Az objektum (vagy másnéven logikai) erőforrások szerepe

A WPF-erőforrásokat akkor tudjuk a leghatékonyabban használni, ha az egyedi .NET-objektumokat ágyazunk a szerelvényekbe, hogy alkalmazásunkban használhassuk őket. Első pillantásra ez elég furcsa ötletnek tűnhet. Azonban így a programunk több területén közösen használható grafikus elemeket (ecseteket, tollakat stb.) definiálhatunk. Ezt a módszert leginkább akkor használhatjuk, amikor WPF-alkalmazásaink számára egyedi témákat és stílusokat készítünk. A következőkben ezt a témakört vizsgáljuk meg részletesen.

Stílusok készítése és alkalmazása WPF-vezérlőelemeken

A WPF-alkalmazás felhasználói felületének létrehozása során a vezérlőelemek gyakran közös megjelenéssel rendelkeznek. Gondoskodhatunk arról, hogy a gombok magassága, szélessége, háttérszíne és sztringtartalmuk betűmérete egységes legyen. Noha megtehetjük, hogy az egyes gombok tulajdonságait megegyező értékre állítjuk, ez a megközelítés menetközben megnehezíti a módosítások megvalósítását, mivel változtatások esetén ugyanazokat a tulajdonságokat kell több objektumon alaphelyzetbe állítanunk.

Szerencsére a WPF több módszert biztosít, amelyek segítségével minimális erőfeszítések árán módosíthatjuk a felhasználói felület elemeinek (stílusok, sablonok, arculatok stb.) megjelenését. Látjuk majd, hogy a stílusok építése érinti a fejezetben eddig tárgyalt témaköröket (kétdimenziós grafikákat, animációkat és erőforrásokat). Bemelegítésként kezdjük a stílusok alkalmazásának vizsgálatával.

Az inline stílusok használata

A WPF-vezérlők megjelenését és működését a *stílusok* segítségével módosíthatjuk. A stílus a webalapú stíluslapok rokona, mivel nem rendelkezik saját felhasználói felülettel, csak bevezet néhány olyan tulajdonságot, amelyet más felhasználóifelület-elemek alkalmazhatnak. A `Control` osztály leszármazottai – közöttük természetesen a `Window` típus is – támogatják a stílusokat (a `Style`

tulajdonság révén). Ha szeretnénk stílust létrehozni, ennek egyik módja, hogy a tulajdonság-elem szintaxist alkalmazzuk, amely lehetővé teszi a stílus „inline" hozzárendelését.

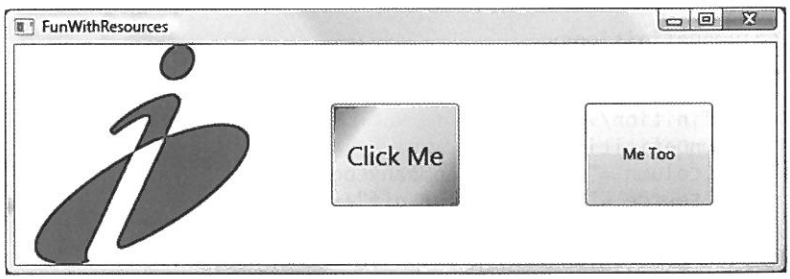
Ennek bemutatására módosítsuk a FunWithResources projekt <Grid> elemét, és definiáljunk egy-egy gombot a fennmaradó két cellában. Tanulmányozzuk a következő markupot, amely egyedi stílust hoz létre a btnClickMe gombhoz (de a második btnClickMeToo gombra nem vonatkozik):

```
<Window x:Class="FunWithResources.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FunWithResources" Height="207" Width="612"
  WindowStartupLocation="CenterScreen">
  <Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Image Grid.Column ="0" Name ="companyLogo"
              Source ="IntertechBlue.gif"/>

  <!-- A gomb inline stílussal rendelkezik! -->
  <Button Grid.Column ="1" Name="btnClickMe" Height="80"
               Width = "100" Content ="Click Me">
    <Button.Style>
      <Style>
        <Setter Property ="Button.FontSize" Value ="20"/>
        <Setter Property ="Button.Background">
          <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
              <GradientStop Color="Green" Offset="0" />
              <GradientStop Color="Yellow" Offset="0.25" />
              <GradientStop Color="Pink" Offset="0.75" />
              <GradientStop Color ="Red" Offset="1" />
            </LinearGradientBrush>
          </Setter.Value>
        </Setter>
      </Style>
    </Button.Style>
  </Button>

  <!-- Erre a gombra nem érvényes a stílus! -->
  <Button Grid.Column ="2" Name="btnClickMeToo"
              Height="80" Width = "100" Content ="Me Too"/>
  </Grid>
</Window>
```

Mint látjuk, a WPF-stílust a `<Style>` elem segítségével definiálhatjuk. Ebben a hatókörben tetszőleges számú `<Setter>` elemet definiálhatunk, amelyek segítségével bevezethetjük a beállítani kívánt tulajdonságok név/érték párjait. A példában a gomb `FontSize` tulajdonságának értéke 20, a `Background` tulajdonságát pedig a `LinearGradientBrush` típussal állítjuk be, amelyet négy összekapcsolt szín alkot.

Megjegyzés Ha szükséges, programozott módon létrehozhatunk stílust a kódfájlunkban. Egyszerűen állítsuk be a Style tulajdonságot a vezérlőelem-leszármazott típuson.

Noha a stílusok készítésének ezen módja szintaktikai szempontból helyes, az inline stílusok egyik nyilvánvaló korlátozása, hogy a felhasználóifelület-típus meghatározott példányához (példánkban a btnClickMe gombhoz) kapcsolódnak, nem pedig a hatókörben található gombokhoz. Vegyük észre a 30.13. ábrán, hogy a második gombra, a btnClickMeToo-ra a btnClickMe gombhoz rendelt stílus nem vonatkozik.



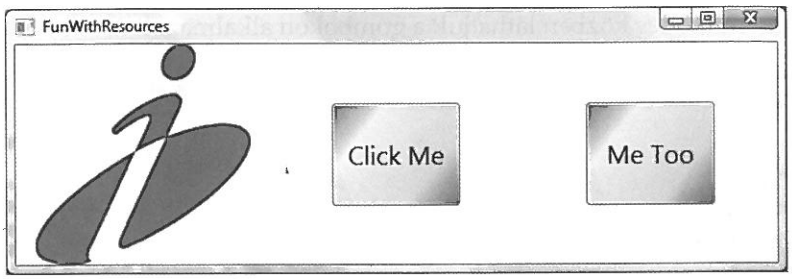
30.13. ábra: *Az inline stílusok ahhoz a vezérlőelemhez kapcsolódnak, amelyek definiálták őket*

A megnevezett stílusok használata

Ha olyan stílust szeretnénk készíteni, amelyet ugyanazon típus több felhasználóifelület-eleme alkalmazhat (például az összes `Button`, az összes `ListBox` típus), a stílust létrehozhatjuk a tároló *erőforrásszótárában*, ilyenformán objektum (vagy másnéven logikai) erőforrást definiálva. Például megnevezett stílussal bővíthetjük a `<Window>` erőforrásszótárát, és a `Key` tulajdonságban meghatározott névvel azonosíthatjuk azt. Így a XAML-dokumentumban mindenhol ugyanarra a témára hivatkozhatunk. Nézzük meg a következő módosítást:

```
<Window x:Class="FunWithResources.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FunWithResources" Height="207"
  Width="612" WindowStartupLocation="CenterScreen">

  <!-- Logikai erőforrás hozzárendelése az ablak
       erőforrásszótárához -->
  <Window.Resources>
    <Style x:Key ="MyFunkyStyle">
      <Setter Property ="Button.FontSize" Value ="20"/>
      <Setter Property ="Button.Background">
        <Setter.Value>
          <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
            <GradientStop Color="Green" Offset="0" />
            <GradientStop Color="Yellow" Offset="0.25" />
            <GradientStop Color="Pink" Offset="0.75" />
            <GradientStop Color ="Red" Offset="1" />
          </LinearGradientBrush>
        </Setter.Value>
      </Setter>
    </Style>
  </Window.Resources>
  <Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Image Grid.Column ="0" Name ="companyLogo"
              Source ="IntertechBlue.gif"/>

  <!-- Mindkét gomb ugyanazzal a stílussal rendelkezik -->
  <Button Grid.Column ="1" Name="btnClickMe" Height="80"
          Width = "100" Style ="{StaticResource MyFunkyStyle}"
          Content = "Click Me"/>
  <Button Grid.Column ="2" Name="btnClickMeToo" Height="80"
          Width = "100" Style ="{StaticResource MyFunkyStyle}"
          Content = "Me Too"/>
  </Grid>
</Window>
```

Ezúttal vegyük észre, hogy a stílust a `<Window.Resources>` elem hatókörében definiáltuk, és a Key attribútum segítségével a stílusnak a MyFunkyStyle nevet adtuk. Ettől eltekintve maga a stílusdeklaráció megegyezik az előbbiekben létrehozott inline stílussal. Vegyük észre azt is, hogy ha alkalmazni szeretnénk a stílust (mint a `<Button>` definíciók esetén), akkor a `StaticResource` markupbővítőt hívhatjuk segítségül (lásd a 28. fejezetet). A módosítást követően - a 30.14. ábrán látható módon - minden gomb ugyanazzal a megjelenéssel rendelkezik.



30.14. ábra: *A megnevezett stílusokat az adott hatókörben több felhasználóifelület-elem alkalmazhatja*

Stílusbeállítások felülbírálása

Fontos rámutatnunk arra, hogy ha a felhasználói felület eleme meghatározott stílust alkalmaz (legyen az inline stílus vagy megnevezett stílus), az elemnek módjában áll „felülbírálni" a tulajdonságok beállításait. Tételezzük fel például, hogy szeretnénk a második gombon használni a `MyFunkyStyle` stílus `Background` beállítását, de kisebb betűméretet alkalmaznánk. Hogy ezt megvalósítsuk, egyszerűen rendeljünk új értéket a módosítani kívánt tulajdonsághoz a nyitó XAML-elemben:

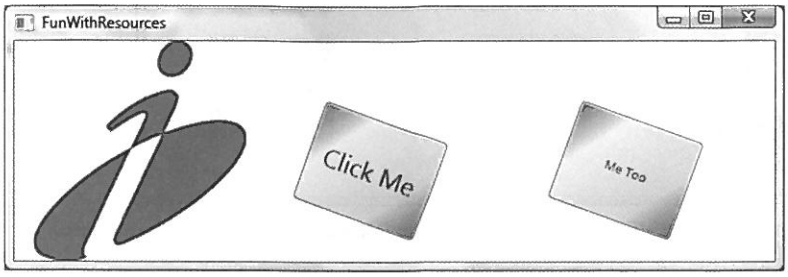
```
<Button Grid.Column ="2" Name="btnClickMeToo" Height="80"
        Width = "100" Style ="{StaticResource MyFunkyStyle}"
        FontSize = "10" Content = "Me Too"/>
```

Létező stílusok leszármaztatása

Létező stílusok alapján létrehozhatunk új stílusokat a `BasedOn` tulajdonság révén, ha a stílust, amelyet bővíteni szeretnénk, a Key tulajdonság segítségével elláttuk a megfelelő névvel. Például a következő `NewFunkyStyle` stílus (amelyet a `<Window.Resources>` hatókör gyermekelemeként adtunk a kódhoz) alkalmazza a `MyFunkyStyle` stílus betűméretét és háttérszínét, de 20 fokkal elforgatja a felhasználói felület elemet:

```
<Style x:Key ="NewFunkyStyle"
         BasedOn = "{StaticResource MyFunkyStyle}">
  <Setter Property = "Button.Foreground" Value = "Blue"/>
  <Setter Property = "Button.RenderTransform">
    <Setter.Value>
      <RotateTransform Angle = "20"/>
    </Setter.Value>
  </Setter>
</Style>
```

A 30.15. ábrán működés közben láthatjuk a gombokon alkalmazott új stílust.



30.15. ábra: *A leszármazott stílus alkalmazása*

A stílusok kiterjesztése

A stílus erőforrásszótárba helyezése mindenképpen megfelelő lépés. Mi történik azonban akkor, ha több felhasználóifelület-elemen szeretnénk ugyanazt a stílust alkalmazni? Pillanatnyilag a `MyFunkyStyle` stílus csak gombokon alkalmazható, mivel a stílus a tulajdonság–elem szintaxis segítségével explicit módon hivatkozik a `Button` típusra.

A WPF-stílusok egyik roppant érdekes jellemzője, hogy a `<Setter>` hatókörben hozzárendelt értékek tiszteletben tartják a származtatás fogalmát. Így, ha a stílusunkban a felhasználóifelület-elemek közös szülőtípusán (`System.Windows.Controls.Control`) állítunk be tulajdonságokat, hatékony módon definiálhatunk közös stílust az összes WPF-vezérlőelem számára, például a következő stílusmódosítás:

```
<Window.Resources>
  <Style x:Key ="MyFunkyStyle">
    <Setter Property ="Control.FontSize" Value ="20"/>
    <Setter Property ="Control.Background">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
          <GradientStop Color="Green" Offset="0" />
          <GradientStop Color="Yellow" Offset="0.25" />
          <GradientStop Color="Pink" Offset="0.75" />
          <GradientStop Color ="Red" Offset="1" />
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
  ...
</Window.Resources>
```

lehetővé teszi számunkra, hogy a `MyFunkyStyle` stílust `TextBox` típusokon, illetve `Button` típusokon (vagy a `Control` osztályból származó bármely elemen) alkalmazzuk. Tételezzük fel, hogy a következő, új felhasználóifelület-elemet adtuk az aktuális `<Grid>` elem új oszlopához:

```
<TextBox Grid.Column ="3" Name="txtAndMe" Height="40" Width = "100"
         Style ="{StaticResource MyFunkyStyle}" Text = "And me!"/>
```

Megjegyzés Ha olyan stílust készítünk, amely ősosztálytípust alkalmaz, nem kell aggódnunk, hogy a leszármazott típusok által nem támogatott értéket rendelünk a függőségi tulajdonsághoz. Ha a leszármazott típus nem támogatja az adott függőségi tulajdonságot, a rendszer figyelmen kívül hagyja.

A stílusok korlátozása

Ha olyan stílust szeretnénk definiálni, amelyet *kizárólag* a felhasználóifelület-elemek bizonyos típusain (például csak `Button` típusokon, semmi máson) lehet alkalmazni, a stílus nyitó elemében beállíthatjuk a `TargetType` tulajdonságot. A tulajdonságnak az adott vezérlőelem metaadat-leírását kell meghatározni, tehát az x:Type markupbővítőt kell használnunk (lásd a 28. fejezetet). Ennek bemutatására a következőképpen módosíthatjuk a `MyFunkyStyle` stílust (a módosítással markuphibát kapunk, ha az előző `TextBox` típuson próbáljuk alkalmazni ezt a stílust):

```
<Style x:Key ="MyFunkyStyle" TargetType = "{x:Type Button}">
  ...
</Style>
```

Stílusok hozzárendelése implicit módon

A WPF-stílusokat meghatározott XAML-hatókörben implicit módon is beállíthatjuk a felhasználói felület vezérlőin. Megnevezett stílusok készítése során a Key tulajdonság hozzárendelése gyakorlatilag opcionális, *ha* a `Target` tulajdonság segítségével korlátoztuk a stílus alkalmazását:

```
<Style TargetType = "{x:Type Button}">
  ...
</Style>
```

Ily módon, a hatókörben az összes gomb implicit módon alkalmazza a MyFunkyStyle stílust még akkor is, ha nem használják a StaticResource jelölő kiterjesztést:

```
<Button Grid.Column ="1" Name="btnClickMe"
            Height="80" Width = "100" Content = "Click Me"/>

<Button Grid.Column ="2" Name="btnClickMeToo"
            Height="80" Width = "100" Content = "Me Too"/>
```

Ne feledjük, hogy amikor a TargetType tulajdonsággal olyan stílust definiálunk, amelyben *nem* állítottuk be a Key értéket, a rendszer a stílust kizárólag a megegyező névvel rendelkező típusokon alkalmazza. Ezért, ha az aktuális stílust a következőképpen módosítanánk:

```
<Style TargetType = "{x:Type Control}">
  ...
</Style>
```

sem a Button, sem a TextBox típus nem alkalmazná a stílust! Ez annak köszönhető, hogy a stílusunk ezen iterációja a Control ősosztályt célozza meg. Soha ne feledjük, hogy az osztályt vagy valamely leszármazottat képviselő stílus fogalma csak a megnevezett stílusok esetén alkalmazható.

Forráskód A FunWithResources kódfájlokat a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Stílusok definiálása triggerekkel

A WPF-stílusok következő jellemzője, amelyet megvizsgálunk, a *triggerek* fogalma. A triggerek segítségével <Setter> elemeket definiálhatunk, amelyeket a rendszer kizárólag meghatározott feltétel teljesülése esetén alkalmaz. Például növelhetjük a betűméretet, ha az egeret a gomb fölé húzzuk. Vagy egy meghatározott színnel kiemelhetjük a fókuszban lévő szövegdobozt. A triggerek rendkívül hasznosak lehetnek ilyen helyzetekben, mivel ha egy tulajdonság módosul, lehetővé teszik meghatározott műveletek végrehajtását, és ehhez nem szükséges C#-forráskódot készítenünk mögöttes kódfájlban.

A következő XAML-markup három szövegdobozt definiál, amelyek mindegyikénél a `Style` tulajdonság értéke a `TextBoxStyle` stílus. A szövegdobozok megjelenése (magasság, szélesség stb.) megegyezik, csak annak a szövegdoboznak a háttere sárga színű, amely az adott pillanatban fókuszban van.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Window.Resources>
    <Style x:Key ="TextBoxStyle" TargetType = "{x:Type TextBox}">
      <Setter Property = "Foreground" Value = "Black"/>
      <Setter Property = "Background" Value = "LightGray"/>
      <Setter Property = "Height" Value = "30"/>
      <Setter Property = "Width" Value = "100"/>

      <!-- A következő <setter> elem csak akkor érvényesül,
           ha a szövegdoboz az alkalmazás fókuszában van -->
      <Style.Triggers>
        <Trigger Property = "IsFocused" Value = "True">
          <Setter Property = "Background" Value = "Yellow"/>
        </Trigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>

  <StackPanel>
    <TextBox Name = "txtOne"
             Style = "{StaticResource TextBoxStyle}" />
    <TextBox Name = "txtTwo"
             Style = "{StaticResource TextBoxStyle}" />
    <TextBox Name = "txtThree"
             Style = "{StaticResource TextBoxStyle}" />
  </StackPanel>
</Window>
```

Ha az előző XAML-kódot beírjuk a SimpleXamlPad.exe alkalmazásba, láthatjuk, hogy amikor a szövegdobozok között lépegetünk, az éppen kiválasztott vezérlőelem háttere sárga színű, míg a többiek az alapértelmezés szerinti szürke színű háttérrel rendelkeznek. A triggerek intelligens elemek, hiszen ha a trigger feltétele *nem teljesül*, a rendszer automatikusan az alapértelmezés szerinti értéket rendeli a vezérlőhöz. Ezért, mihelyst a szövegdoboz kikerül az alkalmazás fókuszából, automatikusan az alapértelmezett színnel jelenik meg anélkül, hogy bármit tennünk kellene.

A triggereket úgy is megtervezhetjük, hogy a definiált <Setter> elemeket a rendszer *több feltétel* teljesülése esetén alkalmazza (hasonlít a több feltétellel működő if utasítás programozásához). Tételezzük fel, hogy egy szövegdoboz hátterét csak akkor szeretnénk sárga színnel megjeleníteni, ha a doboz fókuszban van, és az egérmutató a doboz határain belül helyezkedik el. Ennek megvalósításához a feltételeket a <MultiTrigger> elem segítségével definiálhatjuk:

```
<Window.Resources>
  <Style x:Key ="TextBoxStyle" TargetType = "{x:Type TextBox}">
    <Setter Property = "Foreground" Value = "Black"/>
    <Setter Property = "Background" Value = "LightGray"/>
    <Setter Property = "Height" Value = "30"/>
    <Setter Property = "Width" Value = "100"/>
    <!-- A következő <Setter> elemet csak akkor alkalmazza
         a rendszer, ha a szövegdoboz az alkalmazás fókuszában van,
         és az egeret a doboz fölé húzzuk -->
      <Style.Triggers>
        <MultiTrigger>
        <MultiTrigger.Conditions>
          <Condition Property = "IsFocused" Value = "True"/>
          <Condition Property = "IsMouseOver" Value = "True"/>
        </MultiTrigger.Conditions>
        <Setter Property = "Background" Value = "Yellow"/>
      </MultiTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

Forráskód A StyleWithTriggers.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Stílusok hozzárendelése programozott módon

A stílusok vizsgálatának befejezéseként készítsünk egyszerű alkalmazást, amely bemutatja, hogyan rendelhetünk a forráskódban a felhasználóifelület-elemekhez stílusokat az új StylesAtRuntime nevű Visual Studio WPF Windows alkalmazás projekt segítségével.

Célunk, hogy egy gomb számára a <Window> elem erőforrásszótárában három különböző stílust definiáljunk. Az első, a TiltStyle stílus 10 fokkal elforgatja a gombot. A második, a GreenStyle stílus előre meghatározott értékre állítja a Background, a Foreground és a FontSize tulajdonságokat. Az utolsó, a MouseOverStyle stílus a GreenStyle stíluson alapul, de triggerfeltételt ad a kódhoz, amely megnöveli a gomb betűméretét és szövegét. Íme a stílusok XAML-leírásai:

```
<Window.Resources>
  <!-- A stílus 10 fokos szögben megdönti a gombokat -->
  <Style x:Key ="TiltStyle" TargetType = "{x:Type Button}">
    <Setter Property = "RenderTransform">
      <Setter.Value>
        <RotateTransform Angle = "10"/>
      </Setter.Value>
    </Setter>
  </Style>

  <!-- A stílus tavaszias színvilágot kölcsönöz a gomboknak -->
  <Style x:Key ="GreenStyle" TargetType = "{x:Type Button}">
    <Setter Property = "Background" Value ="Green"/>
    <Setter Property = "Foreground" Value ="Yellow"/>
    <Setter Property ="FontSize" Value ="15" />
  </Style>

  <!-- A stílus megnöveli a gomb méretét, amikor az egeret fölé
       húzzuk -->
  <Style x:Key ="MouseOverStyle"
         BasedOn ="{StaticResource GreenStyle}"
         TargetType = "{x:Type Button}">
    <Style.Triggers>
      <Trigger Property ="IsMouseOver" Value ="True">
        <Setter Property ="FontSize" Value ="20" />
        <Setter Property ="Foreground" Value ="Black" />
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

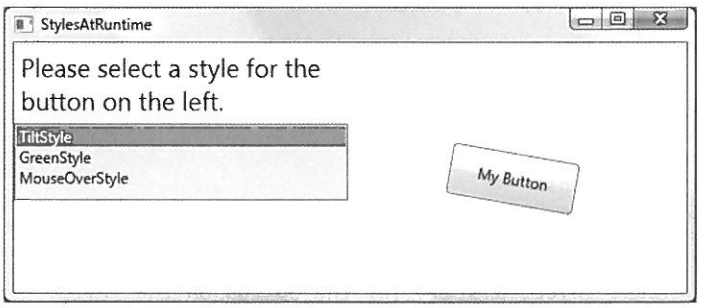
A `Window` objektum tartalmaz egy olyan `Grid` objektumot, amely a `TextBlock`, a `ListBox` és a `Button` típusok helyét rendezi el. A `ListBox` tartalmazza a témák nevét, és kezeli a `SelectionChanged` eseményt. Íme, a felhasználói felület kapcsolódó XAML-kódja:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <StackPanel Grid.Column="0">
    <TextBlock TextWrapping ="Wrap" FontSize ="20"
               Padding="5,5,5,5">
               Please select a style for the button on the left.
    </TextBlock>
    <ListBox Name ="lstStyles" Height ="60" Background = "Yellow"
             SelectionChanged ="comboStyles_Changed" />
  </StackPanel>
  <Button Name="btnMouseOverStyle" Grid.Column="1"
          Height="40" Width="100">My Button</Button>
</Grid>
```

Az utolsó feladat, hogy kitöltsük a ListBox-ot, és a kapcsolódó kódfájlban kezeljük a `SelectionChanged` eseményt. Tanulmányozzuk, hogy a következő forráskódban hogyan nyerjük ki név szerint az aktuális erőforrást az örökölt `FindResource()` metódus segítségével:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();

    // Elemek hozzáadása a listamezőhöz.
    lstStyles.Items.Add("TiltStyle");
    lstStyles.Items.Add("GreenStyle");
    lstStyles.Items.Add("MouseOverStyle");
  }

  protected void comboStyles_Changed(object sender,
                                     RoutedEventArgs args)
  {
    // Kiválasztott stílusnév beolvasása a listamezőből.
    System.Windows.Style currStyle = (System.Windows.Style)
                         FindResource(lstStyles.SelectedValue);

    // A gomb típus stílusának beállítása.
    this.btnMouseOverStyle.Style = currStyle;
  }
}
```

Ezt követően lefordíthatjuk az alkalmazást. Ha az egyes listaelemekre kattintunk, a gomb megjelenése a stílusnak megfelelően változik (lásd a 30.16. ábrát).



30.16. ábra: *Stílusok beállítása programozott módon*

Forráskód A StylesAtRuntime kódfájlokat a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Vezérlőelem felhasználói felületének módosítása sablonok segítségével

A stílusok segítségével hatékony (és egyszerű) módon megváltoztathatjuk a WPF-vezérlőelemek alapvető megjelenését, ha definiáljuk az értékek alapértelmezett készletét a vezérlőelemek tulajdonságkészlete számára. Noha a stílusok segítségével módosíthatjuk a felhasználói felület különféle beállításait, a vezérlőelem általános megjelenése érintetlen marad. Függetlenül attól, hogyan alakítjuk a `Button` típus megjelenését a különböző tulajdonságok segítségével, alapvetően ugyanaz a téglalap alakú vezérlő áll rendelkezésünkre, amelyet évek óta ismerünk. Mi történik akkor, ha szeretnénk teljesen megváltoztatni (és hatszögletű háromdimenziós képre cserélni) a `Button` típus megjelenését, de az objektumnak továbbra is `Button` típusként kellene viselkednie? Mi lenne, ha használni szeretnénk a WPF folyamatjelző funkcionalitását, de a készültségi fok százalékos arányát kördiagrammal jelenítenénk meg a felhasználói felületen? Ahelyett, hogy manuálisan egyedi vezérlőelemet kellene készítenünk (ahogyan más grafikus felület eszközrendszer esetén tennénk), erre a célra a WPF *vezérlőelem-sablonokat* biztosít.

A sablonok egyértelműen elkülönítik a vezérlőelem *felhasználói felületét* (azaz a megjelenését) a vezérlőelem *viselkedésétől* (azaz az eseménykészletétől és a metódusaitól). A sablonok segítségével igényeink szerint szabadon módosíthatjuk a WPF-vezérlők kimenetének megjelenítését. Programozási szempontból a vezérlőelem-sablonokat a `ControlTemplate` ősosztály képviseli, amelyet a XAML-kódban a megegyező névvel rendelkező `<ControlTemplate>` elemmel írhatunk le. Miután létrehoztuk a sablont, WPF-oldalakhoz, -ablakokhoz vagy -vezérlőelemekhez kapcsolhatjuk a `Template` tulajdonság segítségével.

A vezérlőelem-sablon készítésének egyik érdekes szempontja, hogy a `<ContentPresenter>` elem révén a sablonban teljhatalommal rendelkezünk a vezérlő tartalmának pozícionálása felett. Az elem segítségével meghatározhatjuk a tartalom helyét és felhasználói felületét az adott vezérlőelem-sablon számára. Továbbá, ha a sablonban nem definiáljuk a `<ContentPresenter>` elemet, akkor a sablont alkalmazó vezérlőelem nem jelenít meg *semmilyen* tartalmat, még akkor sem, ha definiálja azt:

```
<!-- Ha az alkalmazott sablon nem rendelkezik <ContentPresenter>
     elemmel, nem jeleníti meg az "OK" feliratot -->
<Button Name ="myButton"
        Template ="{StaticResource roundButtonTemplate}">
  Click!
</Button>
```

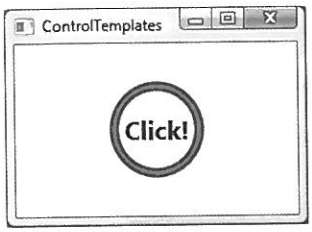
Ha ezen a ponton vezérlőelem-sablont definiálunk, nem meglepő, ha úgy érezzük, a munka hasonlít a stílusok elkészítésének folyamatához. Például a sablonokat is rendszerint erőforrásszótárban tároljuk, a sablonok is támogatják a triggereket, stb. Mindezek ismeretében már jól felkészültünk a sablonok létrehozására. Vizsgáljunk meg néhány sablont működés közben a `ControlTemplates` új WPF Windows alkalmazás projekt segítségével.

Egyedi sablon készítése

Vizsgáljuk meg a következő egyszerű sablont, amely a `<Grid>` elemben két `<Ellipse>` típussal definiál egy kerek gombot. A tartalom a vezérlőelem kellős közepén jelenik meg, mivel a `HorizontalAlignment` és a `VerticalAlignment` tulajdonságokat `Center` értékre állítottuk. Vegyük észre, hogy a sablon a `roundButtonTemplate` nevet kapta (az erőforráskulcs segítségével), amelyre akkor hivatkozhatunk, amikor hozzárendeljük a `Button` típus `Template` tulajdonságát:

```
<Window x:Class="ControlTemplates.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Control Templates" Height="162" Width="281" >
  <Grid>
    <Grid.Resources>
      <!-- A kerek gomb egyszerű sablonja a rács elemei számára -->
        <ControlTemplate x:Key ="roundButtonTemplate"
                         TargetType ="{x:Type Button}">
        <Grid>
          <Ellipse Name ="OuterRing" Width ="75"
                   Height ="75" Fill ="DarkGreen"/>
          <Ellipse Name ="InnerRing" Width ="60"
                   Height ="60" Fill ="MintCream"/>
          <ContentPresenter HorizontalAlignment="Center"
        </Grid>
      </ControlTemplate>
    </Grid.Resources>

    <!-- A sablonunk alkalmazása Button típuson -->
    <Button Name ="myButton" Foreground ="Black"
            FontSize ="20" FontWeight ="Bold"
            Template ="{StaticResource roundButtonTemplate}"
            Click ="myButton_Click"> Click!
    </Button>
  </Grid>
</Window>
```

Azt is vegyük észre, hogy a `Button` típus kezeli a kattintási eseményt (tételezzük fel, hogy a kattintási esemény kezelője tájékoztató üzenetet jelenít meg a `MessageBox.Show()` metódus segítségével). Ennek azért van jelentősége, mert a gomb - annak ellenére, hogy már nem a hagyományos „szürke téglalap" formában jelenik meg - még mindig `System.Windows.Controls.Button` típus, és ugyanazokkal a tulajdonságokkal, metódusokkal és eseményekkel rendelkezik mint az előre becsomagolt felhasználóifelület-megjelenéssel rendelkező gomb. A 30.17. ábra mutatja az egyedi gombtípusunkat működés közben.



30.17. ábra: *Egyszerű sablon a Button típus számára*

Triggerek hozzáadása a sablonokhoz

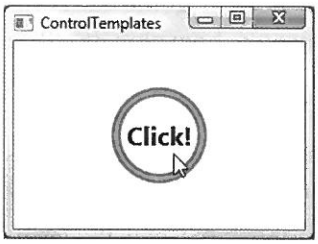
Pillanatnyilag a vezérlőelem-sablon lehetővé teszi, hogy a gomb körkörös formában jelenjen meg. Ha rákattintunk, észrevehetjük, hogy kiváltódik a kattintási esemény (a `MessageBox.Show()` metódus megjeleníti a sztringadatainkat); azonban a gomb megnyomásának látható jele nincs. Ez annak köszönhető, hogy az alapértelmezett gombnyomás-animációt eltávolítottuk, és egyedi felhasználói felületet alkalmaztunk. Ha szeretnénk visszatenni (vagy kicserélni) a nyomógomb-animációt, saját egyedi triggereket kell hozzáadnunk a kódhoz.

A következőkben bemutatjuk az aktuális sablon egy újabb verzióját, amely két triggert kezel. Az első trigger figyeli, hogy az egérmutató a gomb felett helyezkedik-e el vagy sem. Ha igen, módosítjuk a külső ellipszis háttérszínét (egyszerű vizuális effektus). A második trigger figyeli, hogy a gomb felületén a felhasználó kattint-e az egérrel vagy sem. Kattintás esetén a külső ellipszis `Height` és `Width` értékeit megnöveljük, hogy vizuális visszajelzést adjunk a felhasználónak:

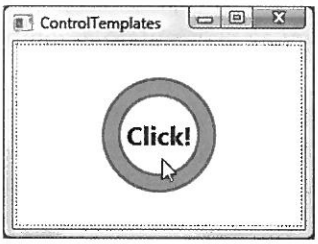
```
<Grid.Resources>
  <!-- Egyszerű sablon a kerek gomb számára -->
  <ControlTemplate x:Key ="roundButtonTemplate"
                   TargetType ="{x:Type Button}">
```

```
      <Grid>
        <Ellipse Name ="OuterRing" Width ="75"
                 Height ="75" Fill ="DarkGreen"/>
        <Ellipse Name ="InnerRing" Width ="60"
                 Height ="60" Fill ="MintCream"/>
        <ContentPresenter HorizontalAlignment="Center"
                          VerticalAlignment="Center"/>
      </Grid>
      <!-- Triggerekkel váltjuk ki a "kattintási" effektust -->
      <ControlTemplate.Triggers>
        <Trigger Property ="IsMouseOver" Value ="True">
          <Setter TargetName ="OuterRing" Property ="Fill"
                  Value ="MediumSeaGreen"/>
        </Trigger>
        <Trigger Property ="IsPressed" Value ="True">
          <Setter TargetName ="OuterRing" Property ="Height"
                  Value ="90"/>
          <Setter TargetName ="OuterRing" Property ="Width"
                  Value ="90"/>
        </Trigger>
      </ControlTemplate.Triggers>
    </ControlTemplate>
  </Grid.Resources>
```

A 30.18. ábrán láthatjuk az `IsMouseOver` trigger effektusát, a 30.19. ábra pedig az `IsPressed` trigger eredményét mutatja.



30.18. ábra: *Az IsMouseOver trigger működés közben*



30.19. ábra: *Az IsPressed trigger működés közben*

Sablonok használata a stílusokban

Pillanatnyilag a sablonunk egyszerűen a gomb megjelenését definiálja. A vezérlőelem alapvető tulajdonságainak (tartalom, betűméret, betűvastagság stb.) beállítása a `Button` típus feladata:

```
<!-- Pillanatnyilag a Button típus állít be alapvető
     tulajdonságértékeket, nem a sablon -->
<Button Name ="myButton" Foreground ="Black" FontSize ="20"
        FontWeight ="Bold"
        Template ="{StaticResource roundButtonTemplate}"
        Click ="myButton_Click">
```

Nagyszerű dolog lenne ezeket az értékeket *a sablonban* beállítani. Ily módon hatékonyan kialakíthatnánk a típus alapértelmezett megjelenését. Talán már kitaláltuk, hogy ez a feladat a WPF-stílusokra hárul. Amikor a stílust készítjük (és gondoskodunk az alapvető tulajdonság beállításokról), a sablont *a stílusban* definiálhatjuk! Íme a módosított `Grid` erőforrás magyarázattal együtt:

```
<Grid.Resources>
  <!-- A stílus a Button típus alapvető beállításait definiálja -->
  <Style x:Key ="roundButtonTemplate" TargetType ="{x:Type Button}">
    <Setter Property ="Foreground" Value ="Black"/>
    <Setter Property ="FontSize" Value ="20"/>
    <Setter Property ="FontWeight" Value ="Bold"/>

    <!-- Íme, a sablon! -->
    <Setter Property ="Template">
      <Setter.Value>
        <!-- Egyszerű sablon a kerek gomb számára -->
        <ControlTemplate TargetType ="{x:Type Button}">
          <Grid>
            <Ellipse Name ="OuterRing" Width ="75" Height ="75"
                     Fill ="DarkGreen"/>
            <Ellipse Name ="InnerRing" Width ="60" Height ="60"
                     Fill ="MintCream"/>
            <ContentPresenter HorizontalAlignment="Center"
                              VerticalAlignment="Center"/>
          </Grid>

          <!-- Trigger, amely a nyomógomb effektust biztosítja -->
          <ControlTemplate.Triggers>
            <Trigger Property ="IsMouseOver" Value ="True">
              <Setter TargetName ="OuterRing"
                      Property ="Fill" Value ="MediumSeaGreen"/>
            </Trigger>
            <Trigger Property ="IsPressed" Value ="True">
```

```
            <Setter TargetName ="OuterRing" Property ="Height"
                    Value ="90"/>
            <Setter TargetName ="OuterRing" Property ="Width"
                    Value ="90"/>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
 </Style>
</Grid.Resources>
```

Először is vegyük észre, hogy a `<ControlTemplate>` helyett a `<Style>` elem kapott erőforráskulcs értéket. Majd figyeljük meg, hogy a stílus ugyanazokat az alapvető tulajdonságokat állítja be, mint amelyeket a Button típus deklarációjában beállítottunk (`Foreground`, `FontSize` és `FontWeight`). A `<ControlTemplate>` elemet a `Template` tulajdonság módosításával, egy normál stílusú `<Setter>` elemmel definiáljuk. Ezzel a módosítással létrehozhatjuk az egyedi gombunkat, ha a következőképpen állítjuk be a `Style` tulajdonságot:

```
<!-- A stílus/sablon alkalmazása a Button típuson -->
<Button Name ="myButton"
        Style ="{StaticResource roundButtonTemplate}"
        Click ="myButton_Click">
  Click!
</Button>
```

Noha a gomb megjelenítése és viselkedése megegyező, a sablonok stílusokba ágyazásának nyilvánvaló előnye az, hogy közös tulajdonságok számára rögzített értékkészletet biztosíthatunk. Emlékezzünk rá, hogy természetesen módosíthatjuk az alapértelmezett értékeket egy vezérlőelemenként eltérő értékre:

```
<!-- Stílus beállítása, de az előtér színének módosítása -->
<Button Name =" myButton"
        Style ="{StaticResource roundButtonTemplate}"
        Click ="myButton_Click" Foreground ="Red">
  Click!
</Button>
```

Forráskód A ControlTemplates kódfájlokat a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Ezzel befejeztük a WPF stílus- és sablonmechanizmusainak vizsgálatát. Mint láthattuk, a sablonok alapját rendszerint több grafikai stílus képezi, és a sablonokat végeredményben bináris erőforrásként csomagoljuk az alkalmazásunkba.

Ezzel a könyv jelen kiadásában a WPF vizsgálatának végéhez érkeztünk. Az utóbbi három fejezetben rengeteget tanultunk a mögöttes WPF programozási modellről, a XAML-szintaxisról, a vezérlőelem-kezelésről és a grafikus tartalom elkészítéséről. Noha a WPF roppant nagy témakör, és a könyvben csak egy részét vizsgáltuk, mindez, a célunknak megfelelően, szilárd alapokat biztosít a további felfedezéshez.

Összefoglalás

Mivel a Windows Presentation Foundation (WPF) grafikaiművelet-intenzív grafikus felület API, nem meglepő, hogy a grafikus kimenet megjelenítéséhez több módszer áll rendelkezésünkre. A fejezet elején megvizsgáltuk a WPF-alkalmazások grafikus renderelésének három módját (alakzatok, rajzok és vizuális eszközök), és tanulmányoztuk a különböző renderelési primitíveket, például az ecseteket, a tollakat és a transzformációkat.

Ezt követően a C#-forráskód, valamint a XAML-deklarációk szempontjából vizsgáltuk meg a WPF animációs szolgáltatások szerepét. Ebben a részben megismerkedtünk az idősorok, a forgatókönyvek és a kulcsképkockák jellemzőivel. Megismerkedtünk a WPF erőforráskezelési API-jaival, láthattuk, hogy a WPF-erőforrások a sztringtáblázatok, ikonok és bittérképtípusok elvárt készlete mellett további elemeket hozhatnak magukkal, de egyedi objektumokat is képviselhetnek, amelyeket az erőforrásszótár tárol.

A fejezet végén összefoglaltuk ezeket a témaköröket, és felfedeztük a WPF-stílusok és -sablonok szerepét. Mint láttuk, a WPF a grafikai primitívek, az animációs szolgáltatások és a beágyazott erőforrások gyűjteménye segítségével megkönnyíti a vezérlőelemek megjelenésének kialakítását.

7. rész

Webes alkalmazások fejlesztése ASP.NET segítségével

HARMINCEGYEDIK FEJEZET

ASP.NET weboldalak készítése

A könyv eddigi alkalmazáspéldái konzol- és asztali GUI-alapú front endekre koncentráltak. A következő három fejezetben bemutatjuk, hogy a .NET platform hogyan könnyíti meg a böngészőalapú megjelenítő rétegek létrehozását az ASP.NET nevű technológia révén. Bevezetésként áttekintünk néhány kulcsfontosságú, webközpontú fogalmat (HTTP, HTML, ügyfél- és kiszolgálóoldali szkriptírás), valamint a Microsoft kereskedelmi webkiszolgálóját (IIS), illetve az ASP.NET fejlesztői webkiszolgálót (`WebDev.WebServer.exe`) vizsgáljuk meg.

A webes bevezetést követően, a fejezet további részei az ASP.NET-weboldalak szerkezetével foglalkoznak (beleértve az egyoldalas és a mögötteskód-modelleket is), és egy `Page`-leszármazott típus felépítését tanulmányozzák. A fejezet emellett a `Web.config` fájl szerepét is bemutatja, amelyet a későbbi, főként a webbel foglalkozó fejezetekben fogunk használni.

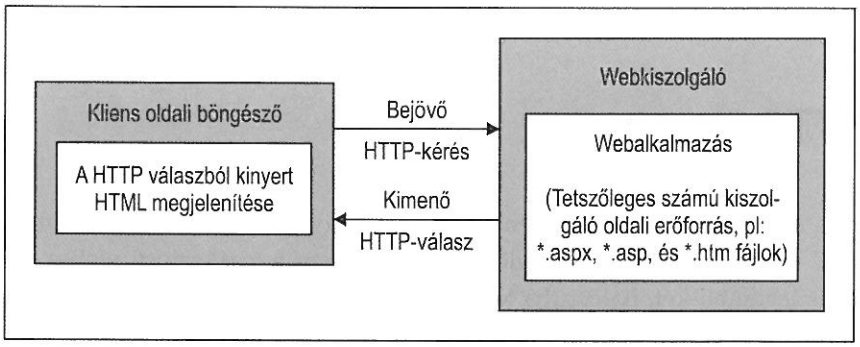
A HTTP szerepe

A webes alkalmazások lényegesen eltérnek a hagyományos asztali alkalmazásoktól. Az első szembetűnő különbség, hogy egy termékszintű webes alkalmazás legalább két, hálózatba kötött gépet igényel (fejlesztés alatt természetesen elég egyetlen gép, amely egyszerre tölti be a böngészőalapú kliens és a webkiszolgáló szerepét). A webes alkalmazások természetéből adódóan a hálózatba kötött gépeknek meg kell egyezniük, hogy milyen átviteli protokollon keresztül küldenek és fogadnak adatokat. A hálózati gépeket összekötő átviteli protokoll a HTTP (Hypertext Transform Protocol).

A HTTP kérés/válasz-ciklus

Amikor a kliensgép elindít egy webböngészőt (mint például az Opera, a Mozilla Firefox vagy a Microsoft Internet Explorer), egy HTTP-kéréssel fordul egy adott erőforráshoz (általában weboldalhoz) a távoli kiszolgálón. A HTTP szövegalapú protokoll, amely szabványos kérés/válasz paradigmára épül. Például, ha a http://www.intertech.com oldalra navigálunk, a böngésző a *Domain Name Service* (DNS) segítségével a regisztrált URL-t négy részből álló, 32 bites numerikus értékké konvertálja, amelyet *IP-címnek* nevezünk. Ekkor a böngésző megnyit egy csatornát (általában nem biztonságos kapcsolatot a 80-as porton keresztül), és elküldi a HTTP-kérést a célhelyre, feldolgozásra.

A webkiszolgáló fogadja a bejövő HTTP-kérést, és eldönti, hogy feldolgozza-e a kilens által küldött bemeneti értékeket (például egy szövegdobozban lévő értékeket, egy jelölőnégyzet-jelölést stb.), és összeállítja a megfelelő HTTP-választ. A webprogramozók számos technológiát alkalmazhatnak (CGI, ASP, ASP.NET, JSP stb.) a HTTP-válaszban megjelenített tartalom dinamikus generálására. Ekkor az ügyféloldali böngésző megjeleníti a webkiszolgálótól kapott HTML-t. A 31.1. ábra az alapvető HTTP kérés/válasz-ciklust mutatja be.



31.1. ábra: *A HTTP kérés/válasz-ciklus*

A HTTP állapotmentes protokoll

A webes fejlesztést az a tény is markánsan megkülönbözteti a hagyományos asztali programozástól, hogy a HTTP lényegében *állapotmentes* átviteli protokoll. Amint a webkiszolgáló elküldi a választ a kliensnek, mindent elfelejt az előző interakcióról. Ez persze nem igaz a hagyományos asztali alkalmazásokra, ahol a végrehajtható fájl állapota általában élő és működik, amíg a felhasználó le nem állítja a kérdéses alkalmazást.

Emiatt a fejlesztő felelőssége, hogy lépéseket tegyen a webhelyre bejelentkezett felhasználókkal kapcsolatos adatok (például bevásárlókosár elemei, hitelkártyaszámok, otthoni és munkahelyi címek stb.) „megjegyzése" érdekében. Ahogy az a 33. fejezetben kiderül majd, az ASP.NET több módszert (munkamenet-változók, sütik és alkalmazásváltozók) biztosít az állapot kezelésére, amelyek közül sok szinte bármely webplatformon megtalálható, ugyanúgy, mint egyes .NET-specifikus módszerek (például az ASP.NET profilkezelő API).

Webes alkalmazások és webkiszolgálók

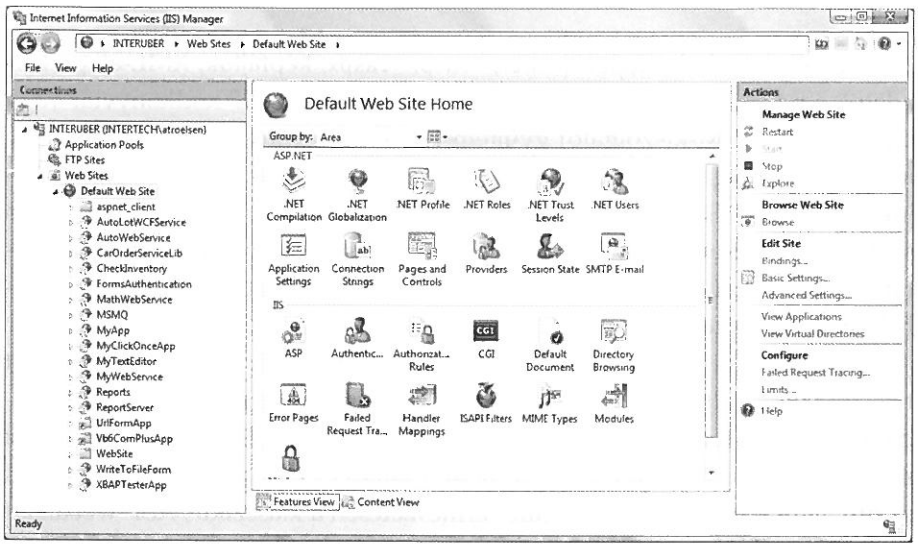
A *webes alkalmazásokat* fájlok (`*.htm`, `*.asp`, `*.aspx`, képfájlok, XML-alapú fájladatok stb.) és kapcsolódó összetevőik (például egy .NET-kódkönyvtár vagy egy korábbi COM-kiszolgáló) gyűjteményeként is felfoghatjuk, amelyet egy adott webkiszolgáló könyvtárkészlete tárol. Ahogyan azt a 33. fejezetben látjuk majd, az ASP-webalkalmazások sajátos életciklussal rendelkeznek, és számos olyan eseményt (például kezdeti indítás vagy végső leállítás) biztosítanak, amelyeket felhasználva speciális műveleteket hajthatunk végre a webhely működése közben.

A *webkiszolgálók* olyan szoftvertermékek, amelyek webalkalmazásokat hosztolnak, és általában több kapcsolódó szolgáltatást nyújtanak, mint például az integrált adatvédelem, a fájlátviteli protokoll (FTP), az üzenetváltási szolgáltatás stb. Az Internet Information Services (IIS) a Microsoft vállalati szintű webkiszolgáló terméke, ami természetesen a klasszikus ASP-webalkalmazások mellett az ASP.NET-webalkalmazásokat is támogatja.

Termékszintű ASP.NET-webalkalmazás létrehozásakor gyakran kell majd együttműködnünk az IIS-sel. Azonban jó, ha tudjuk, hogy a Windows operációs rendszer telepítésekor az IIS-t a telepítő *nem* telepíti automatikusan (ráadásul nem minden Windows-verzió támogatja az IIS-t, például a Windows XP Home sem). Ezért, a fejlesztő gépünk konfigurációjától függően, szükség lehet az IIS telepítésére, mielőtt továbbhaladunk a fejezetben. Ehhez nyissuk meg az Add/Remove Programot a Control Panel mappából, és válasszuk az Add/Remove Windows Components lehetőséget. További részletekért forduljunk a Windows-súgóhoz.

Megjegyzés A legjobb, ha az IIS telepítését a Visual Studio 2008 telepítése *előtt* elvégezzük. Ha a Visual Studio 2008 telepítését *követően* telepítjük az IIS-t, az ASP.NET-webalkalmazásaink nem fognak megfelelően működni (csak egy üres lapot kapunk). Szerencsére az IIS-t újrakonfigurálhatjuk, hogy hosztolja a .NET-alkalmazásokat, az `aspnet_regiis.exe` parancssori eszköz futtatásával, és az `/i` opció megadásával.

Ha a megfelelő módon telepítettük az IIS-t a munkaállomásunkra, az Administrative Tools mappában elérhetjük (amely a Control Panel mappában található), ha kétszer az Internet Information Services kisalkalmazásra kattintunk. Ebben a fejezetben kizárólag a Default Web Site csomóponttal foglalkozunk (lásd a 31.2. ábrát).



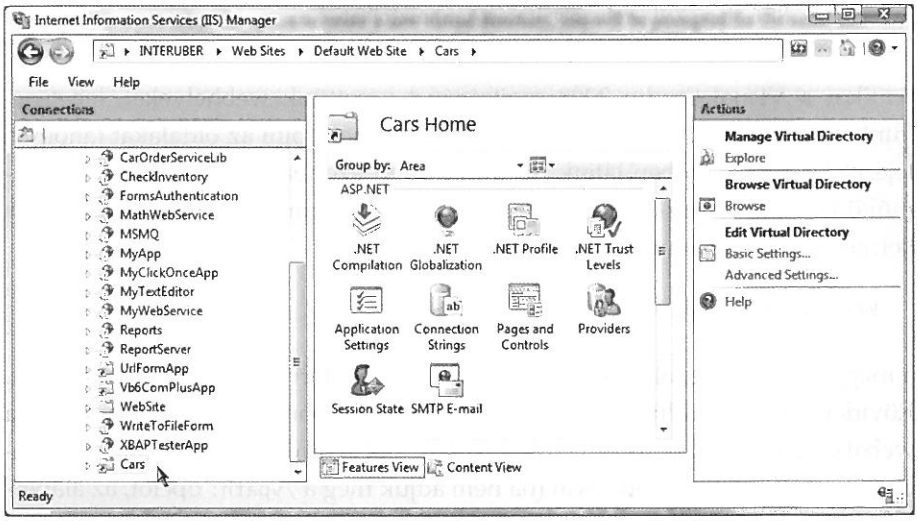
31.2. ábra: *Az IIS-kisalkalmazás*

Az IIS virtuális könyvtárainak szerepe

Egyetlen IIS-konfiguráció több olyan webalkalmazást képes hosztolni, amelyek mindegyike egy *virtuális könyvtárban* található. Minden virtuális könyvtár a merevlemez egy fizikai könyvtárára képezhető le. Ezért, ha létrehozunk egy új, CarsRUs nevű virtuális könyvtárat, a külvilág például a http://www.CarsRUs.com URL segítségével navigálhat erre a webhelyre (feltéve, hogy a webhely IP-címét regisztráltattuk). Ez a virtuális könyvtár a háttérben a webkiszolgáló egy fizikai gyökérkönyvtárára képezhető le, mint például a C:\inetpub\wwwroot\AspNetCarsSite, amely a CarsRUs webalkalmazás tartalmát foglalja magában.

Ahogy azt a fejezet későbbi részében láthatjuk, amikor ASP.NET-webalkalmazásokat hozunk létre a Visual Studio 2008 segítségével, lehetőségünk van arra, hogy az IDE automatikusan generáljon egy új virtuális könyvtárat az aktuális webhely számára. Ha szükséges, természetesen kézzel is létrehozhatunk virtuális könyvtárakat, ha a jobb gombbal az IIS Default Web Site csomópontjára kattintunk, majd az úszómenüből (a context menüből) a New > Virtual Directory lehetőséget választjuk (Vista esetében csak válasszuk az Add Virtual Directory menüpontot).

Az új virtuális könyvtár létrehozása után meg kell adnunk a nevét és azt a fizikai mappát, amelyben a webtartalmat elhelyezzük. Az IIS használatának bemutatására (és első webes példánk megalapozására), hozzunk létre egy új könyvtárat a merevlemezen a webtartalom számára. A példa kedvéért tételezzük fel, hogy ez a könyvtár a C:\CodeTests\CarsWebSite. Kattintsuk a jobb gombbal az IIS Default Web Site csomópontjára az új, Cars nevű virtuális könyvtár létrehozásához, amely erre az új könyvtárra képezhető le. A 31.3. ábra a végeredményt mutatja.



31.3. ábra: *A Cars virtuális könyvtár*

Nemsokára némi tartalommal töltjük meg ezt a webhelyet.

Az ASP.NET fejlesztőkiszolgálója

A .NET 2.0 előtt az ASP.NET-fejlesztőknek a webtartalom fejlesztése és tesztelése során az IIS virtuális könyvtárait kellett használniuk. Az IIS-től való erős függőség eredményeképpen sok esetben a szükségesnél jóval bonyolultabb volt a fejlesztőcsapat dolga (arról nem is beszélve, hogy a hálózati rendszergazdák nem örültek, hogy az összes fejlesztői gépen telepíteniük kellett az IIS-t). Szerencsére ma már lehetőségünk nyílik a `WebDev.WebServer.exe` webkiszolgáló használatára. Ez a segédprogram lehetővé teszi, hogy a fejlesztők az IIS nélkül hosztoljanak ASP.NET-webalkalmazásokat. Az eszköz segítségével a weboldalakat a gépünk bármely könyvtárában létrehozhatjuk és tesztelhetjük. Nagy hasznunkra válhat, ha csapatban fejlesztünk, vagy ha olyan Windows-verzióval készítünk ASP.NET-webprogramokat, amely nem támogatja az IIS-t (mint például a Windows XP Home).

Megjegyzés A `WebDev.WebServer.exe` nem használható klasszikus (COM-alapú) ASP-webalkalmazások tesztelésére vagy hosztolására. Ez a webkiszolgáló csak ASP.NET-webalkalmazásokat és/vagy .NET-alapú XML-webszolgáltatásokat hosztolhat.

Amikor a Visual Studio 2008 segítségével készítünk webhelyeket, lehetőségünk van a `WebDev.WebServer.exe` segítségével hosztolni az oldalakat (ahogy a fejezet későbbi részében látjuk majd). Emellett azonban manuálisan is együttműködhetünk az eszközzel a Visual Studio 2008 parancssorán keresztül. Ha beírjuk az alábbi parancsot:

```
WebDev.WebServer.exe -?
```

a megjelenő üzenetdobozban az érvényes parancssori paramétereket láthatjuk. Röviden, meg kell adnunk egy használaton kívüli portot (a `/port:` opcióval), a webalkalmazás gyökérkönyvtárát (a `/path:` opcióval) és egy opcionális virtuális útvonalat a `/vpath:` opcióval (ha nem adjuk meg a `/vpath:` opciót, az alapértelmezés a `/`). Tekintsük meg a következő parancsot, amely tetszőleges portot nyit meg a korábban létrehozott C:\CodeTests\CarsWebSite könyvtár tartalmának megtekintésére:

```
WebDev.WebServer.exe /port:12345 /path:"C:\CodeTests\CarsWebSite"
```

A parancs beírása után elindíthatjuk a böngészőnket az oldalak lekéréséhez. Ha a CarsWebSite mappában létezik `Default.aspx` nevű fájl, akkor írjuk be a következő URL-t:

```
http://localhost:12345/CarsWebSite/Default.aspx
```

Ebben és a következő fejezetben látható legtöbb példa a Visual Studio 2008 révén használja a `WebDev.WebServer.exe` kiszolgálót, ahelyett, hogy IIS-beli virtuális könyvtárban hosztolná a webtartalmat. Miközben ez a megközelítés egyszerűsítheti a webalkalmazások fejlesztését, ne feledjük, hogy ezt a webkiszolgálót *nem* termékszintű webalkalmazások hosztolására hozták létre. Tisztán fejlesztési és tesztelési célokra alkalmas. Ha elkészültünk a webalkalmazással, a webhelyet be kell másolnunk egy IIS-beli virtuális könyvtárba.

Megjegyzés A Mono-projektben (lásd a B függeléket) található egy ingyenes ASP.NET-bővítmény az Apache-webkiszolgálóhoz. Ennek segítségével lehetséges ASP.NET-webalkalmazásokat létrehozni és hosztolni Microsoft Windowstól eltérő operációs rendszereken. További információkért látogassunk el a http: //www.mono-project.com/ASP.NET oldalra.

A HTML szerepe

Most, hogy konfiguráltunk egy könyvtárat a webalkalmazásunk hosztolására, és kiválasztottunk egy webkiszolgálót, amely hosztként szolgál, hozzuk létre magát a tartalmat. Emlékezzünk vissza arra, hogy a „webalkalmazás" csak egy kifejezés a webhely működését biztosító fájlkészletre. Igazság szerint ezen fájlok nagy része HTML-ben definiált tokeneket tartalmaz. A HTML szabványos jelölőnyelv, amelyet annak leírására használunk, hogy a literális szövegek, képek, külső hivatkozások és különféle HTML-alapú felhasználói felületi vezérlők hogyan jelenjenek meg az ügyféloldali böngészőben.

A webes fejlesztésnek ez a jellege az egyik fő oka annak, hogy sok programozó idegenkedik a webalapú programok készítésétől. Noha a modern IDE-k (köztük a Visual Studio 2008) és webes fejlesztői platformok (mint például az ASP.NET) automatikusan generálják a HTML nagy részét, jobban járunk, ha gyakorlati HTML-tudásra teszünk szert az ASP.NET használata során.

Megjegyzés Emlékezzünk rá a 2. fejezetből, hogy a Microsoft több ingyenes IDE-t kiadott az Express-termékcsaláddal (mint például a Visual C# Expresst). Ha érdeklődünk a webes fejlesztés iránt, letölthetjük a Visual Web Developert is. Ezt az ingyenes IDE-t kifejezetten az ASP.NET-webalkalmazások létrehozására fejlesztették.

Annak ellenére, hogy ez a rész nem dolgozza fel teljes körűen a HTML-t, vegyük át az alapokat, és hozzunk létre egyszerű webalkalmazásokat HTML, klasszikus (COM-alapú) ASP, illetve IIS segítségével. Ez majd útmutatóul szolgál azoknak, akik hagyományos asztali alkalmazásfejlesztési háttérrel térnek át az ASP.NET-re.

Megjegyzés Ha már otthonosan mozgunk a weboldal-fejlesztés folyamatainak világában, nyugodtan ugorjunk az „A klasszikus ASP problémái" című részhez.

HTML-dokumentumstruktúrák

A HTML-fájlok olyan tagekből épülnek fel, amelyek egy adott weboldal megjelenését és működését írják le. Ahogy várhattuk, a HTML-dokumentumok alapstruktúrája ugyanolyan marad. Például a `*.htm` fájlok (vagy a `*.html` fájlok) `<html>` és `</html>` tagekkel kezdődnek és végződnek, általában egy `<body>` részt definiálnak, és így tovább. Ne feledjük, hogy a hagyományos HTML *nem* különbözteti meg a kis- és nagybetűket. Ezért a hosztböngésző szempontjából a `<HTML>`, a `<html>` és a `<HtmL>` ugyanazt jelenti.

A HTML-alapok szemléltetéséhez nyissuk meg a Visual Studio 2008 alkalmazást, hozzunk létre egy üres HTML-fájlt a File > New > File menüpont segítségével, majd mentsük a C:\CodeTests\CarsWebSite könyvtárba `default.htm` néven. Láthatjuk, hogy a kezdeti markupban nincs semmi különös:

```
<!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>Untitled Page</title>
</head>
<body>

</body>
</html>
```

Először is figyeljük meg, hogy ez a HTML-fájl egy DOCTYPE feldolgozási utasítással kezdődik. Ez tudatja az IDE-vel, hogy a tárolt HTML-tageket az XHTML-szabvány szerint kell ellenőrizni. Ahogy gondoltuk, a hagyományos HTML szintaxisa elég „laza" volt, például megtehettük, hogy olyan nyitó elemet definiáljunk (például sortöréshez
), amelyhez nem tartozott megfelelő záróelem (ebben az esetben </br>), nem különböztette meg a kis- és nagybetűket, és így tovább. Az XHTML-szabvány W3C-specifikáció, amely néhány igen fontos korlátozással látja el a HTML jelölőnyelvet.

Megjegyzés Alapértelmezés szerint a Visual Studio 2008 az összes HTML-dokumentumot az XHTML 1.0 Transitional ellenőrző séma szerint érvényesíti. Tulajdonképpen a HTML-*ellenőrző-sémákkal* biztosítjuk, hogy a markup megfeleljen bizonyos szabványoknak. Ha egy másik ellenőrzősémát szeretnénk megadni, nyissuk meg a Tools > Options párbeszédablakot, majd jelöljük ki a Validation csomópontot a HTML alatt. Ha nem szeretnénk megtekinteni az ellenőrzés során talált hibákat, kapcsoljuk ki a Show Errors jelölőnégyzetet.

A <html> és a </html> címkékkel a dokumentumok kezdetét és végét jelöljük. Figyeljük meg, hogy a <html> nyitó címkét egy xmlns (XML-névtér) attribútum tovább minősíti, amely a dokumentumban esetlegesen előforduló, különféle címkéket minősíti (ismétlésképpen, ezek a tagek alapértelmezés szerint az XHTML-szabványra épülnek). A webböngészők ezen címkék segítségével tudják, mikor kezdjék alkalmazni a dokumentum törzsében meghatározott megjelenítési formátumokat. Az aktuális tartalom jelentős részét a <body> hatókörében definiáljuk. Hogy kissé csinosabbá tegyük, módosítsuk az oldal fejlécét az alábbiak szerint:

```
<head>
  <title>This Is the Cars Web site</title>
</head>
```

Nem meglepő, hogy a <title> címkével a hívó webböngésző címsorában megjelenő sztringet határoztuk meg.

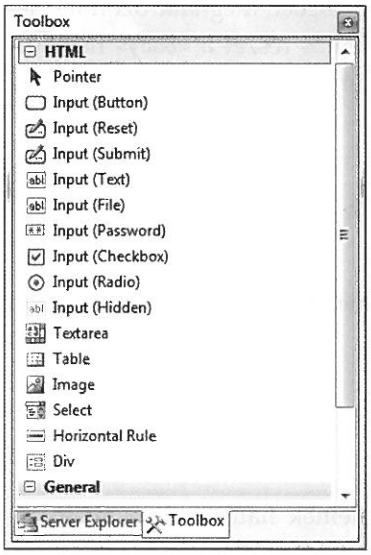
HTML-űrlapok fejlesztése

A legtöbb *.htm fájl lényegi része a <form> elemek hatókörében található. A *HTML-űrlap* olyan kapcsolódó felhasználóifelületi-elemek megnevezett csoportja, amelyeket a felhasználói bemenetek összegyűjtésére használunk, ami később a webalkalmazásba továbbítódik egy HTTP-kérésen keresztül.

A HTML-űrlapot ne tévesszük össze a böngészőben látható teljes megjelenítő területtel. A HTML-űrlap valójában inkább a <form> és a </form> címkékben elhelyezett vezérlők *logikai csoportosítása*:

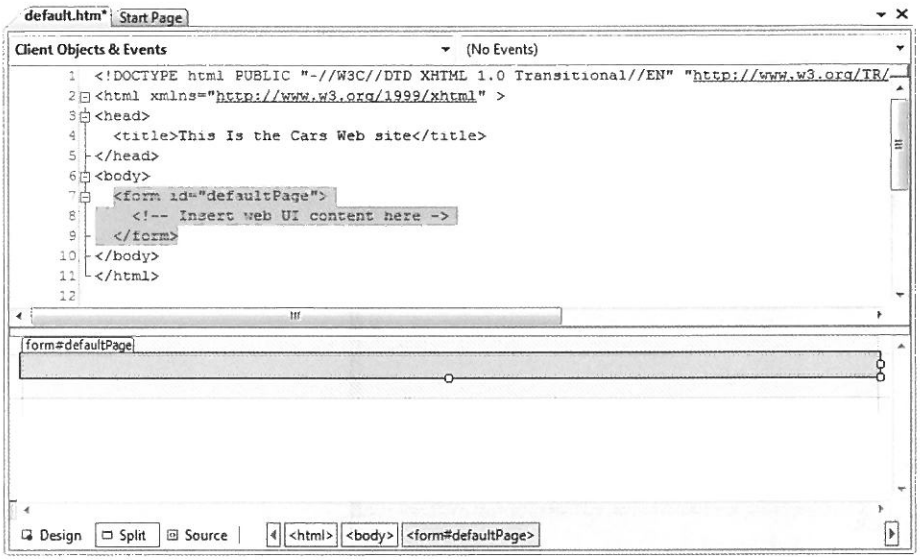
```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>This Is the Cars Web site</title>
</head>
<body>
  <form id="defaultPage">
    <!-- Ide szúrjuk be a webes felhasználói felület tartalmát ->
  </form>
</body>
</html>
```

Ehhez az űrlaphoz a „defaultPage" nevet és azonosítót rendeltük hozzá. A nyitó <form> címke általában tartalmaz egy műveleti attribútumot, amely azt az URL-t adja meg, ahová az űrlap adatait továbbítjuk, illetve meghatározza az adatátvitel metódusát (POST vagy GET). Hamarosan megvizsgáljuk a <form> címkét ebből a szempontból, előtte azonban nézzük meg, milyen elemeket helyezhetünk el egy HTML-űrlapon (az egyszerű szöveges részeken túl). A Visual Studio 2008 eszköztára tartalmaz egy HTML-fület, amelyen a 31.4. ábrán látható módon kiválaszthatjuk az egyes HTML-alapú felhasználóifelületi-vezérlőket.



31.4. ábra: *Az eszköztár HTML-füle*

Hasonlóan a Windows Forms vagy a WPF-alkalmazások készítésének folyamatához, ezeket a HTML-vezérlőelemeket a HTML-tervezőfelületre húzhatjuk. A HTML-szerkesztő alsó ablaktáblája alapértelmezés szerint a HTML vizuális elrendezését mutatja, a felső pedig a kapcsolódó markupot. A szerkesztő másik előnye, hogy amikor kijelölünk egy markupot vagy egy HTML-beli felhasználóifelület-elemet, a kapcsolódó ábrázolást a rendszer kiemeli. Ezáltal könnyedén átláthatjuk a módosításaink hatását (lásd a 31.5. ábrát).



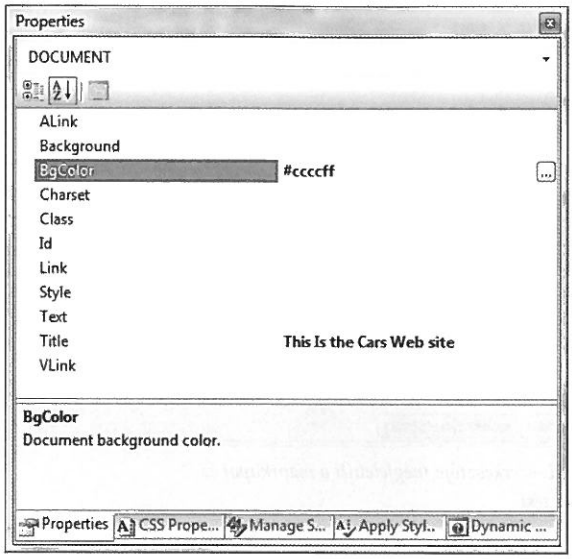
31.5. ábra: *A Visual Studio 2008 HTML-szerkesztője megjeleníti a maprkupot és a felhasználói felületi elrendezést*

HTML-alapú felhasználói felület készítése

Mielőtt a HTML `<form>` űrlaphoz hozzáadnánk a HTML-vezérlőket, érdemes kiemelni, hogy a Visual Studio 2008 integrált HTML-tervezője és a Properties ablak révén lehetővé teszi, hogy a `*.htm` fájlok megjelenését és működését átszabjuk. Ha a 31.6. ábrán látható módon kijelöljük a DOCUMENT elemet a Properties ablak legördülő listájában, beállíthatjuk a HTML-lap különböző tulajdonságait, például a háttérszínt, a háttérképet, a fejlécet és így tovább.

Módosítsuk a default.htm fájl `<body>` törzsét, hogy olyan szöveget jelenítsen meg, amely felhasználónév és jelszó beírására szólítja fel a felhasználót, majd válasszunk ki egy tetszőleges háttérszínt (szöveges tartalmat közvetlenül a HTML-tervezőbe is beírhatunk és formázhatunk):

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>This is the Cars Web site</title>
</head>
<body bgcolor="NavajoWhite">
  <!-- Felhasználói bemenet kérése -->
  <h1 align="center"> The Cars Login Page</h1>
  <p align="center"> <br/>
    Please enter your <i>user name</i> and <i>password</i>.
  </p>
  <form id="defaultPage">
  </form>
</body>
</html>
```

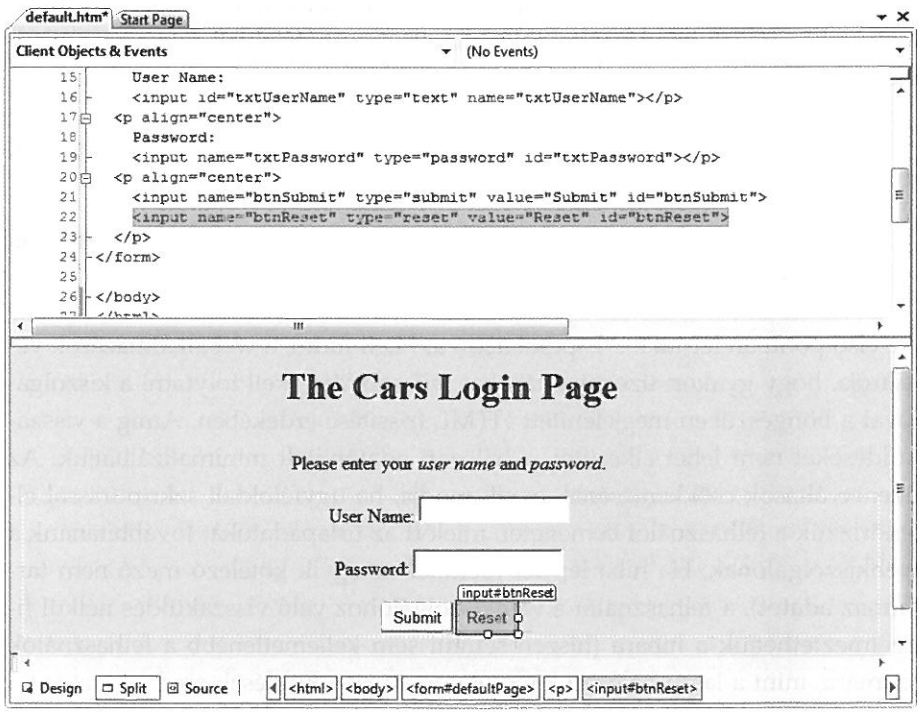


31.6. ábra: *HTML-dokumentum szerkesztése a Visual Studio 2008 Properties ablaka segítségével*

Most készítsük el magát a HTML-űrlapot. Általánosságban elmondható, hogy minden HTML-vezérlőt egy név attribútummal (az elem programozott azonosítására szolgál) és egy típus attribútummal (a `<form>` deklarációba beszúrt felhasználóifelület- elem típusának meghatározására szolgál) írunk le. Attól függően, hogy melyik felhasználóifelület-elemet kezeljük, az adott elemhez kapcsolódó további attribútumokat találunk a Properties ablakban, amelyeket módosíthatunk.

A felhasználói felület, amelyet hamarosan elkészítünk, két szövegmezőt (ebből az egyik egy `Password` vezérlő) és két gombot (az egyik az űrlapadatok továbbításához, a másik az űrlapadatok értékének alaphelyzetbe állításához szükséges) foglal magában:

```
<!-- Készítsünk űrlapot a felhasználói adatok begyűjtésére -->
<form id="defaultPage">
  <p align="center">
    User Name:
    <input id="txtUserName" type="text" name="txtUserName"/></p>
  <p align="center">
    Password:
    <input name="txtPassword" type="password" id="txtPassword"/></p>
  <p align="center">
    <input name="btnSubmit" type="submit" value="Submit"
           id="btnSubmit"/>
    <input name="btnReset" type="reset" value="Reset"
           id="btnReset"/>
  </p>
</form>
```



31.7. ábra: *A default.htm oldal kezdeti állapota*

Figyeljük meg, hogy minden vezérlőhöz vonatkozó nevet és azonosítót rendeltünk (`txtUserName`, `txtPassword`, `btnSubmit` és `btnReset`). Rendkívül fontos megjegyezni, hogy minden egyes bemeneti elemhez tartozik egy típus nevű kiegészítő attribútum.

Ez az attribútum olyan felhasználóifelületi-elemként jelöli a gombokat, amelyek automatikusan törlik az összes mezőt és visszaállítják a kezdőértéküket (`type="reset"`), jelszóként maszkolják a bemenetet (`type="password"`), vagy továbbítják az űrlapadatokat (`type="submit"`). A 31.7. ábrán az eddig elkészült oldal látható.

Az ügyféloldali szkriptírás szerepe

Egy adott `*.htm` fájl a HTML felhasználóifelületi-elemeken kívül olyan szkriptkódblokkokat is tartalmazhat, amelyeket a válaszfolyamba továbbítunk azért, hogy a kérelmező böngésző feldolgozhassa. Az ügyféloldali szkriptírás alkalmazásának két fő oka van:

- Már a böngészőben szeretnénk ellenőrizni a felhasználói bemenetet, mielőtt visszaküldenénk azt a webkiszolgálónak.
- Együttműködhetünk a böngésző dokumentum-objektummodelljével (DOM).

Az első pontban leírtakkal kapcsolatban azt kell tudni, a webalkalmazások velejárója, hogy gyakori üzenetváltásokat (*visszaküldést*) kell folytatni a kiszolgálóval a böngészőben megjelenített HTML frissítése érdekében. Amíg a visszaküldéseket nem lehet elkerülni, a hálózati adatátvitelt minimalizálhatjuk. Az üzenetváltások csökkentésének egyik módja, ha ügyféloldali szkriptírással ellenőrizzük a felhasználói bemenetet, mielőtt az űrlapadatokat továbbítanánk a webkiszolgálónak. Ha hiba lép fel (például az egyik kötelező mező nem tartalmaz adatot), a felhasználót a webkiszolgálóhoz való visszaküldés nélkül figyelmeztethetjük a hibára (hiszen semmi sem kellemetlenebb a felhasználók számára, mint a lassú hálózati kapcsolaton a visszaküldések eredményeként a bemeneti hibákkal kapcsolatos figyelmeztetések megjelenése!).

Megjegyzés Tudni kell, hogy még ügyféloldali ellenőrzés alkalmazásakor (amit a válaszidő csökkentése érdekében teszünk) *is* előfordulhat ellenőrzés magán a webkiszolgálón. Ezzel biztosíthatjuk, hogy az adatok nem változtak meg az adatátvitel során. Ahogy azt a következő fejezet bővebben tárgyalja, az ASP.NET ellenőrző vezérlőelemek automatikusan végrehajtják az ügyfél- és kiszolgálóoldali ellenőrzéseket.

A felhasználói bemenetek ellenőrzése mellett az ügyféloldali szkriptek révén kapcsolatba léphetünk a webkiszolgáló objektummodelljével (DOM). A legtöbb kereskedelmi forgalomban lévő böngésző tartalmaz olyan objektumkészletet, amellyel befolyásolhatjuk a böngésző viselkedését. Azonban kellemetlen lehet, hogy a különböző böngészők hasonló, de mégsem azonos objektummodelleket tartalmaznak. Ezért, ha olyan ügyféloldali szkriptkódblokkot bocsájtunk ki, amely használja a DOM-ot, nem biztos, hogy az összes böngészőben ugyanúgy működik majd.

Megjegyzés Az ASP.NET biztosítja a `HttpRequest` tulajdonságot. A böngészőtulajdonság segítségével futási időben meghatározhatjuk az aktuális kérést küldő böngésző képességeit.

Az ügyféloldali szkriptkódokat több szkriptnyelven megírhatjuk. A két legkedveltebb nyelv a VBScript és a JavaScript. A VBScript a Visual Basic 6.0 programozási nyelv része. A Microsoft Internet Explorer az egyetlen olyan webböngésző, amely beépített támogatással rendelkezik az ügyféloldali VBScript számára (más böngészők tartalmazhatnak opcionális beépülő modulokat). Ezért, ha azt szeretnénk, hogy a HTML-oldalaink bármely böngészőben megfelelően működjenek, *ne* VBScriptben írjuk meg az ügyféloldali szrkipteket.

A másik népszerű szkriptnyelv a JavaScript. Rendkívül fontos tudni, hogy a JavaScript semmilyen tekintetben nem része a Java nyelvnek. Amíg a JavaScript és a Java szintaxisa valamelyest hasonló, a JavaScript nem teljesen önálló OOP-nyelv, ezért jóval kevésbé hatékony, mint a Java. Jó hír, hogy az összes mai webböngésző támogatja a JavaScriptet, ezért leginkább ez alkalmas szkriptírásra.

Megjegyzés Hogy tovább bonyolítsuk a dolgot, emlékezzünk vissza arra, hogy a JScript.NET olyan felügyelt nyelv, amellyel érvényes .NET-szerelvényeket készíthetünk egy szkriptszerű szintaxissal.

Példa az ügyféloldali szkriptírásra

Az ügyféloldali szkritírás szerepének bemutatásához először vizsgáljuk meg, hogyan kell elkapni az ügyféloldali HTML grafikus felületi vezérlők eseményeit. Tételezzük fel, hogy hozzáadtunk egy további HTML-gombot (`btnHelp`) a `default.htm` oldalhoz, amely révén a felhasználó súgóinformációkat tekinthet meg.

A gomb kattintási eseményének kezeléséhez jelöljük ki a HTML-űrlaptervező bal felső legördülő listájában a `btnHelp` lehetőséget, majd válasszuk az „`onclick`" eseményt a jobb oldali legördülő listából. Ezzel egy `onclick` attribútumot adtunk az új `Button` típus definíciójához:

```
<input id="btnHelp" type="button" value="Help" language="javascript"
       onclick="return btnHelp_onclick()" />
```

A Visual Studio 2008 szintén létrehoz egy üres JavaScript függvényt, amelyet a rendszer akkor hív meg amikor a felhasználó a gombra kattint. Ebben az üres rutinban az `alert()` metódust használjuk ügyféloldali üzenetdoboz megjelenítésére:

```
<script language="javascript" type="text/javascript">
// <!CDATA[
function btnHelp_onclick() {
  alert("Dude, it is not that hard. Click the Submit button!");
}
// ]]>
</script>
```

Figyeljük meg, hogy a szkriptblokkot egy CDATA szakaszba illesztettük be. Ennek egyszerű oka van, ha az oldal olyan böngészőbe kerül, amely nem támogatja a JavaScriptet, a böngésző megjegyzésblokként értelmezi a kódot, és figyelmen kívül hagyja. Az oldal természetesen kevésbé működőképes, de a lényeg, hogy nem robban szét, amikor megjelenik a böngészőben.

A default.htm űrlapadatainak ellenőrzése

Most frissítsük a default.htm oldalt, hogy támogasson néhány ügyféloldali ellenőrző algoritmust. A cél annak biztosítása, hogy amikor a felhasználó a Submit gombra kattint, meghívjunk egy JavaScript függvényt, amely ellenőrzi, hogy nincsenek-e üres értékek az egyes szövegdobozokban. Ha vannak, egy előugró figyelmeztetéssel utasítjuk a felhasználót a kért adat megadására. Először kezeljük az `onclick` eseményt a Submit gombra:

```
<input name="btnSubmit" type="submit" value="Submit" id="btnSubmit"
       language="javascript" onclick="return btnSubmit_onclick()">
```

Az alábbiak szerint implementáljuk az eseménykezelőt:

```
function btnSubmit_onclick(){
  // Ha az egyik elemről megfeledkeztek, jelenítsünk meg egy
  // üzenetdobozt.
  if((defaultPage.txtUserName.value == "") ||
     (defaultPage.txtPassword.value == ""))
  {
    alert("You must supply a user name and password!");
    return false;
  }
  return true;
}
```

Most mentsük az eddigi munkánkat. Nyissuk meg a böngészőnket, navigáljunk a default.htm oldalra, amelyet a Cars virtuális könyvtárunk hosztol, és teszteljük az ügyféloldali szkriptünket:

```
http://localhost/Cars/default.htm
```

Ha a Help vagy a Submit gombra kattintunk, a böngészőnk a megfelelő üzenetdobozokat jeleníti meg. Figyeljük meg, hogy ha véletlenszerű adatokat írunk be a szövegdobozokba, a Submit gombra kattintva nem jelenik meg ellenőrzési hibaüzenet.

Az űrlapadatok továbbítása (a GET és a POST)

Most, hogy már van egy egyszerű HTML-oldalunk, vizsgáljuk meg, hogyan kell az űrlapadatokat visszaküldeni a webkiszolgálónak feldolgozásra. Amikor HTML-űrlapot készítünk, általában elhelyezünk egy műveletattribútumot a nyitó `<form>` címkében a bemeneti űrlapadatok címzettjének meghatározásához. A lehetséges címzettek levélkiszolgálók, más HTML-fájlok, egy Active Server Pages (ASP-) fájl és egyéb objektumok lehetnek. Ebben a példában a `ClassicAspPage.asp` nevű, klasszikus ASP-fájlt használjuk. Módosítsuk a `default.htm` fájlt a következő attribútum megadásával a nyitó `<form>` címkében:

```
<form name="defaultPage" id="defaultPage"
    action="http://localhost/Cars/ClassicAspPage.asp" method="GET">
  ...
</form>
```

A kiegészítő attribútumok biztosítják, hogy amikor a felhasználó az űrlap Submit gombjára kattint, az űrlapadatokat a rendszer a megadott URL-en keresztül a ClassicAspPage.asp fájlba továbbítsa. Amikor az adatátvitel módjaként a `method="GET"` metódust adjuk meg, az űrlapadatot egy & jellel elválasztott név/érték párral fűzzük hozzá a querystringhez.

```
http://localhost/Cars/ClassicAspPage.asp?txtUserName=
Andrew&txtPassword=Foo$&btnSubmit=Submit
```

Az űrlapadatok webkiszolgálóra továbbításának másik módja a `method="POST"` metódus meghatározása:

```
<form name="defaultPage" id="defaultPage"
  action="http://localhost/Cars/ClassicAspPage.asp" method = "POST">
  ...
</form>
```

Ebben az esetben az űrlapadatot nem fűzzük hozzá a querystringhez, hanem az a HTTP-fejléc külön soraként jelenik meg. A `POST` használatakor a külvilág nem láthatja közvetlenül az űrlapadatokat. Nagyon fontos, hogy a `POST` adatok karakterszáma nem korlátozott (míg sok böngésző korlátozza a `GET` lekérdezések karakterszámát). Egyelőre a HTTP `GET` metódusa révén küldjük el az űrlapadatokat a fogadó `*.asp` oldalra.

Klasszikus ASP-oldal készítése

Egy klasszikus ASP-oldal HTML és kiszolgálóoldali szkriptkód keveréke. Ha eddig még nem használtuk a klasszikus ASP-t, tudnunk kell, az ASP célja, hogy dinamikusan készíthessünk HTML-t olyan *kiszolgálóoldali* szkript révén, amely COM-objektumok kis gyűjteményét és egy kis munkát igényel. Például adott egy kiszolgálóoldali VBScript (vagy JavaScript) blokk, amely egy adatforrás tábláját olvassa a klasszikus ADO segítségével, a sorokat pedig generikus HTML-táblaként adja vissza.

Ebben a példában az ASP-oldal a belső ASP Request COM-objektumot használja a querystringhez fűzött, bemeneti űrlapadatok értékeinek kiolvasására, majd visszaküldi azokat a hívónak (nem valami izgalmas, de jól szemlélteti a kérés/válasz ciklus alapvető működését). A kiszolgálóoldali szkriptlogika VBScriptet használ (ahogy azt a nyelvdirektíva jelzi).

Ehhez hozzunk létre új HTML-fájlt a Visual Studio 2008 segítségével, és mentsük el `ClassicAspPage.asp` néven abban a mappába, amelyre a virtuális könyvtárunkat leképeztük (például C:\CodeTests\CarsWebSite). Az oldal megvalósítása a következő:

```
<%@ language="VBScript" %>
<html>
<head>
  <title>The Cars Page</title>
</head>
  <body>
    <h1 align="center">Here is what you sent me:</h1>
    <P align="center"> <b>User Name: </b>
    <%= Request.QueryString("txtUserName") %> <br>
    <b>Password: </b>
    <%= Request.QueryString("txtPassword") %> <br>
    </P>
  </body>
</html>
```

Most a klasszikus ASP Request COM-objektum segítségével hívtuk a `QueryString()` metódust a `method="GET"`-tel továbbított HTML-vezérlők értékeinek vizsgálatára. A `<%= ...%>` jelölés röviden ennyit jelent: „A következők beszúrása közvetlenül a kimenő HTTP-válaszba." A rugalmasság növelése érdekében az ASP Response COM-objektummal együttműködhetünk egy kiszolgálóoldali szkriptblokkon keresztül (ezt a `<%, %>` jelölés mutatja). Azonban erre most nincs szükségünk, hiszen a következő példa rendkívül egyszerű:

```
<%
  Dim pwd
  pwd = Request.QueryString("txtPassword")
  Response.Write(pwd)
%>
```

A klasszikus ASP `Request` és `Response` objektumai nyilvánvalóan rengeteg további taggal rendelkeznek az itt bemutatottakon kívül. Ráadásul a klasszikus ASP is definiál néhány kiegészítő COM-objektumot (`Session`, `Server`, `Application` stb.), amelyeket webalkalmazásaink készítésekor felhasználhatunk.

Megjegyzés Ezek a COM-objektumok az ASP.NET alatt hivatalosan már nem léteznek. Azonban látjuk majd, hogy a `System.Web.UI.Page` ősosztály azonos nevű tulajdonságokat definiál, amelyek hasonló működésű objektumokat foglalnak magukban.

Ezt követően mentsük mindegyik webfájlunkat. Az ASP-logika teszteléséhez csak töltsük be egyszerűen a default.htm oldalt egy böngészővel, majd továbbítsuk az űrlapadatokat. Miután a webkiszolgáló feldolgozta a szkriptet, a 31.8 ábrán látható teljesen új (dinamikusan generált) HTML-t kapjuk vissza.

31.8. ábra: *A dinamikusan generált HTML*

A default.htm fájl jelenleg a HTTP GET metódust adja meg az űrlapadatok küldéséhez az `*.asp` célfájlba. Így a különféle grafikus felületi vezérlőkben tárolt értékeket a querystring végéhez fűzzük hozzá. Fontos megjegyezni, hogy az ASP `Request.QueryString()` metódus *kizárólag* a `GET` metódussal továbbított adatok kiemelésére képes.

Ha inkább a HTTP `POST` metódussal szeretnénk továbbítani az űrlapadatokat a webes erőforráshoz, az értékeket a `Request.Form` gyűjteménnyel olvashatjuk a kiszolgálón, például a következőképpen:

```
<body>
  <h1 align="center">Here is what you sent me:</h1>
  <P align="center">
    <b>User Name: </b>
    <%= Request.Form("txtUserName") %> <br>
    <b>Password: </b>
    <%= Request.Form("txtPassword") %> <br>
  </P>
</body>
```

Ezzel a webbel részletesen foglalkozó bevezető végére értünk. Ha most ismerkedünk a webes fejlesztéssel, remélhetőleg nagyobb rálátásunk nyílt a webalapú alkalmazások alapvető építőelemeire. Azonban mielőtt megnéznénk, hogyan tökéletesíti az ASP.NET webplatform a jelenlegi helyzetet, szánjunk néhány percet a klasszikus ASP kritikájára, valamint főbb korlátainak vizsgálatára.

Forráskód A ClassicAspCars kódfájlokat a forráskódkönyvtár 31. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A klasszikus ASP problémái

Noha sok sikeres webhelyet a klasszikus ASP-vel hoztak létre, ennek az architektúrának is megvannak a maga hátrányai. A klasszikus ASP legnagyobb hátránya talán ugyanaz, mint ami hatékony platformmá teszi: a kiszolgálóoldali szkriptnyelvek. A szkriptnyelvek, mint a VBScript és a JavaScript interpretált, típus nélküli entitások, amelyek nem alkalmasak robusztus OO programozási módszerekre.

A klasszikus ASP másik problémája, hogy egy `*.asp` oldal nem szolgál modularizált kóddal. Mivel az ASP HTML és szkript keveréke *egyetlen* oldalon, a legtöbb ASP-webalkalmazás két különböző programozási módszer zavaros elegye. Amíg igaz, hogy a klasszikus ASP révén az újrafelhasználható kódokat különálló fájlokra oszthatjuk, az alapul szolgáló objektummodell nem támogatja a kapcsolatok valódi szétválasztását. Az ideális az lenne, ha a webes keretrendszerek lehetővé tennék, hogy a megjelenítő logika (például a HTML-címkék) az üzleti logikától (például a funkcionális kódtól) függetlenül létezzen.

Szintén megfontolandó probléma, hogy a klasszikus ASP túl sok sablonszöveget és redundáns szkriptkódot igényel, amelyek a projektek között ismétlődnek. Szinte az összes webalkalmazásnak ellenőriznie kell a felhasználói bemeneteket, újra fel kell töltenie a HTML-vezérlők állapotát a HTTP-válasz kiadása előtt, generálnia kell egy HTML-adattáblát, és így tovább.

Az ASP.NET 1.x főbb előnyei

Az ASP.NET első kiadása (1.x verzió) nagyszerűen feloldotta a klasszikus ASP korlátait. Röviden, a .NET platform a következő módszereket vezette be a Microsoft webes fejlesztési paradigmán:

- Az ASP.NET a *mögötteskód*-modellt használja, amely révén szétválaszthatjuk a megjelenítési logikát az üzleti logikától.
- Az ASP.NET-oldalak implementálása .NET programozási nyelveken történik, és nem interpretált szkriptnyelveken. A kódfájlokat érvényes .NET-szerelvényekre fordítjuk (ami jóval gyorsabb végrehajtást eredményez).

- A webes vezérlőelemek segítségével a programozók a Windows Forms/WPF alkalmazásoknál megszokott módon készíthetik el a webalkalmazások grafikus felületét.
- Az ASP.NET webes vezérlőelemek automatikusan karban tartják az állapotukat a visszaküldések alatt a `__VIEWSTATE` nevű rejtett űrlapmező segítségével.
- Az ASP.NET-webalkalmazások teljes mértékben objektumorientáltak és az egységes típusrendszert (CTS) használják.
- Az ASP.NET-webalkalmazások könnyedén konfigurálhatók szabványos IIS-beállítások *vagy* webalkalmazás-konfigurációs fájl (`Web.config`) segítségével.

Az ASP.NET legfőbb újdonságai

Miközben az ASP.NET 1.x volt az első fontos lépés a megfelelő irányba, az ASP.NET 2.0 további újdonságokkal bővült, ezek a következők:

- A `WebDev.WebServer.exe` tesztelő webkiszolgáló bevezetése.
- További nagy számú webes vezérlőelem (navigációs vezérlőelemek, biztonsági vezérlőelemek, új adat-vezérlőelemek, új felhasználóifelület-vezérlőelemek stb.).
- A *mesteroldalak* bevezetése, amelyek révén közös felhasználóifelület-keretet csatolhatunk az oldalakhoz.
- A *témák* támogatása, amelyek révén deklaratív módon módosíthatjuk a teljes webalkalmazás megjelenését és működését.
- A *webkijelzők* (web part) támogatása, amellyel lehetővé tehetjük a végfelhasználók számára a weboldal megjelenésének és működésének testreszabását.
- Webalapú konfigurációs és kezelési eszköz bevezetése, amely a `Web.config` fájlokat tartja karban.

A .NET 3.5 legfőbb webes újdonságai

Ahogy gondolhattuk, a .NET 3.5 tovább növelte az ASP.NET programozási modell hatókörét. Talán a legfontosabb, hogy ma már tartalmazza a következőket:

- Új vezérlőelemeket, amelyek támogatják a Silverlight-fejlesztést (emlékezzünk rá, hogy ez egy WPF-alapú API, amelyet a webhelyek gazdag médiatartalmának tervezésére használhatunk).
- Integrált támogatást az Ajax-stílusú fejlesztéshez, amely lényegében engedélyezi, hogy a „mikro-visszaküldések" a weboldal egy részét a lehető leggyorsabban frissítsék.

Mivel a könyv nem kizárólag a webes fejlesztésre összpontosít, az itt nem említett témákkal kapcsolatos további részletekért forduljunk a .NET Framework 3.5 dokumentációjához. Az igazság az, hogy ha minden szempontból megvizsgálnánk az ASP.NET-et, a könyv akár kétszer ilyen hosszú is lehetne. Biztosak lehetünk benne, hogy mire a könyv ezen szakaszának a végére érünk, szilárd ASP.NET-alapokkal rendelkezünk.

Megjegyzés Ha átfogóan szeretnénk tanulmányozni az ASP.NET-et, ehhez Matthew MacDonald *Pro ASP.NET 3.5 in C# 2008, Second Edition* (Apress, 2007) című könyvét ajánljuk.

Az ASP.NET-névterek

A .NET 3.5 megjelenése óta jóval több mint 30 webközpontú névtér létezik az alaposztály-könyvtárakban. Ezek a névterek az alábbi főbb kategóriába sorolhatók:

- alapvető funkcionalitás (például azok a típusok, amelyek révén kommunikálhatunk a HTTP-kérésekkel és -válaszokkal, Web Form infrastruktúra, téma- és profiltámogatás, webkijelzők , biztonság stb.),
- Web Form és HTML-vezérlőelemek,
- mobil webes fejlesztés,

- Silverlight-fejlesztés,
- Ajax-fejlesztés,
- XML-webszolgáltatások.

A 31.1. táblázat néhány alapvető ASP.NET-névteret ismertet.

Névterek	Jelentés
`System.Web`	Ez a névtér olyan típusokat definiál, amelyek engedélyezik a böngésző és a kiszolgáló közötti kommunikációt (mint például a kérési és reagálási lehetőségek, a sütik kezelése és a fájlátvitel).
`System.Web.Caching`	Ez a névtér olyan típusokat definiál, amelyek megkönnyítik a webalkalmazások gyorsítótárazási támogatását.
`System.Web.Hosting`	Ez a névtér olyan típusokat definiál, amelyek lehetővé teszik, hogy egyedi hosztokat hozzunk létre az ASP.NET-futtatókörnyezet számára.
`System.Web.Management`	Ez a névtér az ASP.NET-webalkalmazások állapotát kezelő és megfigyelő típusokat definiál.
`System.Web.Profile`	Ez a névtér az ASP.NET felhasználói profilokat megvalósító típusokat definiálja.
`System.Web.Security`	Ez a névtér olyan típusokat definiál, amelyek lehetővé teszik, hogy programozott módon gondoskodjunk a webhely biztonságáról.
`System.Web.SessionState`	Ez a névtér olyan típusokat definiál, amelyek lehetővé teszik az állapotmegőrző információk karbantartását felhasználónként (például munkamenetállapot-változók).
`System.Web.UI`, `System.Web.UI.WebControls`, `System.Web.UI.HtmlControls`	Ezek a névterek több olyan típust definiálnak, amelyek révén a webalkalmazásunk grafikus front endjét készíthetjük el.

31.1. táblázat: *Az ASP.NET alapvető webközpontú névterei*