaki manuálisan leállítja a webhelyet az IIS segítségével. Mindkét esetben a rendszer automatikusan a `HttpApplication`-leszármazott típus `Application_End()` metódusát hívja meg. Ebben az eseménykezelőben elhelyezhetjük az alkalmazás leállításakor végrehajtani kívánt kódot:

```
void Application_End(Object sender, EventArgs e)
{
  // Az aktuális alkalmazásváltozók kiírása
  // adatbázisba vagy egyéb szükséges kód...
}
```

Forráskód Az AppState kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az alkalmazás-gyorsítótár használata

Az ASP.NET-ben másik, rugalmasabb módszerrel is kezelhetjük az alkalmazásszintű adatokat. Emlékezzünk rá, hogy a `HttpApplicationState` objektum addig tárolja memóriában az értékeket, amíg a webalkalmazás fut és működik. Bizonyos esetekben az alkalmazásadatok egy részét csak meghatározott ideig szeretnénk tárolni. Olyan ADO.NET `DataSet` objektumra van például szükségünk, amely csak öt percig érvényes. Öt perc leteltével friss `DataSet` objektumot olvasunk be, hogy gondoskodjunk a lehetséges adatbázis-módosításokról. Bár gyakorlatilag ezt az infrastruktúrát a `HttpApplicationState` osztály és egy manuálisan megírt megfigyelő segítségével megvalósíthatjuk, feladatunkat az ASP.NET alkalmazás-gyorsítótár alkalmazása jelentősen egyszerűbbé teszi.

Az ASP.NET `System.Web.Caching.Cache` objektuma (amely a `Context.Cache` tulajdonságon keresztül érhető el) lehetővé teszi olyan objektum létrehozását, amely meghatározott ideig minden felhasználó számára (minden oldalról) hozzáférhető. A legegyszerűbb formájában a gyorsítótár használata megegyezik a `HttpApplicationState` típussal folytatott munkával:

```
// Elem felvétele a gyorsítótárba.
// Az elem érvényessége *nem* jár le.
Context.Cache["SomeStringItem"] = "This is the string item";

// Elem lekérdezése a gyorsítótárból.
string s = (string)Context.Cache["SomeStringItem"]
```

Megjegyzés Ha a `Cache` objektumot a `Global.asax` fájlból szeretnénk elérni, a `Context` tulajdonságot kell használnunk. Ha viszont a `System.Web.UI.Page`-leszármazott típus hatókörében vagyunk, közvetlenül használhatjuk a `Cache` objektumot.

Ha nem áll szándékunkban az alkalmazásszintű adatpontot automatikusan frissíteni (vagy eltávolítani), a `Cache` objektum használatából nem származik előnyünk, mivel közvetlenül használhatjuk a `HttpApplicationState` típust. Ha azonban azt szeretnénk, hogy az adatpont meghatározott idő után megsemmisüljön – és erről akár értesülni is szeretnénk–, a `Cache` típus nagyon hasznosnak bizonyulhat.

A `System.Web.Caching.Cache` osztályban kevés tag található a típus indexelőjén kívül. Az `Add()` metódussal vehetünk fel például új, még nem definiált elemet a gyorsítótárba (ha a meghatározott elem már létezik, az `Add()` gyakorlatilag semmit sem csinál). Az `Insert()` metódus is tagot helyez el a gyorsítótárban. De ha az elemet már definiáltuk, az `Insert()` felülírja a típust az aktuális elemmel. Mivel leginkább erre a viselkedésre van szükségünk, a fejezet hátralévő részében kizárólag az `Insert()` metódusra koncentrálunk.

Adatok gyorsítótárazása

Nézzünk meg egy példát. Első lépésként hozzunk létre új ASP.NET-web-alkalmazást CacheState néven, és adjunk hozzá egy `Global.asax` fájlt. A `HttpApplicationState` típus által karbantartott alkalmazásszintű változóhoz hasonlóan a `Cache` bármilyen `System.Object`-leszármazott típust tárolhat, és általában az `Application_Start()` eseménykezelőből töltjük fel adatokkal. A következő példában a `DataSet` objektum tartalmát 15 másodpercenként automatikusan frissítjük. A kérdéses `DataSet` az ADO.NET ismertetése során létrehozott AutoLot adatbázis Inventory táblájának aktuális rekordkészletét tárolja.

A fentiek alapján állítsuk be az `AutoLotDAL.dll` szerelvény (lásd a 22. fejezetet) referenciáját, és módosítsuk a `Global` osztálytípust a következőképpen:

```
<%@ Application Language="C#" %>
<%@ Import Namespace = "AutoLotConnectedLayer" %>
<%@ Import Namespace = "System.Data" %>

<script runat="server">
  // Definiáljunk statikus szintű gyorsítótár-tagváltozót.
  static Cache theCache;

  void Application_Start(Object sender, EventArgs e)
  {
    // Először rendeljük hozzá a statikus 'theCache' változót.
    theCache = Context.Cache;

    // Amikor az alkalmazás elindul, olvassuk be az AutoLot
    // adatbázis Inventory táblájának aktuális rekordjait.
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventory();
    dal.CloseConnection();

    // A DataTable objektumot a gyorsítótárban tároljuk.
    theCache.Insert("AppDataTable",
              theCars, null,
              DateTime.Now.AddSeconds(15),
              Cache.NoSlidingExpiration,
              CacheItemPriority.Default,
              new CacheItemRemovedCallback(UpdateCarInventory));
  }

  // A CacheItemRemovedCallback metódusreferencia célpontja.
  static void UpdateCarInventory(string key, object item,
        CacheItemRemovedReason reason)
  {
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventory();
    dal.CloseConnection();

    // A gyorsítótárban tároljuk.
    theCache.Insert("AppDataTable",
              theCars, null,
              DateTime.Now.AddSeconds(15),
              Cache.NoSlidingExpiration,
              CacheItemPriority.Default,
              new CacheItemRemovedCallback(UpdateCarInventory));
  }
  ...
</script>
```

Először vegyük észre, hogy a `Global` típus definiálta a statikus `Cache` tagváltozót. Ennek az az oka, hogy két olyan megosztott tagot definiáltunk, amelynek hozzá kell férnie a `Cache` objektumhoz. Emlékezzünk rá, hogy a statikus tagoknak nincs hozzáférése az örökölt tagokhoz, ezért nem használhatjuk a `Context` tulajdonságot.

Az `Application_Start()` eseménykezelőben töltsünk fel egy `DataTable` objektumot, és helyezzük el az alkalmazás gyorsítótárában. Sejtéseinknek megfelelően a `Context.Cache.Insert()` metódus többszörösen túlterhelt. A fenti példában valamennyi lehetséges paraméterének értéket adtunk. Tekintsük meg a következő `Add()` hívást, amelyhez megjegyzéseket is fűztünk:

```
theCache.Add("AppDataTable",
             // Név, amellyel a gyorsítótárban elhelyezett
             // elemet azonosítjuk.
  theCars,   // A gyorsítótárban elhelyezni kívánt objektum.
  null,      // Van függőségi kapcsolata az objektumnak?
  DateTime.Now.AddSeconds(15),
             // Az elem meddig legyen a gyorsítótárban?
  Cache.NoSlidingExpiration,
             // Rögzített vagy csúszó lejárati idő?
  CacheItemPriority.Default,
             // A gyorsítótárelem prioritásszintje.
  // A CacheItemRemove esemény metódusreferenciája.
  new CacheItemRemovedCallback(UpdateCarInventory));
```

Az első két paraméter az elem név/érték párját alkotja. A harmadik paraméterrel definiálhatjuk a `CacheDependency` típust (amely jelen esetben `null`, mivel nincs más olyan entitás a gyorsítótárban, amely a `DataTable` objektumtól függene).

Megjegyzés A `CacheDependency` típus meghatározásának lehetősége meglehetősen érdekes. Létrehozhatunk például függőségi viszonyt egy tag és egy külső fájl között. Ha a fájl tartalma megváltozna, a rendszer automatikusan frissíthetné a típust. További részletekért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

A következő három paraméterrel azt határozhatjuk meg, hogy az elem meddig maradhat az alkalmazás gyorsítótárában, illetve megadhatjuk az elem prioritásának szintjét. Meghatároztuk a `Cache.NoSlidingExpiration` írásvédett mező segítségével, amely tájékoztatja a gyorsítótárat, hogy a megadott időkorlát (15 másodperc) abszolút időtartam. Végül, és a példában ez a legfontosabb, létrehoztunk a `CacheItemRemovedCallback` új metódusreferencia-típust, és átadtuk a `DataSet` törlésekor meghívni kívánt metódus nevét.

Amint az UpdateCarInventory() metódus szignatúrájából látszik, a CacheItemRemovedCallback metódusreferencia csak a következő szignatúrának megfelelő metódusokat hívhatja meg:

```
static void UpdateCarInventory(string key, object item,
                               CacheItemRemovedReason reason)
{
}
```

Tehát, amikor az alkalmazás elindul, feltöltjük adatokkal a DataTable objektumot, és elhelyezzük a gyorsítótárban. A DataTable objektumot 15 másodpercenként kiürítjük, frissítjük, és ismét elhelyezzük a gyorsítótárban. Ahhoz, hogy lássuk ennek hatását, létre kell hoznunk egy Page objektumot, amely lehetőséget ad a felhasználói beavatkozásra.

Az *.aspx fájl módosítása

Módosítsuk a kezdeti *.aspx fájl felhasználó felületét a 33.6. ábrának megfelelően.

Az oldal Load eseménykezelőjében állítsuk be a GridView objektumot, hogy a gyorsítótárazott DataTable aktuális tartalmát jelenítse meg, amikor a felhasználó először küld adatokat az oldalra (a kódfájlban ne felejtsük el importálni az AutoLotConnectedLayer névteret):

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    carsGridView.DataSource = (DataTable)Cache["AppDataTable"];
    carsGridView.DataBind();
  }
}
```

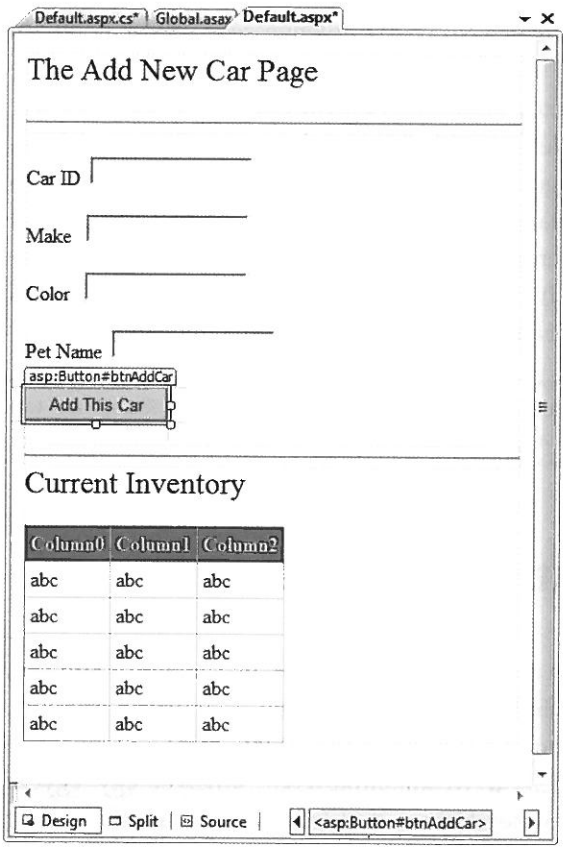
Az „Add This Car" gomb kattintási eseménykezelőjében szúrjuk be új rekordot az AutoLot adatbázisba az InventoryDAL segítségével. Miután a rekordot beszúrtuk, hívjuk a RefreshGrid() segédfüggvényt, amely frissíti a felhasználói felületet:

```
protected void btnAddCar_Click(object sender, EventArgs e)
{
  // Az Inventory tábla frissítése
  // és a RefreshGrid() hívása.
  InventoryDAL dal = new InventoryDAL();
```

```
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    dal.InsertAuto(int.Parse(txtCarID.Text), txtCarColor.Text,
        txtCarMake.Text, txtCarPetName.Text);
    dal.CloseConnection();
    RefreshGrid();
}

private void RefreshGrid()
{
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventory();
    dal.CloseConnection();

    carsGridView.DataSource = theCars;
    carsGridView.DataBind();
}
```



33.6. ábra: *A gyorsítótár-alkalmazás grafikus felülete*

Teszteljük az alkalmazás gyorsítótárának használatát: indítsuk el az aktuális programot (Ctrl + F5), majd a böngészőben megjelenő URL-t másoljuk a vágólapra. Ezt követően indítsuk el az Internet Explorer újabb példányát, és illesszük az URL-t ebbe a böngészőpéldányba. Ekkor a webböngésző két példányával rendelkezünk, amelyek ugyanazt a `Default.aspx` oldalt mutatják azonos tartalommal.

A böngésző egyik példányában vegyünk fel egy új autó bejegyzését. Abban a böngészőben, amelyikből a visszaküldést kezdeményeztük, látható a frissített GridView.

A másik böngészőpéldányban kattintsunk a Refresh (F5) gombra. A példányban még nem látszik az új elem, mivel a `Page_Load` eseménykezelő közvetlenül a gyorsítótárból olvas. (Ha mégis láttuk az értéket, akkor a 15 másodperc már letelt. Vagy gépeljünk gyorsabban, vagy növeljük meg az időtartamot ameddig a `DataTable` a gyorsítótárban marad.) Várjuk néhány másodpercet, majd ismét kattintsunk a második böngészőpéldány `Refresh` gombjára. Most már látnunk kell az új elemet, mivel a `DataTable` érvényessége a gyorsítótárban lejárt, és a `CacheItemRemovedCallback` metódusreferencia célmetódusa automatikusan frissítette a gyorsítótárazott `DataTable` objektumot.

Látható, hogy a `Cache` típus legnagyobb előnye, hogy biztosíthatjuk a lehetőséget a közbelépésre, ha egy tagot a rendszer eltávolít. Ebben a példában elkerülhettük volna a `Cache` használatát, és beállíthattuk volna, hogy a `Page_Load()` eseménykezelő mindig közvetlenül az AutoLot adatbázisból olvassa az adatokat. A célunk azonban az volt, hogy megmutassuk: a gyorsítótár lehetővé teszi az adatok automatikus frissítését a gyorsítótárazási mechanizmus segítségével.

Forráskód A CacheState kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Munkamenetadatok kezelése

Eddig az alkalmazásszintű és a gyorsítótárazott adatok vizsgálatáról volt szó, most pedig vizsgáljuk meg a felhasználónként történő adattárolás szerepét. Ahogy azt már említettük, a *munkamenet* egy kicsivel több, mint a meghatározott felhasználó együttműködése a webalkalmazással, amelyet egy egyedi `HttpSessionState` objektum képvisel. Ha az adott felhasználóállapottal ellátott adatainkat tárolni szeretnénk, a `HttpApplication`-leszármazott típus és bármely

System.Web.UI.Page-leszármazott típus hozzáférhet a Session tulajdonsághoz. A felhasználónkénti adattárolás igényének klasszikus példája az online bevásárlókosár. Ha tíz ember bejelentkezik az online áruházba, minden egyes felhasználó a megvásárolni kívánt termékek egyedi készletét kezeli.

Amikor új felhasználó jelentkezik be a webalkalmazásba, a .NET-futtatókörnyezet automatikusan a felhasználóhoz rendel egy egyedi munkamenet-azonosítót, amely a kérdéses felhasználót egyértelműen azonosítja. A rendszer minden munkamenet-azonosítóhoz a HttpSessionState típus egyedi példányát rendeli, amely felhasználóspecifikus adatokat rögzít. A munkamenetadatok beszúrása és lekérdezési szintaxisa megegyezik az alkalmazásadatok kezelésével:

```
// Az aktuális felhasználóhoz tartozó munkamenet-változók
// hozzáadása és lekérése.
Session["DesiredCarColor"] = "Green";
string color = (string) Session["DesiredCarColor"];
```

A HttpApplication-leszármazott típus lehetővé teszi, hogy kezeljük egy munkamenet indítását és leállítását a Session_Start() és a Session_End() eseménykezelők segítségével. A Session_Start() metódusban szabadon létrehozhatunk tetszőleges felhasználónkénti adatelemet, míg a Session_End() biztosítja azon feladatok végrehajtását, amelyeket a munkamenet lezárásakor kell elvégeznünk:

```
<%@ Application Language="C#" %>
...
void Session_Start(Object sender, EventArgs e)
{
  // Új munkamenet! Előkészülünk, ha szükséges.
}

void Session_End(Object sender, EventArgs e)
{
  // A felhasználó kijelentkezett/túllépte az időkorlátot.
  // Lezárjuk a munkamenetet, ha szükséges.
}
```

A HttpApplicationState típushoz hasonlóan a HttpSessionState tetszőleges típust tárolhat, többek között az egyedi osztályainkat is. Tételezzük fel, hogy van egy új webalkalmazásunk (SessionState), amely a UserShoppingCart nevű osztályt definiálja:

```
public class UserShoppingCart
{
  public string desiredCar;
  public string desiredCarColor;
  public float downPayment;
  public bool isLeasing;
  public DateTime dateOfPickUp;
  public override string ToString()
  {
    return string.Format
           ("Car: {0}<br>Color: {1}<br>$ Down: {2}<br>Lease:
           {3}<br>Pick-up Date: {4}",
           desiredCar, desiredCarColor, downPayment, isLeasing,
           dateOfPickUp.ToShortDateString());
  }
}
```

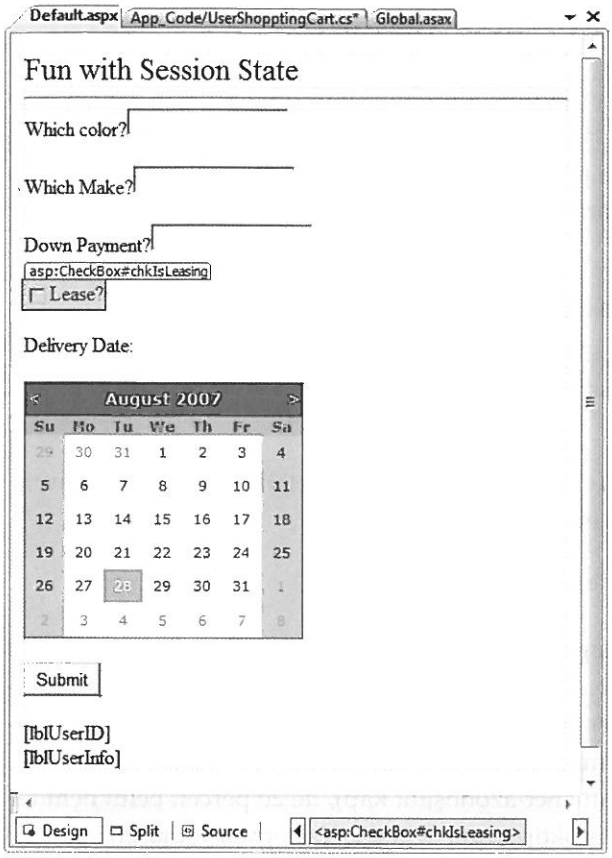
A `Session_Start()` eseménykezelőben minden felhasználóhoz a `UserShoppingCart` osztály újabb példányát rendelhetjük:

```
void Session_Start(Object sender, EventArgs e)
{
  Session["UserShoppingCartInfo"] = new UserShoppingCart();
}
```

Ahogy a felhasználó a különböző oldalak között navigál, minden oldalon lekérhetjük a felhasználó `UserShoppingCart` példányát, és kitölthetjük a mezőket a felhasználóspecifikus adatokkal. Tételezzük fel például, hogy van egy egyszerű `*.aspx` lapunk, amely bemeneti vezérlőelemek készletét definiálja. A vezérlők mindegyike a `UserShoppingCart` típus mezőinek felel meg, `Button`-nal állítjuk be az értékeket, és két `Label`lel jelenítjük meg a felhasználó munkamenet-azonosítóját és a munkamenet-információkat (lásd a 33.7. ábrát).

A gomb kiszolgálóoldali kattintási eseménykezelője magáért beszél (kiszedjük az adatokat a szövegdobozokból, és megjelenítjük a kosár adatait a címkéken):

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
  // Aktuális felhasználói beállítások meghatározása.
  UserShoppingCart cart =
      (UserShoppingCart)Session["UserShoppingCartInfo"];
  cart.dateOfPickUp = myCalendar.SelectedDate;
  cart.desiredCar = txtCarMake.Text;
  cart.desiredCarColor = txtCarColor.Text;
  cart.downPayment = float.Parse(txtDownPayment.Text);
  cart.isLeasing = chkIsLeasing.Checked;
  lblUserInfo.Text = cart.ToString();
  Session["UserShoppingCartInfo"] = cart;
}
```



33.7. ábra: *A munkamenet-alkalmazás grafikus felülete*

A `Session_End()` metódusban a `UserShoppingCart` mezőit rögzíthetjük az adatbázisban vagy valahol (a fejezet végén azonban látjuk majd, hogy az ASP.NET Profile API ezt a feladatot automatikusan végrehajtja). Valamint célszerű a `Session_Error()` megvalósításával elkapni a hibás bemenetet (vagy a `Default.aspx` oldalon különféle ellenőrzési vezérlőelemek alkalmazásával gondoskodhatunk az ilyen a felhasználói hibák kivédéséről).

Mindenesetre, ha két vagy három példányban indítjuk el a választott böngészőt ugyanazzal az URL-lel (másolás/beillesztés műveletekkel, ahogyan azt a gyorsítótár példájában tettük), azt tapasztaljuk, hogy minden felhasználó egyedi kosarat készíthet, amelyek mindegyike a `HttpSessionState` típus egyedi példányára képezhető le.

A HttpSessionState további tagjai

A `HttpSessionState` osztály a típusindexelőn kívül több érdekes taggal rendelkezik. Először is, a `SessionID` tulajdonság az aktuális felhasználó egyedi azonosítóját adja vissza. Ha például szeretnénk megtekinteni a példában automatikusan kiosztott munkamenet-azonosítót, a következőképpen kezeljük az oldal `Load` eseményét:

```
protected void Page_Load(object sender, EventArgs e)
{
  lblUserID.Text = string.Format("Here is your ID: {0}",
                   Session.SessionID);
}
```

A `Remove()` és a `RemoveAll()` metódusokkal törölhetjük a felhasználó `HttpSessionState` példányának elemeit:

```
Session.Remove["SomeItemWeDontNeedAnymore"];
```

A `HttpSessionState` típus tagkészletet definiál, amely az aktuális felhasználói munkamenet lejárati házirendjét szabályozza. Az alapértelmezés szerint minden felhasználói munkamenet `HttpSessionState` objektumát 20 perc inaktivitás után a rendszer megsemmisíti. Így, ha egy felhasználó belép a webalkalmazásba (és egyedi munkamenet-azonosítót kap), de 20 percen belül nem tér vissza a webhelyre, a futtatókörnyezet feltételezi, hogy a felhasználót már nem érdekli az oldal, és az összes munkamenettel kapcsolatos adatot törli. Az alapértelmezett 20 perces lejárati értéket akár felhasználónként szabadon módosíthatjuk a `Timeout` tulajdonság segítségével. Ezt leggyakrabban a `Session_Start()` metódusunk hatókörében lehet megtenni:

```
void Session_Start(Object sender, EventArgs e)
{
  // Minden felhasználói munkamenet 5 perc inaktivitás után
  // megsemmisül.
  Session.Timeout = 5;
  Session["UserShoppingCartInfo"] = new UserShoppingCart();
}
```

Megjegyzés Ha nem kell minden felhasználó `Timeout` értékét módosítani, a `Web.config` fájlban (amelyet a fejezet végén megvizsgálunk) a `<sessionState>` elem `Timeout` attribútumával az összes felhasználó számára módosíthatjuk az alapértelmezett 20 perces értéket.

A `Timeout` tulajdonság előnye az, hogy külön-külön minden felhasználóhoz egyedi időkorlátértéket állíthatunk be. Képzeljük el például, hogy a webalkalmazásunk lehetővé teszi a felhasználók számára, hogy készpénzért tagsági szintet vásároljanak. Az arany fokozatú tagok munkamenete egy órán belül jár le, míg a legalacsonyabb szinten a tagok csak 30 másodpercet kapnak. Felmerül a kérdés, hogy hogyan tárolhatunk felhasználóspecifikus adatokat (mint például az aktuális tagsági szintet) a látogatások között. Az egyik lehetséges válasz a `HttpCookie` típus használata.

Forráskód A SessionState kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A sütikről

A következő állapotkezelési módszer, amelyet megvizsgálunk, a felhasználói adatok *sütikben* történő tárolása, amelyet gyakran a felhasználó gépén egy szövegfájl (vagy több fájl) segítségével valósítunk meg. Amikor a felhasználó bejelentkezik egy meghatározott webhelyre, a böngésző megvizsgálja, hogy a felhasználó számítógépén létezik-e a kérdéses URL-hez sütifájl, és ha igen, hozzáfűzi-e a fájl adatait a HTTP-kéréshez.

A fogadó kiszolgálóoldali weblap beolvassa a süti adatait, és az aktuális felhasználói beállításoknak megfelelő grafikus felületet hoz létre. Biztosan tapasztaltuk már, hogy amikor meglátogatjuk az egyik kedvenc webhelyünket, az valahonnan „tudja", hogy milyen tartalmat szeretnénk megtekinteni. Ennek (részben) a számítógépünkön tárolt süti az oka, amely a meghatározott webhely számára érdekes információkat foglal magában.

Megjegyzés A sütifájlok pontos helye attól függ, hogy milyen böngészőt és milyen operációs rendszert használunk.

Az adott sütifájl tartalma URL-enként változó, de ne feledjük, hogy ezek végül is csak szövegfájlok. Ezért a süti a lehető legrosszabb választás, ha bizalmas információkat kell tárolnunk az aktuális felhasználóról (mint például hitelkártyaszámot, jelszót stb.). Még akkor is, ha időt szakítunk az adatok titkosítására, egy alattomos hacker visszafejtheti az értékeket, és rossz célokra használhatja.

Mindenesetre a sütik fontos szerepet játszanak a webalkalmazások fejlesztésében, úgyhogy nézzük meg, hogy az ASP.NET hogyan dolgozik ezzel az állapotkezelési módszerrel.

Sütik létrehozása

Mindenekelőtt nézzük meg az ASP.NET-sütik két fajtáját: az állandó és az ideiglenes sütiket. Az állandó sütiket tekintjük a sütiadatok hagyományos meghatározásának, mivel a böngésző a név/érték párokat fizikailag a felhasználó merevlemezére menti. Az *ideiglenes* süti (vagy *munkamenetsüti)* ugyanolyan adatokat tartalmaz, mint az állandó süti, de a név/érték párokat a böngésző soha nem menti a merevlemezre; az adatok csak a HTTP-fejlécben találhatók meg. Amikor a felhasználó bezárja a böngészőt, a munkamenetsüti minden adata megsemmisül.

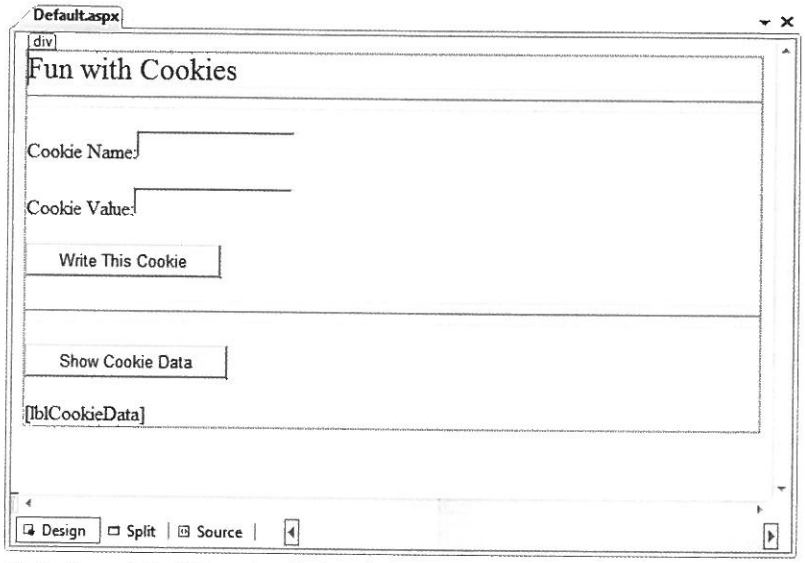
Megjegyzés A legtöbb böngésző legfeljebb 4096 bájtos sütifájlok használatát támogatja. A méretkorlát miatt a sütikben leginkább kis adatmennyiséget, például felhasználói azonosítót célszerű tárolni, amelynek segítségével azonosíthatjuk a felhasználót, és lekérdezhetjük az adatait az adatbázisból.

A sütiadatokat (akár állandó, akár ideiglenes) a kiszolgálóoldalon a `System.Web.HttpCookie` osztálytípus képviseli. Új sütit a `Response.Cookies` tulajdonság segítségével hozhatunk létre. Ha az új `HttpCookie` objektumot beillesztjük a belső gyűjteménybe, a név/érték párokat a HTTP-fejlécben megkapja a böngésző.

A sütik viselkedésének megismerésére hozzunk létre új ASP.NET-webalkalmazást (CookieStateApp néven), és alakítsuk ki a 33.8. ábrán látható felhasználói felületet.

Az első gomb kattintási eseménykezelőjében hozzunk létre új `HttpCookie` objektumot, és adjuk hozzá a `Cookie` gyűjteményhez, amely a `HttpRequest.Cookies` tulajdonsághoz biztosít hozzáférést. Ne feledkezzünk meg arról, hogy az adatokat a rendszer nem tárolja a felhasználó merevlemezén, hacsak explicit módon nem állítjuk be a lejárat idejét a `HttpCookie.Expires` tulajdonság használatával. A következő megvalósítás ideiglenes sütit hoz létre, amelyet a rendszer megsemmisít, amikor a felhasználó leállítja a böngészőt:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
  // Ideiglenes süti létrehozása.
  HttpCookie theCookie =
             new HttpCookie(txtCookieName.Text,
             txtCookieValue.Text);
  Response.Cookies.Add(theCookie);
}
```



33.8. ábra: *A CookieStateApp alkalmazás felhasználói felülete*

A következő metódus állandó sütit hoz létre, amely 2009. március 24-én jár le:

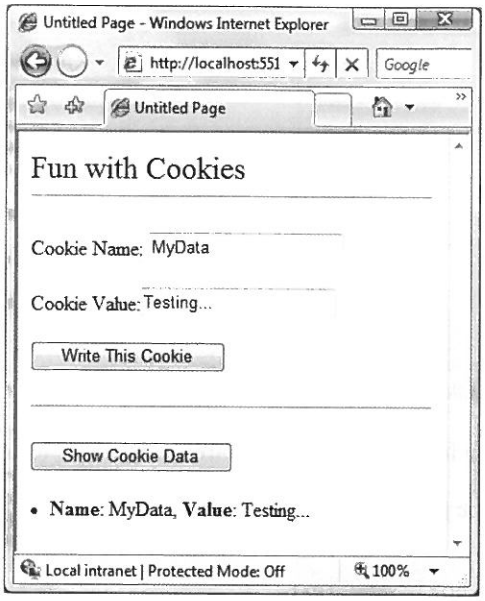
```
protected void btnCookie_Click(object sender, EventArgs e)
{
  HttpCookie theCookie =
             new HttpCookie(txtCookieName.Text,
             txtCookieValue.Text);
  theCookie.Expires = DateTime.Parse("03/24/2009");
  Response.Cookies.Add(theCookie);
}
```

Bemeneti sütik adatainak olvasása

Emlékezzünk rá, a böngésző feladata, hogy az állandó sütikhez hozzáférjen, amikor egy korábban meglátogatott oldalhoz navigálunk. ASP.NET alatt a bemeneti süti adataival a `HttpRequest.Cookies` tulajdonság hozzáférésével dolgozhatunk. Ennek bemutatásához a másik gomb kattintási eseménykezelőjét a következőképpen valósítsuk meg:

```
protected void btnShowCookie_Click(object sender, EventArgs e)
{
  string cookieData = "";
  foreach (string s in Request.Cookies)
  {
    cookieData +=
          string.Format("<li><b>Name</b>: {0}, <b>Value</b>:
          {1}</li>", s, Request.Cookies[s].Value);
  }
  lblCookieData.Text = cookieData;
}
```

Ha most futtatjuk az alkalmazást, és az új gombra kattintunk, azt látjuk, hogy a süti adatait a böngésző valóban elküldte (lásd a 33.9. ábrát).



33.9. ábra: *A süti adatának megtekintése*

Forráskód A CookieStateApp kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A <sessionState> elem szerepe

A fejezet során példák segítségével megvizsgáltuk, hogyan tárolhatunk információkat a felhasználóinkról. Mint láthattuk, a nézetállapotot, az alkalmazás-, a gyorítótár-, a munkamenet- és a sütiadatokat nagyjából egyformán

kezeljük (az indexelő segítségével). Azt is láthattuk, hogy a `HttpApplication` típus segítségével elfoghatunk és reagálhatunk a webalkalmazás élettartama során bekövetkezett eseményekre.

Az alapértelmezés szerint az ASP.NET a munkamenet-állapotot a folyamaton belül (in-proc) tárolja, amelyet az ASP.NET feldolgozó folyamata (`aspnet_wp.exe`) hosztol. A `*.dll` fájlokhoz hasonlóan ennek előnye, hogy az információ hozzáférése a lehető leggyorsabb. Viszont hátránya, hogy ha az alkalmazástartomány bármilyen okból leáll, minden felhasználó állapotadata megsemmisül. Továbbá, ha az állapotadatokat folyamatbeli `*.dll`-ben tároljuk, nem dolgozhatunk együtt hálózati webfarmmal. Rögzítsük az alapértelmezés szerinti viselkedést a `machine.config` fájlunk `<sessionState>` elemében a következőképpen:

```
<sessionState
  mode="InProc"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

Az adattárolás alapértelmezett módja jól használható, ha a webalkalmazást egyetlen webkiszolgáló hosztolja. Gyaníthatjuk, hogy a modell nem ideális webkiszolgáló-farmok esetén, mivel a munkamenet-állapotot az alkalmazástartomány „zárolja".

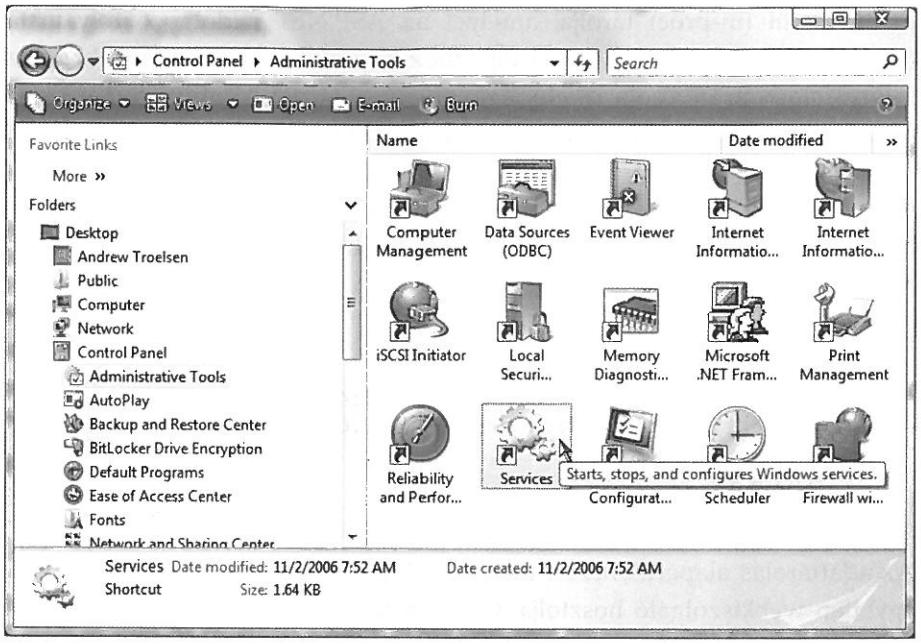
Munkamenetadatok tárolása ASP.NET munkamenet-állapotkiszolgálón

Az ASP.NET alatt utasíthatjuk a futtatókörnyezet, hogy a munkamenet-állapot `*.dll`-t helyettesítő folyamatban, az ASP.NET munkamenet-állapotkiszolgálóban (`aspnet_state.exe`) hosztolja. Ha így teszünk, a `*.dll` fájlt áthelyezhetjük az `aspnet_wp.exe` folyamatból egyedi `*.exe` fájlba, amelyet a webfarm bármely számítógépén tárolhat. Ha az `aspnet_state.exe` folyamatot ugyanazon a számítógépen futtatjuk, mint a webkiszolgálót, kihasználhatjuk az előnyt, hogy az állapotadatokat egyedi folyamatban különítjük el (mivel ez tartósabb megoldás).

A munkamenet-állapotkiszolgáló használatának első lépéseként indítsuk el a célszámítógépen az `aspnet_state.exe` Windows-szolgáltatást. A következőt kell beírnunk a parancssorba:

```
net start aspnet_state
```

Az `aspnet_state.exe` szolgáltatást a 33.10. ábrán látható módon elindíthatjuk a Control Panel Administrative Tools mappájából, a Services kisalkalmazásból.



33.10. ábra: *Az aspnet_state.exe indítása a Services kisalkalmazás segítségével*

A megközelítés alapvető előnye, hogy a Properties ablak segítségével beállíthatjuk, hogy az `aspnet_state.exe` szolgáltatás automatikusan elinduljon, amikor a számítógépet bekapcsoljuk. Ha a munkamenet-állapotkiszolgáló már fut, adjuk a `Web.config` fájlunkhoz a következő `<sessionState>` elemet:

```
<sessionState
  mode="StateServer"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

Itt a `mode` attribútumot `StateServer` értékre állítottuk. Ennyi az egész. Ekkor a CLR a munkamenettel kapcsolatos adatokat az `aspnet_state.exe` folyamatban tárolja. Így, ha a webalkalmazást hosztoló alkalmazástartomány leáll, a munkamenetadatok megmaradnak. Vegyük észre, hogy a `<sessionState>` elem támogathatja a `stateConnectionString` attribútumot. Az alapértelmezett TCP/IP

cím érték (127.0.0.1) a helyi gépre mutat. Ha azt szeretnénk, hogy a .NET-futtatórendszer másik hálózati gépen található `aspnet_state.exe` szolgáltatást használjon (gondoljunk a webfarmokra), az értéket nyugodtan módosíthatjuk.

Munkamenetadatok tárolása dedikált adatbázisban

Végül, ha a legmagasabb szintű elszigetelést és a legnagyobb megbízhatóságot kell a webalkalmazásunk számára biztosítani, utasíthatjuk a futtatókörnyezetet, hogy az összes munkamenetállapot-adatot a Microsoft SQL Server adatbázisban tárolja. A `Web.config` fájlban az alábbi egyszerű módosítást kell elvégezni:

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

Mielőtt elindítanánk a webalkalmazást, győződjünk meg arról, hogy a célszámítógépen (amelyet az `sqlConnectionString` attribútum határoz meg) elvégeztük a megfelelő beállításokat. A .NET Framework 3.5 SDK (vagy a Visual Studio 2008) telepítése során az `InstallSqlState.sql` és az `UninstallSqlState.sql` fájl is a számítógépre kerül, amelyek az alapértelmezés szerint a C:\Windows\Microsoft.NET\Framework\<verzió> könyvtárban találhatók. A célszámítógépen futtatnunk kell az `InstallSqlState.sql` fájlt például a Microsoft SQL Server Management Studio programhoz hasonló eszközzel (amelyet a Microsoft SQL Server 2005 alkalmazással együtt áll a rendelkezésünkre).

Miután az SQL-szkript lefutott, láthatjuk, hogy létrehoztunk egy új SQL Server adatbázist (ASPState), amely az ASP.NET-futtatórendszer által meghívott tárolt eljárásokat, valamint a munkamenetadatokat tároló táblák készletét foglalja magában (valamint a tempdb adatbázis kiegészült néhány táblával, amelyek a cserét segítik majd). Sejthetjük, hogy a leglassabb lehetőség, ha webalkalmazásunkban a munkamenetadatok SQL-adatbázisban történő tárolását állítjuk be. A lehetőség előnye az, hogy a felhasználói adatok rendkívül tartósak (az adatok a webkiszolgáló újraindítása után sem vesznek el).

Megjegyzés Ha ASP.NET munkamenet-állapotkiszolgálón vagy SQL-kiszolgálón tároljuk a munkamenet adatait, minden egyedi típust meg kell jelölnünk a `[Serializable]` attribútummal, amelyet a `HttpSessionState` objektumban szeretnénk tárolni.

Az ASP.NET profil-API-ja

A fejezetben eddig több olyan módszert megvizsgáltunk, amellyel felhasználó- és alkalmazásszintű adatokat tárolhatunk. Ezen módszerek komoly hátránya, hogy az adatokat csak addig őrzik meg, amíg a munkamenet él, vagy a webalkalmazás fut. Sok webhelyen fontos követelmény, hogy a felhasználói adatokat munkamenetek között rögzítsük. Szükség lehet például arra, hogy lehetővé tegyük felhasználói fiókok létrehozását a webhelyen. Előfordulhat, hogy a `ShoppingCart` osztály példányait szeretnénk tárolni munkamenetek között (például online webáruházban). Esetleg a felhasználói felülettel kapcsolatos alapvető beállításokat szeretnénk rögzíteni (témák stb.).

Bár ennek tárolására kialakíthatnánk egy egyedi adatbázist (több tárolt eljárással), de akkor egyedi kódkönyvtárakat kellene fejlesztenünk az adatbázis-objektum kezelésére. Ez nem feltétlenül bonyolult feladat, de a lényeg, hogy ezt az infrastruktúrát akkor is *saját magunknak* kell kifejlesztenünk.

Hogy egyszerűbbé tegye az életünket, az ASP.NET erre a célra kész felhasználóiprofil-kezelő API-t és adatbázisrendszert biztosít számunkra. A szükséges infrastruktúra biztosításán túl a profil-API lehetőséget nyújt arra, hogy a tárolni kívánt adatokat közvetlenül a `Web.config` fájlban definiáljuk (az egyszerűség kedvéért); ennek ellenére bármilyen `[Serializable]` típust rögzíthetünk. Mielőtt nagyon előre szaladnánk, nézzük meg, hogy a profil-API hol tárolja a meghatározott adatokat.

Az ASPNETDB.mdf adatbázis

Minden Visual Studio 2008 segítségével készült ASP.NET-webhelyen a rendszer automatikusan létrehozza az App_Data alkönyvtárat. Alapértelmezés szerint a profil-API (egyéb szolgáltatásokhoz hasonlóan, mint például az ASP.NET szerepkörtagság-API-ja, amelyet a könyvben nem vizsgálunk) az App_Data mappában található helyi `ASPNETDB.mdf` nevű SQL Server 2008 adatbázist használja. Ezt az alapértelmezett viselkedést a helyi számítógépen az aktuális .NET-telepítés `machine.config` fájljának beállítása határozza meg. Ha a programunk olyan ASP.NET-szolgáltatást használ, amelynek az App_Data mappára van szüksége, az `ASPNETDB.mdf` adatfájl menetközben automatikusan létrejön, ha az még nem létezett.

Ha azt szeretnénk, hogy az ASP.NET-futtatókörnyezet másik hálózati számítógépen tárolt ASPNETDB.mdf fájllal kommunikáljon, vagy az adatbázist inkább SQL Server 7.0 (vagy újabb verziójú) adatbázis-kezelőre telepítenénk, az ASPNETDB.mdf fájlt manuálisan kell létrehoznunk az aspnet_regsql.exe parancssori segédprogram segítségével. Mint minden jó parancssori eszköz, az aspnet_regsql.exe is több opcióval rendelkezik, de ha az eszközt argumentumok nélkül futtatjuk, így:

```
aspnet_regsql
```

akkor a grafikus felülettel rendelkező varázsló végigvezet az ASPNETDB.mdf számítógépünkre (és a kívánt verziójú SQL-kiszolgálóra) történő létrehozásának és telepítésének folyamatán.

Ha a webhelyünk nem az adatbázis helyi másolatát használja az App_Data mappában, utolsó lépésként módosítsuk a Web.config fájlt, hogy az ASPNETDB.mdf fájl egyedi helyére mutasson. Tételezzük fel, hogy az ASPNETDB.mdf adatbázist a ProductionServer nevű számítógépen telepítettük. Az alábbi Web.config fájlrészlet (a webhely neve ShoppingCart) segítségével elmagyarázhatjuk a profil-API-nak, hol keresse a szükséges adatbáziselemeket:

```
<configuration>
  <connectionStrings>
    <add name="SqlServices"
         connectionString ="Data Source=ProductionServer;Integrated
         Security=SSPI;Initial Catalog=aspnetdb;"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <profile defaultProvider ="SqlProvider">
      <providers>
        <clear/>
        <add name="AspNetSqlProfileProvider"
             connectionStringName="LocalSqlServer"
             applicationName="ShoppingCart"
             type="System.Web.Profile.SqlProfileProvider,
             System.Web,
             Version=2.0.0.0,
             Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </profile>
  </system.web>
</configuration>
```

A legtöbb *.config fájlhoz hasonlóan ez is sokkal bonyolultabbnak tűnik a valóságos helyzetnél. Gyakorlatilag megadjuk a <connectionString> elemet a szükséges adatokkal együtt, majd az SqlProfileProvider megnevezett példányát (ezt az alapértelmezett szolgáltatót használjuk attól függetlenül, hogy fizikailag hol található az ASPNETDB.mdf fájl). A konfigurációs szintaxis további részleteiért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

Megjegyzés Az egyszerűség kedvéért a példában feltételezzük, hogy a webalkalmazás App_Data alkönyvtárában található automatikusan generált ASPNETDB.mdf adatbázist használjuk.

Felhasználói profil meghatározása a Web.config fájlban

Amint azt már korábban említettük, a felhasználói profilt a Web.config fájlban definiáljuk. Igazán tetszetős ebben a megközelítésben, hogy a profilt az örökölt Profile tulajdonság segítségével erősen típusos módon kezelhetjük. Ennek bemutatására hozzuk létre a FunWithProfiles új webhelyet, és nyissuk meg Web.config fájlját szerkesztésre.

Az a célunk, hogy létrehozzunk egy profilt, amely a bejelentkezett felhasználók otthoni címét és az általuk kezdeményezett adatküldések számát tartja nyilván. Nem meglepő, hogy a profil adatait a <profile> elemben kell megadni név/érték típus párok formájában. Nézzük a következő profilt, amelyet a <system.web> elem hatókörében hoztunk létre:

```
<profile>
  <properties>
    <add name="StreetAddress" type="System.String" />
    <add name="City" type="System.String" />
    <add name="State" type="System.String" />
    <add name="TotalPost" type="System.Int32" />
  </properties>
</profile>
```

Itt meghatároztunk egy nevet és CLR-adattípust a profil minden elemének (természetesen további elemeket vehetnénk fel az irányítószám, név stb. tárolására, de a példa így is érthető). Valójában a type attribútum opcionális; az alapértelmezett típus a System.String. Elvárásainknak megfelelően több attribútumot megadhatunk a profilbejegyzésben, amelyekkel tovább finomíthatjuk, hogy az ASPNETDB.mdf hogyan rögzítse az információkat. A 33.4. táblázat bemutat néhány alapvető attribútumot.

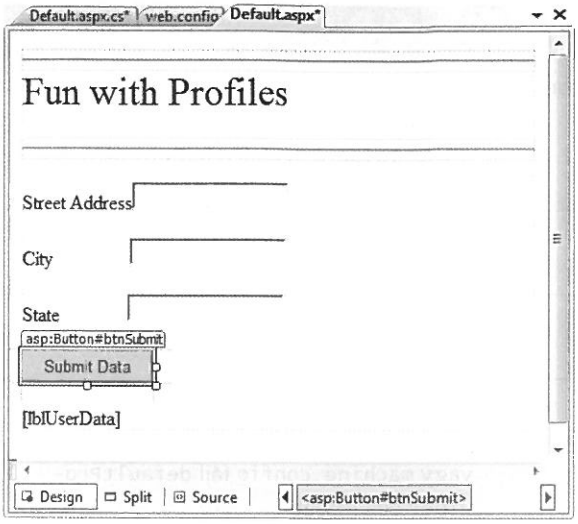
Attribútum	Értéke	Jelentés
`allowAnonymous`	True | False	Ez az attribútum korlátozza vagy engedélyezi az érték névtelen hozzáférését. Ha az értéke `false`, a névtelen felhasználók nem férhetnek hozzá az adott profilértékhez.
`defaultValue`	String	Ez az attribútum megadja a visszatérési értéket, ha a tulajdonságot nem állítottuk be.
`name`	String	Ez az attribútum a tulajdonság egyedi azonosítója.
`provider`	String	Ez az attribútum az érték kezeléséhez használt szolgáltató, felülbírálja a `Web.config` vagy `machine.config` fájl `defaultProvider` beállítását.
`readOnly`	True | False	Ez az attribútum korlátozza vagy engedélyezi az írási hozzáférést.
`serializeAs`	String | XML | Binary	Ez az attribútum az adattárban rögzített értékek formátuma.
`type`	Primitív | Egyedi típus	Ez az attribútum .NET-primitív-típus vagy -osztály. Az osztály teljesen meghatározott nevét kell megadni (pl. `MyApp.UserData.ColorPrefs`).

33.4. táblázat: *A profiladatok legfontosabb attribútumai*

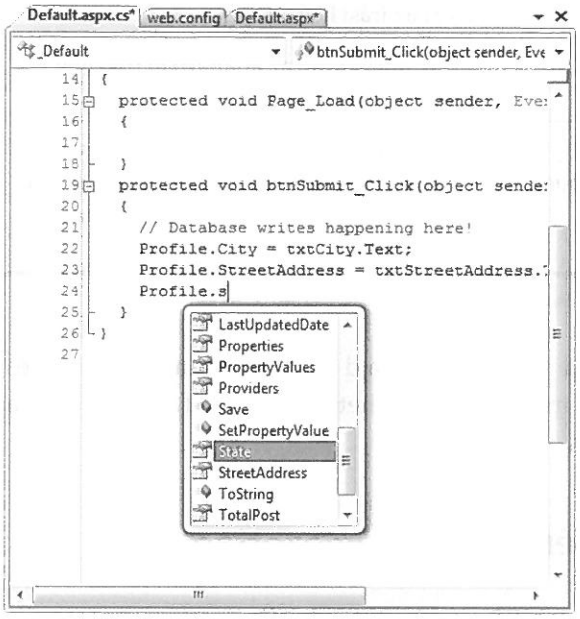
Az aktuális profil módosítása során látjuk majd az attribútumokat működés közben. Addig is tekintsük meg, hogyan érhetjük el ezeket az adatokat a weblapokról programozottan.

Profiladatok hozzáférése programozottan

Emlékezzünk rá, hogy az ASP.NET-profil egyetlen célja, hogy automatizálja az adatok dedikált adatbázisba írásának (és az adatok olvasásának) folyamatát. Ennek kipróbálására módosítsuk a `Default.aspx` fájl felhasználói felületét: adjunk hozzá néhány szövegdobozt (magyarázó címkékkel együtt), amelyekben a felhasználó címét (utca, város, ország) kérjük be. Valamint adjunk hozzá egy gombot (btnSubmit néven) és egy utolsó címkét (lblUserData néven), amely a 33.11. ábrán látható módon a rögzített adatokat jeleníti meg.



33.11. ábra: *A FunWithProfiles Default.aspx lapjának felhasználói felülete*



33.12. ábra: *A profiladatok erősen típusosak*

A gomb kattintási eseménykezelőjében az örökölt `Profile` tulajdonság segítségével mentsük el a profilinformációkat, amelyeket a felhasználó a kapcsolódó szövegdobozban meghatározott. Mint azt a 33.12. ábrán láthatjuk, a Visual Studio 2008 minden profiladatot erősen típusos tulajdonságként biztosít. Valójában a `web.config` fájl segítségével definiáltuk az egyedi struktúrát.